一、编译四步骤：

1 编译预处理

读取 c 源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理
伪指令主要包括以下四个方面：
（1）**宏定义指令**，如#define Name TokenString，#undef 等。对于前一个伪指令，预编译所要做的是将程序中的所有 Name 用 TokenString 替换，但作为字符串常量的 Name 则不被替换。对于后者，则将取消对某个宏的定义，使以后该串的出现不再被替换。
（2）**条件编译指令**，如#ifdef，#ifndef，#else，#elif，#endif 等。这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉
（3） **头文件包含指令**，如#include "FileName"或者#include <FileName>等。在头文件中一般用伪指令#define 定义了大量的宏（最常见的是字符常量），同时包含有各种外部符号的声明。采用头文件的目的主要是为了使某些定义可以供多个不同的 C 源程序使用。因为在需要用到这些定义的 C 源程序中，只需加上一条#include 语句即可，而不必再在此文件中将这些定义重复一遍。预编译程序将把头文件中的定义统统都加入到它所产生的输出文件中，以供编译程序对之进行处理。包含到 c 源程序中的头文件可以是系统提供的，这些头文件一般被放在/usr/include 目录下。在程序中#include 它们要使用尖括号（<>）。另外开发人员也可以定义自己的头文件，这些文件一般与 c 源程序放在同一目录下，此时在#include 中要用双引号（""）。
（4）**特殊符号**，预编译程序可以识别一些特殊的符号。
例如在源程序中出现的 LINE 标识将被解释为当前行号（十进制数），FILE 则被解释为当前被编译的 C 源程序的名称。预编译程序对于在源程序中出现的这些串将用合适的值进行替换。预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。这个文件的含义同没有经过预处理的源文件是相同的，但内容有所不同。下一步，此输出文件将作为编译程序的输出而被翻译成为机器指令。

2. 编译、优化阶段

经过预编译得到的输出文件中，只有常量；如数字、字符串、变量的定义，以及 C 语言的关键字，如 main,if,else,for,while,{,}, +,-,*,\等等编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

优化处理是编译系统中一项比较艰深的技术。它涉及到的问题不仅同编译技术本身有关，而且同机器的硬件环境也有很大的关系。优化一部分是对中间代码的优化。这种优化不依赖于具体的计算机。另一种优化则主要针对目标代码的生成而进行的。对于前一种优化，主要的工作是删除公共表达式、循环优化（代码外提、强度削弱、变换循环控制条件、已知量的合并等）、复写传播，以及无用赋值的删除，等等。后一种类型的优化同机器的硬件结构密切相关，最主要的是考虑是如何充分利用机器的各个硬件寄存器

存放的有关变量的值，以减少对于内存的访问次数。另外，如何根据机器硬件执行指令的特点（如流水线、RISC、CISC、VLIW 等）而对指令进行一些调整使目标代码比较短，执行的效率比较高，也是一个重要的研究课题。经过优化得到的汇编代码必须经过汇编程序的汇编转换成相应的机器指令，方可能被机器执行。

3. 汇编过程

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个 C 语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。目标文件由段组成。通常一个目标文件中至少有两个段：代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。UNIX 环境下主要有三种类型的目标文件：

(1)可重定位文件，其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据

(2)共享的目标文件 这种文件存放了适合于在两种上下文里链接的代码和数据。 第一种是链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个 目标文件；第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。

(3)可执行文件，它包含了一个可以被操作系统创建一个进程来执行之的文件。汇编程序生成的实际上是第一种类型的目标文件。对于后两种还需要其他的一些处理方能得到，这个就是链接程序的工作了。

4. 链接程序

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。

例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号(如变量或者函数调用等);在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够诿操作系统装入执行的统一整体。

根据开发人员指定的同库函数的链接方式的不同，链接处理可分为两种：

(1)静态链接

在这种链接方式下，函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。

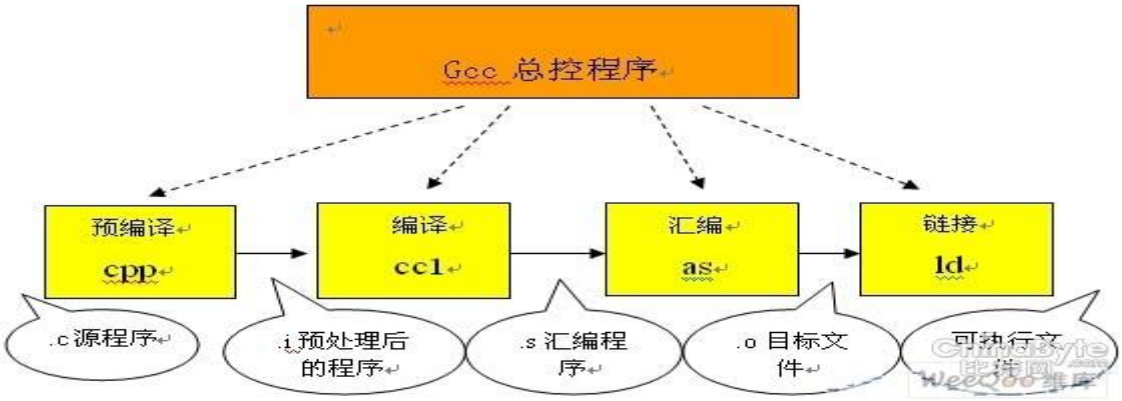
(2) 动态链接

在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将**被映射到运行**时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链

接要优越。在某些情况下动态链接可能带来一些性能上损害。

我们在 linux 使用的 gcc 编译器便是把以上的几个过程进行捆绑，使用户只使用一次命令就把编译工作完成，这的确方便了编译工作，但对于初学者了解编译过程就很不利了，下图便是 gcc 代理的编译过程：



从上图可以看到：

预编译

将.c 文件转化成 .i 文件

使用的 gcc 命令是：gcc －E

对应于预处理命令 cpp

编译

将.c/.h 文件转换成.s 文件

使用的 gcc 命令是：gcc －S

对应于编译命令 cc －S

汇编

将.s 文件转化成 .o 文件

使用的 gcc 命令是：gcc －c

对应于汇编命令是 as

链接

将.o 文件转化成可执行程序

使用的 gcc 命令是： gcc

对应于链接命令是 ld

二、I2C 和 SPI 的区别

1. IIC 总线

Inter-Integrated Circuit 串行总线的缩写，是 PHILIPS 公司推出的芯片间串行传输总线。它以 1 根串行数据线（SDA）和 1 根串行时钟线（SCL）实 现了双工的同步数据传输。具

有接口线少，控制方式简化，器件封装形式小，通信速率较高等优点。在主从通信中，可以有多个 I2C 总线器件同时接到 I2C 总线 上，通过地址来识别通信对象。

IIC 接口的协议里面包括设备地址信息，可以同一总线上连接多个从设备，通过应答来互通数据及命令。但是传输速率有限，标准模式下可达到 100Kbps，快速模式下可达到 400Kbps（我们开发板一般在 130Kbps），高速模式下达到 4Mbps，不能实现全双工，不适合传输很多的数据。

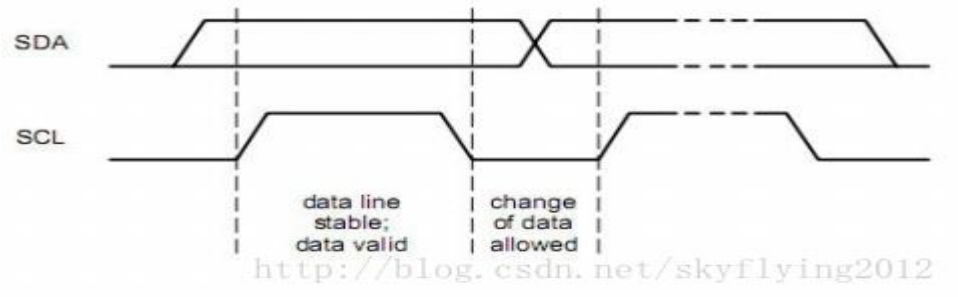
IIC 总线是一个真正的多主机总线，总线上多个主机初始化传输，可以通过传输检测和仲裁来防止数据被破坏 。

下来详细了解 IIC 总线时序：

1.1 总线数据有效性

IIC 总线是单工，因此同一时刻数据只有一个流向，因此采样有效时钟也是单一的，是在 SCL 时钟的高电平采样数据。

IIC 总线上 SDA 数据在 SCL 时钟低电平是可以发生变化，但是在时钟高电平时必须稳定，以便主从设备根据时钟采样数据，如下图：



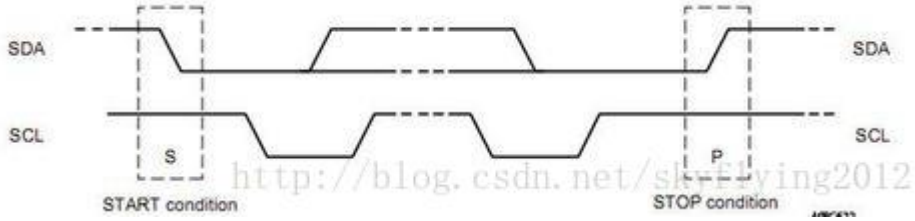
1.2 总线空闲条件

IIC 总线上设备都释放总线（发出传输停止）后，IIC 总线根据上拉电阻变成高电平，SDA SCL 都是高电平。

1.3 总线数据传输起始和结束条件

IIC 总线 SCL 高电平时 SDA 出现由高到低的跳变，标志总线上数据传输的开始条件

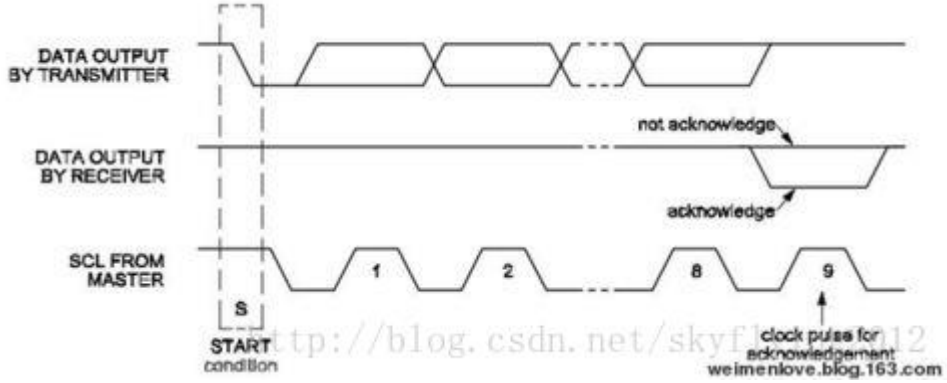
IIC 总线 SCL 高电平时 SDA 出现由低到高的跳变，标志总线上数据传输的结束条件



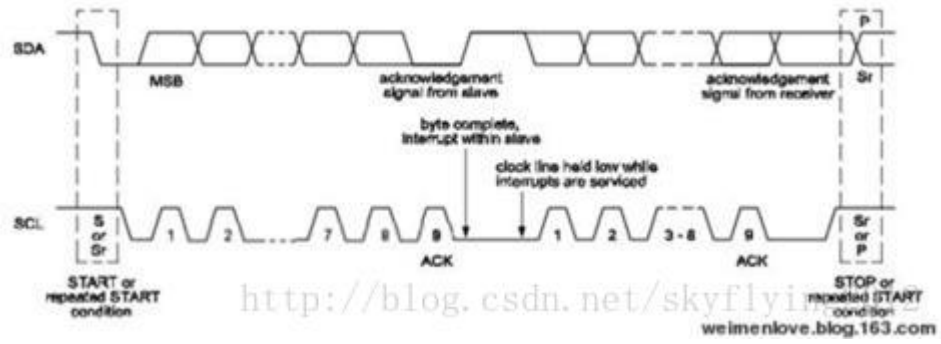
1.4 总线数据传输顺序以及 ACK 应答

IIC 总线上数据传输是 MSB 在前，LSB 在后，从示波器上看，从左向右依次读出数据即可

IIC 总线传输的数据不收限制，但是每次发到 SDA 上的必须是 8 位，并且主机发送 8 位后释放总线，从机收到数据后必须拉低 SDA 一个时钟，回应 ACK 表示数据接收成功，我们如果示波器上看到的波形就是每次 9 位数据，8bit+1bit ack。如下：



从机收到一字节数据后，如果需要一些时间处理，则会拉低 SCL，让传输进入等待状态，处理完成，释放 SCL，继续传输，如下：



1.5 总线读写时序

数据的传输在起始条件之后，发送一个 7 位的从机地址，紧接着第 8 位是数据方向（R/ W），0-表示发送数据（写），1-表示接收数据（读）。数据传输一般由主机产生的停止位（P）终止。但是如果主机仍希望在总线上通讯，它可以产生重复起始条件（Sr），和寻址另一个从机，而不是首先产生一个停止条件。在这种传输中，可能有不同的读/写格式结合。

IIC 总线主设备读写从设备，一般都是与从设备的寄存器打交道，这个可以通过阅读从设备的 datasheet 获取。总线写时序如下：

master start + master addr|w + slave ack + master reg|w + slave ack + master data + slave ack + master restart。。master data + slave nack + master stop

总线读时序如下：

master start + master addr|w + slave ack + master reg|w + slave ack + master restart + master addr|r + slave ack + slave data + master nack + master stop

总线读时序与写的不同之处在于读需要 2 次传输才能完成一次读取，首先要写寄存器地址到从设备，其实是写到了从设备的控制寄存器或者命令寄存器，从设备内部会根据这个地址来寻址所要操作的寄存器。

我在读我们的 bios 和内核时发现，2 者在总线读时序上的实现不太一样，在于第一次寄存器地址写入后，一个发的是 restart，一个发的是 stop，然后再 start 开始读取数据，示波器抓波形发现读取数据都正确，说明这 2 种时序都是正确的。

IIC 总线的读写时序比较固定，设备通信严格遵循协议，因此 iIC 总线设备驱动程序的编写也就相对简单一些。

主要应用的 iIc 总线设备有 touchscreen rtc 外扩 io 等

2. SPI 总线

SPI--Serial Peripheral Interface，串行外围设备接口，是 Motorola 公司推出的一种同步串行通讯方式，是一种三线同步总线，因其硬件功能很强，与 SPI 有关的软件就相当简单，使 CPU 有更多的时间处理其他事务。

SPI 通常有 SCK 时钟，STB 片选，DATA 数据信号三个信号。 I2C 通常有 SDA 数据和 SCL 时钟两个信号。SPI 总线真正实现了全双工数据传输，SPI 有 3 线跟 4 线两种，4 线的话，就是多了一条叫 SDC 的线，用来告知从设备现在传输的是数据还是指令。这个接口较快，可以传输较连续的数据。SPI 要想连接多个从设备，就需要给每个从设备配备一根片选信号。如果要可以实现全双工，也是需要多加一根数据线（MOSI MISO）。

也就是说 SPI 总线是通过片选来选择从设备。spi 总线速度要比 iic 要快，我们开发板最快能达到 30MHZ。

spi 总线特点：

1. 采用主-从模式(Master-Slave) 的控制方式

SPI 规定了两个 SPI 设备之间通信必须由主设备 (Master) 来控制次设备 (Slave). 一个 Master 设备可以通过提供 Clock 以及对 Slave 设备进行片选 (Slave Select) 来控制多个 Slave 设备, SPI 协议还规定 Slave 设备的 Clock 由 Master 设备通过 SCK 管脚提供给 Slave 设备, Slave 设备本身不能产生或控制 Clock, 没有 Clock 则 Slave 设备不能正常工作.

2. 采用同步方式(Synchronous)传输数据

Master 设备会根据将要交换的数据来产生相应的时钟脉冲(Clock Pulse), 时钟脉冲组成了时钟信号(Clock Signal) , 时钟信号通过时钟极性 (CPOL) 和 时钟相位 (CPHA) 控制着两个 SPI 设备间何时数据交换以及何时对接收到的数据进行采样, 来保证数据在两个设备之间是同步传输的.

3. 数据交换(Data Exchanges)

SPI 设备间的数据传输之所以又被称为数据交换, 是因为 SPI 协议规定一个 SPI 设备不能在数据通信过程中仅仅只充当一个 "发送者(Transmitter)" 或者 "接收者(Receiver)". 也就是说是全双工的, 在每个 Clock 周期内, SPI 设备都会发送并接收一个 bit 大小的数据, 相当于该设备有一个 bit 大小的数据被交换了.

一个 Slave 设备要想能够接收到 Master 发过来的控制信号, 必须在此之前能够被 Master 设备进行访问 (Access). 所以, Master 设备必须首先通过 SS/CS pin 对 Slave 设备进行片选, 把想要访问的 Slave 设备选上.

在数据传输的过程中, 每次接收到的数据必须在下一次数据传输之前被采样. 如果之前接收到的数据没有被读取, 那么这些已经接收完成的数据将有可能会被丢弃, 导致 SPI 物理模块最终失效. 因此, 在程序中一般都会在 SPI 传输完数据后, 去读取 SPI 设备里的数据, 即使这些数据(Dummy Data)在我们的程序里是无用的.

具体 spi 工作原理可以看博客另外一篇文章

SPI 和 IIC 是 2 种不同的通信协议，现在已经广泛的应用在 IC 之间的通信中。并且不少单片机已经整和了 SPI 和 IIC 的借口。但像 51 这种不支持 SPI 和 IIC 的单片机，也可以用模拟时钟的工作方式进行 SPI 和 IIC 的通信的。

下面主要总结一下 2 种总线的异同点:

1 iic 总线不是全双工，2 根线 SCL SDA。spi 总线实现全双工，4 根线 SCK CS MOSI MISO

2 iic 总线是多主机总线，通过 SDA 上的地址信息来锁定从设备。spi 总线只有一个主设备，主设备通过 CS 片选来确定从设备

3 iic 总线传输速度在 100kbps-4Mbps。spi 总线传输速度更快，可以达到 30MHZ 以上。

4 iic 总线空闲状态下 SDA SCL 都是高电平。spi 总线空闲状态 MOSI MISO 也都是 SCK 是有 CPOL 决定的

5 iic 总线 scl 高电平时 sda 下降沿标志传输开始，上升沿标志传输结束。spi 总线 cs 拉低标志传输开始，cs 拉高标志传输结束

6 iic 总线是 SCL 高电平采样。spi 总线因为是全双工，因此是沿采样，具体要根据 CPHA 决定。一般情况下 master device 是 SCK 的上升沿发送，下降沿采集

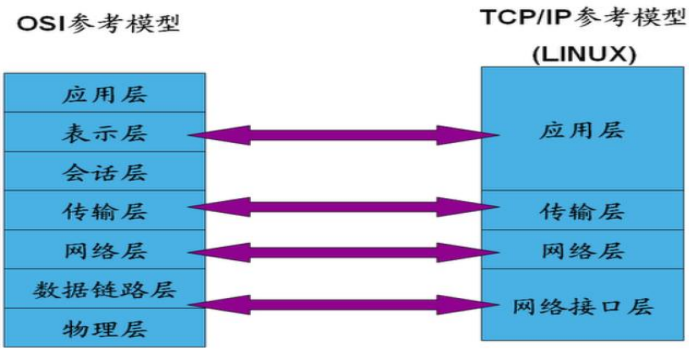
7 iic 总线和 spi 总线数据传输都是 MSB 在前，LSB 在后（串口是 LSB 在前）

8 iic 总线和 spi 总线时钟都是由主设备产生，并且只在数据传输时发出时钟

9 iic 总线读写时序比较固定统一，设备驱动编写方便。spi 总线不同从设备读写时序差别比较大，因此必须根据具体的设备 datasheet 来实现读写，相对复杂一些。

三、网络四层结构

TCP/IP 是一组用于实现网络互连的通信协议。Internet 网络体系结构以 TCP/IP 为核心。基于 TCP/IP 的参考模型将协议分成四个层次，它们分别是：链路层，网络层，运输层，应用层



1.应用层(应用层，表示层，会话层)

应用层对应于 OSI 参考模型的高层，为用户提供所需要的各种服务，例如：FTP、Telnet、DNS、SMTP 等.

2. 传输层(传输层)

传输层对应于 OSI 参考模型的传输层，为应用层实体提供端到端的通信功能，保证了数据包的顺序传送及数据的完整性。该层定义了两个主要的协议：传输控制协议（TCP）和用户数据报协议 (UDP).TCP 协议提供的是一种可靠的、通过“三次握手”来连接的数据传输服务；而 UDP 协议提供的则是不保证可靠的（并不是不可靠）、无连接的数据传输服务.

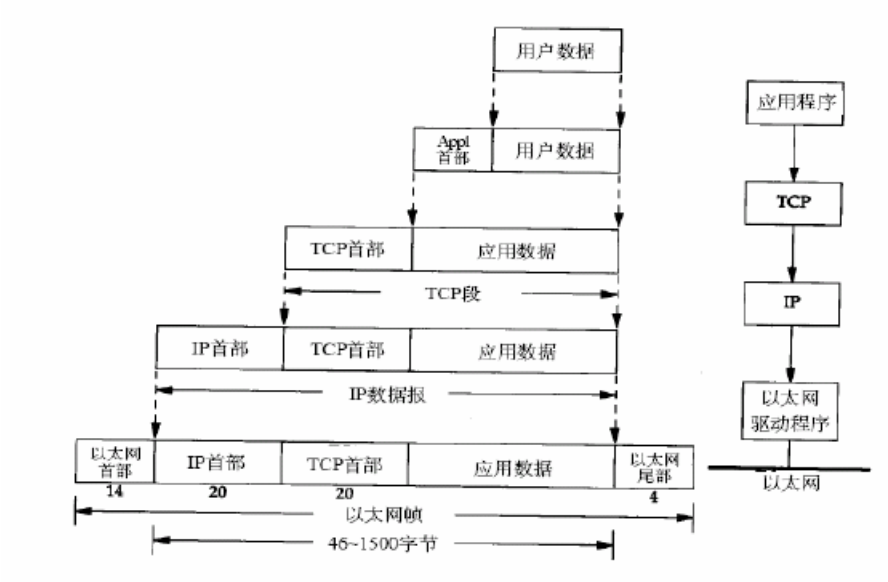
3. 网络层（网络层）

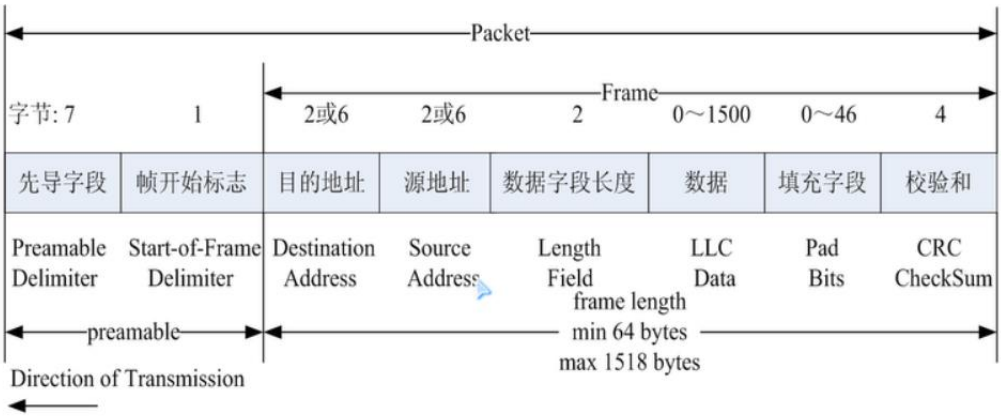
网际互联层对应于 OSI 参考模型的网络层，主要解决主机到主机的通信问题。它所包含的协议设计数据包在整个网络上的逻辑传输。注重重新赋予主机一个 IP 地址来完成对主机的寻址，它还负责数据包在多种网络中的路由。该层有三个主要协议：网际协议（IP）、互联网组管理协议（IGMP）和互联网控制报文协议（ICMP）。IP 协议是网际互联层最重要的协议，它提供的是一个可靠、无连接的数据报传递服务。

4. 网络接入层（链路层，物理层）

网络接入层与 OSI 参考模型中的物理层和数据链路层相对应。它负责监视数据在主机和网络之间的交换。事实上，TCP/IP 本身并未定义该层的协议，而由参与互连的各网络使用自己的物理层和数据链路层协议，然后与 TCP/IP 的网络接入层进行连接。地址解析协议（ARP）工作在此层，即 OSI 参考模型的数据链路层

以太网（数据链路层）头部封装了 IP 协议，IP 封装了传输层协议，TCP/UDP 封装了应用





TCP/IP 实际上一个协同工作的通信家族，为网络数据通信提供通路。为讨论方便可 TCP/IP 协议组大体上分为三部分：1.Internet 协议（IP）

2.传输控制协议（TCP）和用户数据报协议（UDP）

3.处于 TCP 和 UDP 之上的一组应用协议。它们包括：TELNET，文件传送协议(FTP)，域名服务（DNS）和简单的邮件传送程序（SMTP）等。

第一部分称为网络层。主要包括 Internet 协议（IP）、网际控制报文协议（ICMP）和地

址解析协议（ARP）：Internet 协议（IP）该协议被设计成互联分组交换通信网，以形成一个网际通信环境。它负责在源主机和目的地主机之间传输来自其较高层软件的称为数据报文的数据块，它在源和目的地之间提供非连接型传递服务。

网际控制报文协议（ICMP）

它实际上不是 IP 层部分，但直接同 IP 层一起工作，报告网络上的某些出错情况。允许网际路由器传输差错信息或测试报文。

地址解析协议（ARP）

ARP 实际上不是网络层部分，它处于 IP 和数据链路层之间，它是在 32 位 IP 地址和 48 位物理地址之间执行翻译的协议。

第二部分是传输层协议，包括传输控制协议和用户数据报文协议。

传输控制协议（TCP）：

该协议对建立网络上用户进程之间的对话负责，它确保进程之间的可靠通信，所提供的功能如下：1.　监听输入对话建立请求

2. 请求另一网络站点对话

3. 可靠的发送和接收数据

4. 适度的关闭对话

用户数据报文协议（UDP）：

UDP 提供不可靠的非连接型传输层服务，它允许在源和目的地之间传送数据，而不必在传送数据之前建立对话。它主要用于那些非连接型的应用程序，如：视频点播。

应用层：这部分主要包括 Telnet，文件传送协议(FTP 和 TFTP)，简单文件传送协议(SMTP)和域名服务（DNS）等协议。

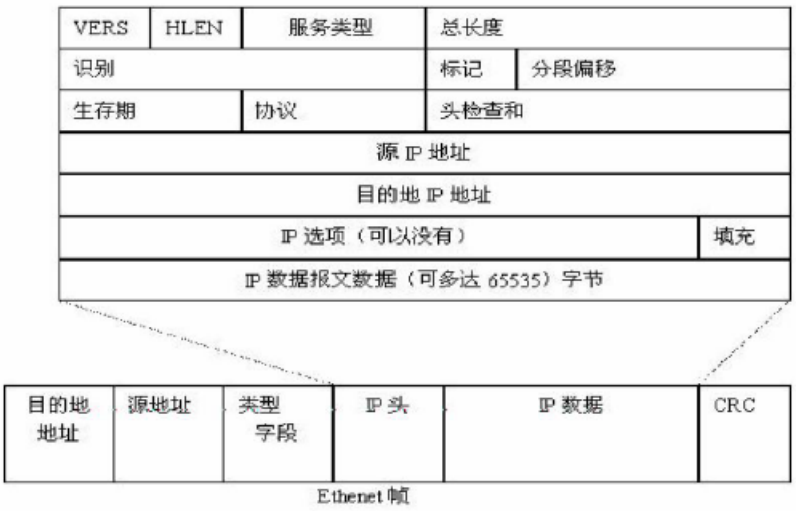
IP 主要有以下四个主要功能：数据传送，寻址，路由选择，数据报文的分段

IP 的主要目的是为数据输入/输出网络提供基本算法，为高层协议提供无连接的传送服务。这意味

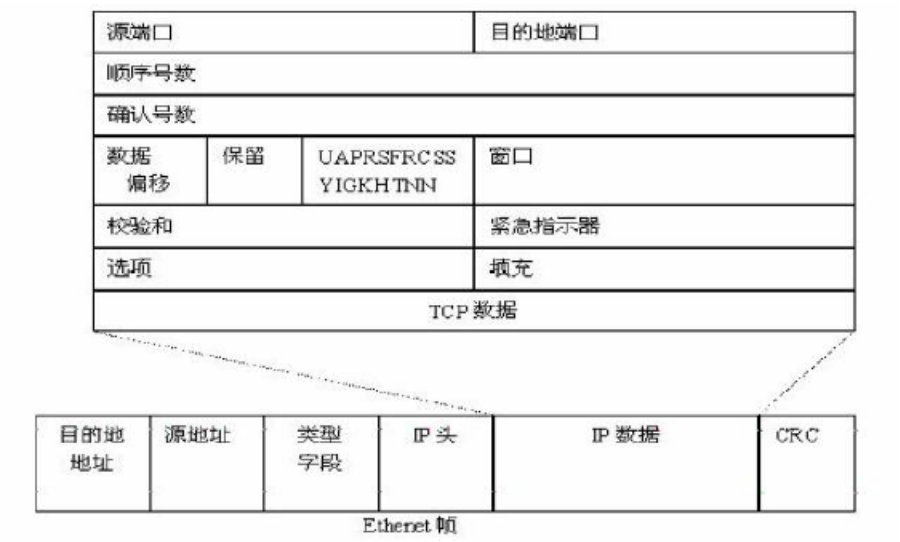
着在 IP 将数据递交给接收站点以前不在传输站点和接收站点之间建立对话。它只是封装和传递数据，但不向发送者或接收者报告包的状态，不处理所遇到的故障。

IP 包由 IP 协议头与协议数据两部分构成。

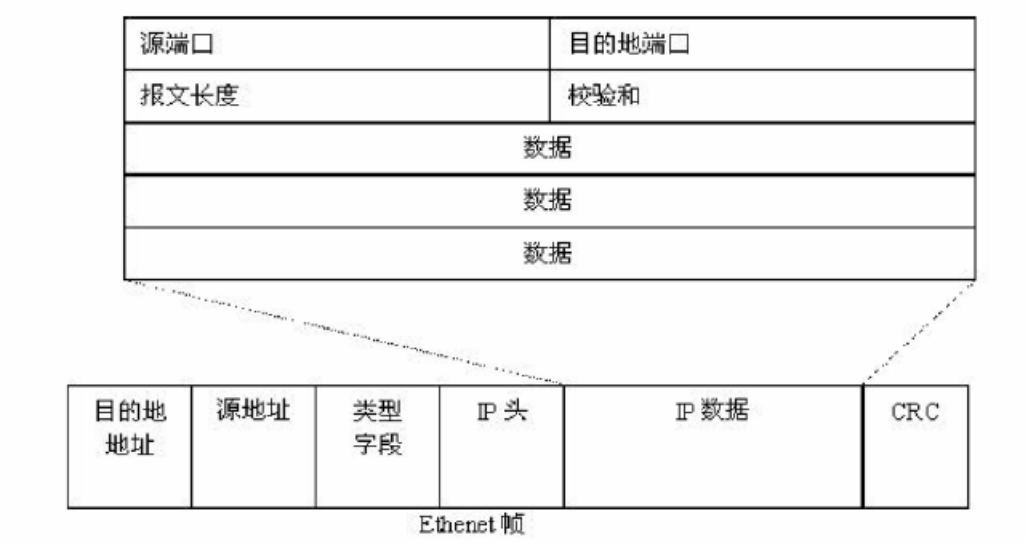
IP 协议头：



TCP 协议头：



Udp 协议头



Socket: Linux 中的网络编程通过 Socket(套接字)接口实现，Socket 是一种文件描述符。

套接字 socket 有三种类型：流式套接字（SOCK_STREAM）

流式的套接字可以提供可靠的、面向连接的通讯流。它使用了 TCP 协议。TCP 保证了数据传输的正确性和顺序性。

数据报套接字（SOCK_DGRAM）

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠，无差错,它使用数据报协议 UDP。

原始套接字:原始套接字允许对低层协议如 IP 或 ICMP 直接访问，主要用于新的网络协议的测试等。

地址结构:struct sockaddr

{

U_short sa_family;

Char sa_data[14];

}

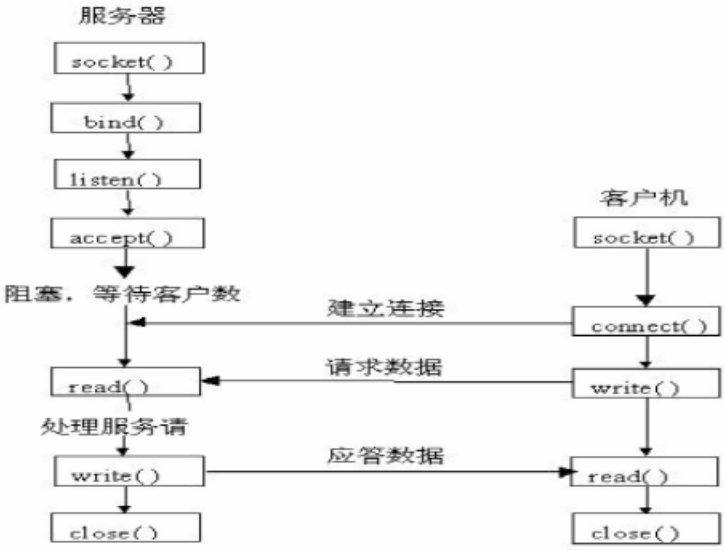
Sa_family:地址族，采用“AF_xxx”的形式，如：AF_INET

Sa_data：14 字节的特定协议地址

实际中：struct sockaddr_in{
short int sin_family; /* Internet 地址族 */
unsigned short int sin_port; /* 端口号 */
struct in_addr sin_addr; /* IP 地址 */unsigned char sin_zero[8]; /* 填 0 */
}
structin_addr
{

unsigned longs_addr; }
S_addr: 32 位的地址。
IP 地址通常由数字加点(192.168.0.1)的形式表示，而在 structin_addr 中使用的是 IP 地址是由 32 位的整数表示的，为了转换我们可以使用下面两个函数：
int inet_aton(constchar*cp,structin_addr*inp)
char* inet_ntoa(structin_addrin)
函数里面 a 代表 asciin 代表 network.第一个函数表示将 a.b.c.d 形式的 IP 转换为 32 位整数的 IP,存储在 inp 指针里面。第二个是将 32 位 IP 转换为 a.b.c.d 的格式。
网络字节顺序是 TCP/IP 中规定好的一种数据表示格式，它与具体的 CPU 类型、操作系统等无关，从而可以保证数据在不同主机之间传输时能够被正确解释。网络字节顺序采用 big endian 排序方式。
为什么要进行字节序转换?
例：INTEL 的 CPU 使用的小端字节序 MOTOROLA68k 系列 CPU 使用的是大端字节序 MOTOROLA 发一个 16 位数据 0X1234 给 INTEL, 传到 INTEL 时 ,就被 INTEL 解释为 0X3412 。
Htons：把 unsigned short 类型从主机序转换到网络序
Htonl：把 unsigned long 类型从主机序转换到网络序
以上是发送使用
Ntohs：把 unsigned short 类型从网络序转换到主机序
Ntohl：把 unsigned long 类型从网络序转换到主机序
使用函数：1.socket：创建一个 socket。
2.bind：用于绑定 IP 地址和端口号到 socket。
3.connect:该函数用于绑定之后的 client 端，与服务器建立连接。
4.listen: 设置能处理的最大连接要求,Listen()并未开始接收连线,只是设置 socket 为 listen 模式。
5.accept: 用来接受 socket 连接。
6.send:发送数据
7.recv：接收数据
网络编程：基于 TCP-服务器:
1. 创建一个 socket，用函数 socket()
2. 绑定 IP 地址、端口等信息到 socket 上，用函数 bind()
3. 设置允许的最大连接数，用函数 listen()
4. 接收客户端上来的连接，用函数 accept()
5. 收发数据，用函数 send()和 recv()，或者 read()和 write()
6. 关闭网络连接
基于 TCP-客户端:

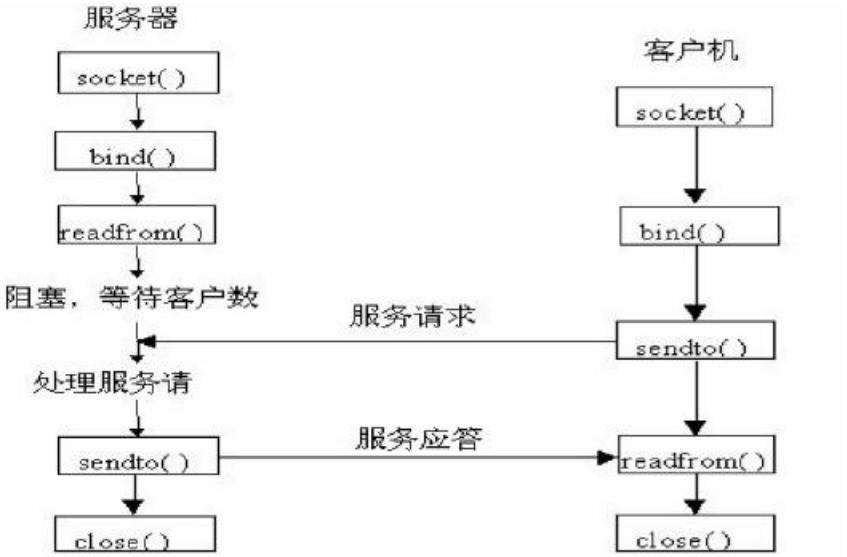
1. 创建一个 socket，用函数 socket()
2. 设置要连接的对方的 IP 地址和端口等属性
3. 连接服务器，用函数 connect()
4. 收发数据，用函数 send()和 recv()，或者 read()和 write()
5. 关闭网络连接基于 UDP-服务器



基于 UDP-服务器：

1. 创建一个 socket，用函数 socket()
2. 绑定 IP 地址、端口等信息到 socket 上，用函数 bind()
3. 循环接收数据，用函数 recvfrom()
4. 关闭网络连接

1. 创建一个 socket，用函数 socket()
2. 绑定 IP 地址、端口等信息到 socket 上，用函数 bind()
3. 设置对方的 IP 地址和端口等属性
4. 发送数据，用函数 sendto()
5. 关闭网络连接



在网络程序里面,一般来说都是许多客户对应一个服务器，为了处理客户的请求，对服务端的

程序就提出了特殊的要求。目前最常用的服务器模型有：

- 循环服务器:服务器在同一个时刻只可以响应一个客户端的请求
- 并发服务器:服务器在同一个时刻可以响应多个客户端的请求

UDP 循环服务器的实现方法:UDP 服务器每次从套接字上读取一个客户端的请求->处理->然后将结果返回给客户机。

因为 UDP 是非面向连接的,没有一个客户端可以老是占住服务端， 服务器对于每一个客户机的请求总是能够满足。

TCP 服务器接受一个客户端的连接,然后处理,完成了这个客户的所有请求后,断开连接。算法如下: TCP 循环服务器一次只能处理一个客户端的请求。只有在这个客户的所有请求都满足后，服务器才可以继续后面的请求。这样如果有一个客户

端占住服务器不放时,其它的客户机都不能工作了，因此,TCP 服务器一般很少用循环服务器模型的。

TCP 并发服务器: 并发服务器的思想是每一个客户机的请求并不由服务器直接处

理,而是由服务器创建一个 子进程来处理。算法如下:

TCP 并发服务器可以解决 TCP 循环服务器客户机独占服务器的情况。但同时也带来了问题：为了响应客户的请求,服务器要创建子进程来处理，而创建子进程是一种非常消耗资源的操作

阻塞函数在完成其指定的任务以前不允许程序继续向下执行。例如：当服务器运行到 accept 语句时，而没有客户请求连接，服务器就会停止在 accept 语句上等待连接请求的到来。这种情况称为阻塞（blocking），而非阻塞操作则可以立即完成。例如，如果你希望服务器仅仅检查是否有客户在等待连接，有就接受连接，否则就继

续做其他事情，则可以通过使用 select 系统调用来实现。除此之外，select 还可以同时监视多个套接字

```
int select(int maxfd, fd_set *readfds, fd_set *writefds, fe_set
                *exceptfds, const struct timeval *timeout)
```

Maxfd: 文件描述符的范围，比待检的最大文件描述符大 1

Readfds:被读监控的文件描述符集

Writefds:被写监控的文件描述符集

Exceptfds:被异常监控的文件描述符集

Timeout:定时器

Timeout 取不同的值，该调用有不同的表现：

Timeout 值为 0，不管是否有文件满足要求，都立刻返回，无文件满足要求返回 0，有文件满足要求返回一个正值。

Timeout 为 NULL，select 将阻塞进程，直到某个文件,满足要求

Timeout 值为正整数，就是等待的最长时间，即 select 在 timeout 时间内阻塞进程

Select 调用返回时，返回值有如下情况：

1. 正常情况下返回满足要求的文件描述符个数；
2. 经过了 timeout 等待后仍无文件满足要求，返回值为 0；
3. 如果 select 被某个信号中断，它将返回-1 并设置 errno 为 EINTR。
4. 如果出错，返回-1 并设置相应的 errno。

1. 设置要监控的文件
2. 调用 Select 开始监控
3. 判断文件是否发生变化

系统提供了 4 个宏对描述符集进行操作：

#include <sys/select.h>
void FD_SET(intfd, fd_set*fdset)
void FD_CLR(intfd, fd_set*fdset)
void FD_ZERO(fd_set*fdset)
void FD_ISSET(intfd, fd_set*fdset)
宏 FD_SET 将文件描述符 fd 添加到文件描述符集 fdset 中；
宏 FD_CLR 从文件描述符集 fdset 中清除文件描述符 fd；
宏 FD_ZERO 清空文件描述符集 fdset；
在调用 select 后使用 FD_ISSET 来检测文件描述符集 fdset 中的文件 fd 发生了变化。

四、动态存储方式与静态动态存储方式

前面已经介绍了，从变量的作用域（即从空间）角度来分，可以分为全局变量和局部变量。从另一个角度，从变量值存在的作时间（即生存期）角度来分，可以分为静态存储方式和动态存储方式。静态存储方式：是指在程序运行期间分配固定的存储空间的方式。动态存储方式：是在程序运行期间根据需要进行动态的分配存储空间的方式。

用户存储空间可以分为三个部分：程序区；静态存储区；动态存储区。

全局变量全部存放在静态存储区，在程序开始执行时给全局变量分配存储区，程序行完毕就释放。在程序执行过程中它们占据固定的存储单元，而不动态地进行分配和释放。动态存储区存放以下数据：函数形式参数；自动变量（未加 static 声明的局部变量）；函数调用实的现场保护和返回地址。对以上这些数据，在函数开始调用时分配动态存储空间，函数结束时释放这些空间。在 C 语言中，每个变量和函数有两个属性：数据类型和数据的存储类别。

1 auto 变量

函数中的局部变量，如不专门声明为 static 存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数中的形参和在函数中定义的变量（包括在复合语句中定义的变量），都属此类，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。这类局部变量称为自动变量。自动变量用关键字 auto 作存储类别的声明。例如：int f(int a){ /* 定义 f 函数，a 为参数 */ auto int b,c=3; /*定义 b，c 自动变量*/ /* …… */}a 是形参，b，c 是自动变量，对 c 赋初值 3。执行完 f 函数后，自动释放 a，b，c 所占的存储单元。关键字 auto 可以省略，auto 不写则隐含定为“自动存储类别”，属于动态存储方式。

2 用 static 声明局部变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为“静态局部变量”，用关键字 static 进行声明。

【例 8-15】考察静态局部变量的值。

```
1. #include <stdio.h>
2.
3. int f(int a){
4.     auto int b=0;
```

```
    static int c=3;
    b=b+1;
    c=c+1;
    return (a+b+c);
}

int main(void){
    int a=2,i;
    for(i=0;i<3;i++)
        printf("%d\n",f(a));
      return 0;
}
```

对静态局部变量的说明：

1. 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占动态存储空间，函数调用结束后即释放。
2. 静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。
3. 如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0（对数值型变量）或空字符（对字符变量）。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

【例 8-16】打印 1 到 5 的阶乘值。

```
#include <stdio.h>

int fac(int n){
    static int f=1;
    f=f*n;
    return f;
}

int main(void){
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i,fac(i));
    return 0;
}
```

3 register 变量

为了提高效率，C 语言允许将局部变量得值放在 CPU 中的寄存器中，这种变量叫“寄存器变量”，用关键字 register 作声明。

【例 8-17】使用寄存器变量。

```
1.  #include <stdio.h>
2.
3.  int fac(int n){
4.      register int i,f=1;
5.      for(i=1;i<=n;i++)
6.          f=f*i;
7.      return f;
8.  }
9.
10. int main(void){
11.     int i;
12.     for(i=0;i<=5;i++)
13.         printf("%d!=%d\n",i,fac(i));
14.     return 0;
15. }
```

对寄存器变量的几点说明：

- 只有局部自动变量和形式参数可以作为寄存器变量；
- 一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量；
- 局部静态变量不能定义为寄存器变量。

4 用 extern 声明外部变量

外部变量（即全局变量）是在函数的外部定义的，它的作用域为从变量定义处开始，到本程序文件的末尾。如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件终了。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字 extern 对该变量作“外部变量声明”。表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

【例 8-18】用 extern 声明外部变量，扩展程序文件中的作用域。

纯文本复制

```
1.  #include <stdio.h>
2.
```

```
3.  int max(int x,int y){
4.      int z;
5.      z=x>y?x:y;
6.      return z;
7.  }
8.
9.  int main(void){
10.     extern A,B;
11.     printf("%d\n",max(A,B));
12.     return 0;
13. }
14.
15. int A=13, B=-8;
```

五、堆和栈的基本区别

0. 预备知识—程序的内存分配

一个由 C/C++编译的程序占用的内存分为以下几个部分

1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后由系统释放。

4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放

5、程序代码区—存放函数体的二进制代码。

二、例子程序

这是一个前辈写的，非常详细

```
//main.cpp
int  a  =  0;  全局初始化区
char  *p1;  全局未初始化区
main()
{
int  b;  栈
char  s[]  =  "abc";  栈
char  *p2;  栈
char  *p3  =  "123456";  123456/0 在常量区，p3 在栈上。
```

static int c =0; 全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20);
分配得来得 10 和 20 字节的区域就在堆区。
strcpy(p1, "123456"); 123456/0 放在常量区，编译器可能会将它与 p3 所指向的"123456" 优化成一个地方。
}

堆和栈的理论知识

1．申请方式

stack: 由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空 间
heap: 需要程序员自己申请，并指明大小，在 c 中 malloc 函数
如 p1 = (char *)malloc(10); 在 C++中用 new 运算符
如 p2 = new char[10]; 但是注意 p1、p2 本身是在栈中的。

2．申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。
堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

3．申请大小的限制

栈：在 Windows 下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有
的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。
堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

4．申请效率的比较

栈由系统自动分配，速度较快。但程序员是无法控制的。
堆是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.

另外，在 WINDOWS 下，最好的方式是用 VirtualAlloc 分配内存，他不是在堆，也不是在栈是
直接在进程的地址空间中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活。

5．堆和栈中的存储内容

栈：　在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可
执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈
的，然后是函数中的局部变量。注意静态变量是不入栈的。
当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。
堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

6.存取效率的比较

```
char  s1[]  =  "aaaaaaaaaaaaaa";
char  *s2  =  "bbbbbbbbbbbbbbbbb";
aaaaaaaaaa 是在运行时刻赋值的;
而 bbbbbbbbbb 是在编译时就确定的;
但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。
比如:
#include
void  main()
{
char  a  =  1;
char  c[]  =  "1234567890";
char  *p  ="1234567890";
a  =  c[1];
a  =  p[1];
return;
}
对应的汇编代码
10:  a  =  c[1];
00401067  8A  4D  F1  mov  cl,byte  ptr  [ebp-0Fh]
0040106A  88  4D  FC  mov  byte  ptr  [ebp-4],cl
11:  a  =  p[1];
0040106D  8B  55  EC  mov  edx,dword  ptr  [ebp-14h]
00401070  8A  42  01  mov  al,byte  ptr  [edx+1]
00401073  88  45  FC  mov  byte  ptr  [ebp-4],al
第一种在读取时直接就把字符串中的元素读到寄存器 cl 中，而第二种则要先把指针值读到
edx 中，再根据 edx 读取字符，显然慢了。
```

7.小结

堆和栈的区别可以用如下的比喻来看出：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

六、进程和线程的区别：

程序：静态实体代码

进程：正在运行的程序；有独立的用户空间

进程（process）和线程（thread）是操作系统的基本概念，但是它们比较抽象，不容易掌握。最近，我读到一篇材料，发现有一个很好的类比，可以把它们解释地清晰易懂。

1.计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。

2.假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务。

3.进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。

4.一个车间里，可以有很多工人。他们协同完成一个任务。

5.线程就好比车间里的工人。一个进程可以包括多个线程。

6.车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。

7.可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。

8.一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。

9.还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。

10.这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做"信号量"（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex是semaphore的一种特殊情况（n=1时）。也就是说，完全可以用后者替代前者。但是，因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。

11.操作系统的设计，因此可以归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

七、指针和数组的区别

实际上关于数组与指针的区别这个问题在《C 专家编程》已经有很详细的阐释，但我想用自己的语言说一说我的理解。

数组是指针?

最近在做数据结构课设，其中一个函数发生了令人费解的错误，简化后的代码如下:

```
#include <stdio.h>

int main()

{

    char foo[] = "abcde";

    char **bar = &foo;

    printf("%c\n", *(*bar));

    return 0;

}
```

程序执行到 printf 语句后便会挂掉，调试时会提示一个 SIGSEGV 信号，根据原来的经验，这时程序试图访问本不应该访问的内存。

原来在 C 语言课堂上老师经常提到数组就是一个指针，指针也可以像数组那样用使用中括号的方式来进行内存访问。以这样的想法来分析前面的程序: foo 是一个字符指针，即 foo 的值即为“abcde”的首字符“a”的地址，*foo 即为 ‘a’；那么 foo 这个指针一定存在某个内存单元，&foo 获得这个内存单元的地址，即 pfoo 是指向 foo 的指针，那么 *pfoo 得到 foo，*(*pfoo)应该得到‘a’了；这样理解的话，程序是不应该有问题的。

下面我们使用指针代替数组来实现上面的程序:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

    char *foo = (char *)malloc(sizeof (char) * 2);
```

```
    *foo = 'a';
    *(foo + 1) = 0;
    char **pfoo = &foo;
    printf("%c\n", *(*pfoo));
    return 0;
}
```

程序这次运行结果和预料的相同，输出一个字母 a。由此可见，数组就是指针，这种说法是错误的。

数组是静态常量指针（static/Compile-time constant）？
有人认为数组是一个静态常量，即数组名代表一个静态的地址值，在编译时确定，下面代码可以证伪这种说法

```
int main()
{
    char foo[] = {'a'};
    static char *p = foo;
    return 0;
}
```

使用 gcc 编译时会有以下错误：

```
error: initializer element is not constant
```

可见数组名并不是代表一个静态量，并非地址常量。如果定义 foo 时加上 static 限定符，编译就会通过，此时数组名才代表了一个静态量。

数组是动态常量指针（const/Runtime constant）？
请看以下代码：

```
int main()
```

```
{
    char foo[] = {'a'};
    char * const bar;      //为什么是这种写法，请自行查阅相关资料
    char *baz;
    foo = baz;              /* 1 */
    bar = baz;              /* 2 */
    return 0;
}
```

gcc 编译时错误信息为：

```
/* 1 */ error: incompatible types when assigning to type 'char[1]' from type 'char *'
/* 2 */ error: assignment of read-only variable 'bar'
```

1 2 两处出错信息并不相同，若数组为动态常量指针，出错信息应像 2 那样。

数组是什么？

数组既不是静态常量，也不是指针，那么数组是什么？

左值和右值

首先补充一些左值和右值的知识，引用《C 专家编程》中的一段话：

出现在赋值符左边的符号有时被称为左值，出现在赋值符右边的符号有时被称为右值。编译器为每个变量分配一个地址（左值）。这个地址在编译时可知，而且该变量在运行时一直保存于这个地址。相反，存储于变量中的值（它的右值）只有在运行时才可知。如果需要用到变量中存储的值，编译器就发出指令从指定地址读入变量值并将它存于寄存器中。

我对左值的理解和书上有些区别，我把这里的“符号”称为“对象”，每一个符号都代表一个对象，对象与地址是一一对应的。即如果声明了 int a，那么 a 作为一个左值时，a 即代表这个保存在某个特定的地址的对象，对这个对象赋值即为把值放在这个特定的地址；a 作为右值时即代表 a 的内容，就是一个单纯的值，而不是对象。一个值是不能作为左值的，比如一个常数 1, 1 = a 这样的赋值语句是无法编译通过的。在我看来，“左值”义同“对象”，“右值”义同“值”，所以下面“左值”和“对象”指的是相同的东西。但是“左值”又有一个子集：“可修改的左值”，只有这个子集中的东西才能放在赋值号左边，因此我认为将引用中的第一句话修改为“出现在赋值符左边的符号有时被称为可修改

的左值”更能表达其实际的意思。为什么要引出这个子集，为的就是要把数组分出来，数组是左值，但并不是可修改的左值，因此你也不能直接把数组名放在等号左边进行赋值。

数组就是数组!

我先把结论放在这里，然后在进行分析：数组就是数组，一个数组名就代表一个数组对象，这个对象内可以有一个或多个元素，每个元素类型都相同；正如 int 就是 int，一个 int 变量名就代表一个 int 类型对象。看到这里，你可能要笑了，这不是什么都没说吗，谁不知道数组是这个意思啊，我想知道数组和指针什么关系。其实对数组的认识就是这样一个返璞归真过程，看我来慢慢解释。

以下代码：

```
/* 1.c */

int main()

{

    int foo[] = {1};

    int bar = 1;

    return 0;

}
```

使用 gcc 将其汇编并以 intel 格式输出汇编语言文件：

```
gcc -S -masm=intel 1.c
```

关键部分为：

```
mov     DWORD PTR [esp+8], 1

mov     DWORD PTR [esp+12], 1
```

esp+8 位置就是那个 int foo[]，esp+12 位置就是那个 int bar。可见，给 int 数组的赋值时就像给一个 int 变量赋值一样，并没用指针来进行间接访问，这个 int 数组对象 foo 的内存地址在编译时就确定了，是 esp+8；正如那个 int 对象 bar 一样，它的内存地址在编译时也确定了，是 esp+12。

以示区别，我将下面代码同样以汇编语言输出：

```
/* 2.c */

#include <stdlib.h>

int main()

{

    int *foo = (int *)malloc(sizeof (int));

    *foo = 1;

    return 0;

}
```

汇编的关键部分为：

```
mov     DWORD PTR [esp], 4

call    _malloc

mov     DWORD PTR [esp+28], eax

mov     eax, DWORD PTR [esp+28]

mov     DWORD PTR [eax], 1
```

前两句为 foo 分配内存空间，第三句将分配的内存空间地址值赋给 foo，foo 的地址为 esp+28，编译时已知。下面是赋值部分，首先从 foo 那里得到地址值，然后向这个地址赋值，这里可以看出和给数组赋值的差别，给数组赋值时是将值直接赋到了数组中，而不用从哪里得到数组的地址。

由上面可以看出，数组更像一个普通的变量，编译时就知道了其地址，可以直接赋值。

数组作为左值

数组不能放在赋值号左边，但数组仍可以作为一个左值或者说对象出现在语句中，一个重要的例子就是取地址操作：&。取地址操作 &的操作数必须是一个左值，而不能是一个右值。比如一个变量 int a = 1，&a 就可以得到 a 的地址，但 &1 是非法的，一个单纯的数值是没有地址的。那么对于一个 int foo[]，&foo 会返回一个什么样的值呢？自然是一个指向数组的指针咯，下面的程序可以看出来：

```
int  main()
```

```
{
    int foo[1];
    int bar[1];
    bar = &foo;      //故意触发一个 error
    return 0;
}
```

那个赋值语句一定会触发一个的错误，我们可以根据编译输出来确定它们的类型，错误为：

```
error: incompatible types when assigning to type 'int[1]' from type 'int (*)[1]'
```

没错，&foo 返回数据类型为 int (*)[1]，就是一个指向数组的指针。指向数组？指向数组的哪里呢？指向数组对象首地址，正如一个指向 int 对象的指针指向那个 int 对象占有的两个或四个内存单元的首地址一样。

把 &foo 赋给一个普通的指针是可以的，不过会触发一个 warning，因为 int * 与 int (*)[1] 并不相容。赋值后普通指针的值与 &foo 的值是相同的，都是数组对象的首地址，只是普通指针把这块内存当做 int 对象处理而已。

由于 C 语言是弱类型语言，你把 &foo 赋给 int **********bar 或者 int *baz 都是可以的，都不会导致 error，只会导致 warning，此时你打印出 *bar 或者 *baz 的值都是 foo 中第一个整数的值（前提是指针和数组占用空间大小相等）。正如文章开头的代码那样，以这个整数的值作为一个地址值进行间接访问（*(*bar)）就会导致非法访问的错误。

数组作为右值

数组作为右值时会发生什么？返回数组对象内的所有值自然不可能，因此 C 语言中采取的方法是数组作为右值时返回对象中元素类型的指针，指针指向第一个元素，类似上一个例子：

```
int main()
{
    int foo[1];
    int bar[1];
```

```
    bar = foo;      //故意触发一个 error
    return 0;
}
```

出错信息为:

```
error: incompatible types when assigning to type 'int[1]' from type 'int *'
```

foo 作为右值时返回了一个 int*，就是这个特性给人造成了数组就是指针的假象。

总结

数组作为左值和数组作为右值时的区别造成了无数人的困惑与误解：foo 作为右值时确实等价于一个指针，因为数组无法像普通对象那样返回它的值，它的元素可能有成百上千个，但作为一个左值时——比如作为取地址操作符的操作数时，数组就是作为一个数组对象而出现的，而不是指针，取地址返回一个指向数组的指针，而不是指向指针的指针。
一句话总结就是：数组就是数组，有着自己的特性

八、大小端的区别

最近在学习 USB，在看 Keil C51 代码的时候发现从 PC 机接收的 USB 数据在 Keil C51 环境里要交换高低字节，这是因为 Keil 的数据结构是大端模式，对于大端模式不是很清楚后来网上搜索发现有一篇文章介绍的比较详细，不敢独享贴出来大家学习。　　所谓的大端模式，是指数据的低位（就是权值较小的后面那几位）保存在内存的高地址中，而数据的高位，保存在内存的低地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

所谓的小端模式，是指数据的低位保存在内存的低地址中，而数 据的高位保存在内存的高地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致。

为什么会有大小端模式之分呢？这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在 C 语言中除了 8bit 的 char 之外，还有 16bit 的 short 型，32bit 的 long 型（要看具体的编译器），另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如果将多个字节安排的问题。因此就导致了大端存储模式和小 端存储模式。例如一个 16bit 的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么 0x11 为高字节，0x22 为低字节。对于 大端模式，就将 0x11 放在低地址中，即 0x0010 中，0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。我们常用的 X86 结构是小端模 式，而 KEIL C51 则为大端模式。很多的 ARM，DSP 都为小端模式。有些 ARM 处理器还可以由硬件来选择是大端模式还是小端模式。所有网络都是采用大端模式

下面这段代码可以用来测试一下你的编译器是大端模式还是小端模式： short int x;
char x0,x1;
x=0x1122;
x0=((char*)&x)[0]; //低地址单元
x1=((char*)&x)[1]; //高地址单元
若 x0=0x11,则是大端; 若 x0=0x22,则是小端.....
上面的程序还可以看出，数据寻址时，用的是低位字节的地址。

一.什么是字节对齐,为什么要对齐?
现代计算机中内存空间都是按照 byte 划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特
定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。
对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的 CPU 在访问
一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对
数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个 int 型（假设为 32 位系统）如果存放在偶地址开始的地方，那
么一个读周期就可以读出这 32bit，而如果存放在奇地址开始的地方，就需要 2 个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该 32bit 数
据。显然在读取效率上下降很多。

九、变量定义和变量声明的区别

1.定义：
为变量分配存储空间，还可以指定初始值
有且只有一个定义
2.声明：
表明变量的类型和名字
定义也是声明，当定义变量时声明了他的类型和名字。
一般为了方便，把建立存储空间的声明称定义，而不把建立存储空间的声明称声明。

十、FCLK，HCLK 和 PCLK 时钟三者之间的关系

FCLK 是提供给 ARM920T 的时钟。HCLK 是提供给用于 ARM920T 存储器控制器，中断控制器，LCD 控制器，DMA 和 USB 主机模块的 AHB 总线的时钟。PCLK 是提供给用于外设如 WDT，IIS，I2C，PWM 定时器，MMC/SD 接口，ADC，UART，GPIO，RTC 和 SPI 的 APB 总线的时钟。 S3C2440A 还支持对 FCLK、 HCLK 和 PCLK 之间分频比例的选择。该比例由 CLKDIVN 控制寄存器中的 HDIVN 和 PDIVN 所决定。
普通模式：普通模式中，包括电源管理模块、CPU 核心、总线控制器、存储器控制器、中断

控制器、DMA 和外部主控在内的所有外设和基本模块完全可以运行。然而除基本模块外，提供给每个外设的时钟都可以由软件有选择的停止以降低功耗。
空闲模式：空闲模式中，停止了除总线控制器、存储器控制器、中断控制器、电源管理模块外的提供给 CPU 核心的时钟。要退出空闲模式，应当激活 EINT[23:0]或 RTC 闹钟中断或其它中断（开启 GPIO 模块前 EINT 不可用） 进入空闲模式 如果置位 CLKCON[2]为 1 来进入空闲模式，S3C2440A 将在一些延时后（直到电源控制逻辑收到 CPU 打包的 ACK 信号）进入空闲模式。 慢速模式 慢速模式中，可以应用慢时钟和排除来自 PLL 的功耗来降低功耗。CLKSLOW 控制寄存器中的 SLOW_VAL 和 CLKDIVN 控制寄存器决定了分频比例。
寄存器的说明：
LOCKTIME 0x4C000000 PLL 锁定时间计数寄存器
MPLLCON 0x4C000004 MPLL 配置寄存器
UPLLCON 0x4C000008 UPLL 配置寄存器

注意： 当你设置 MPLL 和 UPLL 的值时，你必须首先设置 UPLL 值再设置 MPLL 值。（大约需要 7 个 NOP 的间隔）MPLL 控制寄存器
Mpll = (2 × m × Fin) / (p × 2s
)
m = (MDIV + 8), p = (PDIV + 2), s = SDIV
UPLL 控制寄存器
Upll = (m × Fin) / (p × 2s
)
m = (MDIV + 8), p = (PDIV + 2), s = SDIV
CLKCON 0x4C00000C 时钟生成控制寄存器
CLKDIVN 0x4C000014 时钟分频控制寄存器 //1:4:8 时候设置成 5
CAMDIVN 0x4C000018 摄像头时钟分频寄存器

注意：
1. 应当谨慎设置 CLKDIVN，不要使其超过 HCLK 和 PCLK 的最小值。
2. 如果 HDIVN 不为 0，CPU 总线模式应该使用以下指令使其从快总线模式改变为异步总线模式（S3C2440
不支持同步总线模式）。

```
MMU_SetAsyncBusMode
MRC   p15,  0,  r0,  c1,  c0,  0
ORR   r0,   r0,  #R1_nF:OR:R1_iA
MCR   p15,  0,  r0,  c1,  c0,  0
```

如果 HDIVN 不为 0 并且 CPU 总线模式为快总线模式，CPU 运行在 HCLK。可以用此特性在不影响 HCLK 和
PCLK 的情况下改变 CPU 频率为一半或更多。

十一、存储器的区别

存储器（Memory）是计算机系统中的记忆设备，用来存放程序和数据。计算机中全部信息，包括输入的原始数据、计算机程序、中间运行结果和最终运行结果都保存在存储器中。它根据控制器指定的位置存入和取出信息。有了存储器，计算机才有记忆功能，才能保证正常工

作。

按用途存储器可分为主存储器（内存）和辅助存储器（外存），也有分为外部存储器和内部存储器的分类方法。
外存通常是磁性介质或光盘等，能长期保存信息。内存指主板上的存储部件，用来存放当前正在执行的数据和程序，但仅用于暂时存放程序和数据，关闭电源或断电，数据会丢失。

构成
构成存储器的存储介质，目前主要采用半导体器件和磁性材料。存储器中最小的存储单位就是一个双稳态半导体电路或一个 CMOS 晶体管或磁性材料的存储元，它可存储一个二进制代码。由若干个存储元组成一个存储单元，然后再由许多存储单元组成一个存储器。 一个存储器包含许多存储单元，每个存储单元可存放一个字节（按字节编址）。每个存储单元的位置都有一个编号，即地址，一般用十六进制表示。一个存储器中所有存储单元可存放数据的总和称为它的存储容量。假设一个存储器的地址码由 20 位二进制数(即 5 位十六进制数)组成，则可表示 2 的 20 次方，即 1M 个存储单元地址。每个存储单元存放一个字节，则该存储器的存储容量为 1MB。

分类按存储介质分
半导体存储器：用半导体器件组成的存储器。
磁表面存储器：用磁性材料做成的存储器。

按存储方式分
随机存储器:任何存储单元的内容都能被随机存取,且存取时间和存储单元的物理位置无关。
顺序存储器：只能按某种顺序来存取，存取时间和存储单元的物理位置有关。
按存储器的读写功能分
只读存储器(ROM)：存储的内容是固定不变的，只能读出而不能写入的半导体存储器。
随机读写存储器(RAM)：既能读出又能写入的半导体存储器。
按信息的可保存性分非永久记忆的存储器：断电后信息即消失的存储器。
永久记忆性存储器： 断电后仍能保存信息的存储器。
按存储器用途分
根据存储器在计算机系统中所起的作用，可分为主存储器、辅助存储器、高速缓冲存储器、控制存储器等。 为了解决对存储器要求容量大，速度快，成本低三者之间的矛盾，目前通常采用多级存储器体系结构，即使用高速缓冲存储器、主存储器和外存储器。

名称	用途	特点
高速缓冲存储器	Cache	高速存取指令和数据 存取速度快，但存储容量小
主存储器	内存	存放计算机运行期间的大量程序和数据 存取速度较快，存储容量不大
外存储器	外存	存放系统程序和大型数据文件及数据库 存储容量大，位成本低

功能

存储器功能	寻址方式	掉电后	说 明
随机存取存储器（RAM）	读、写	随机寻址	数据丢失
只读存储器（ROM）	读	随机寻址	数据不丢失

工作前写入数据

闪存（Flash Memory）	读、写	随机寻址	数据不丢失
先进先出存储器（FIFO）	读、写	顺序寻址	数据丢失
先进后出存储器（FILO）	读、写	顺序寻址	数据丢失

各类存储器

RAM

RAM(random access memory，随机存取存储器)。存储单元的内容可按需随意取出或存入，且存取的速度与存储单元的位置无关的存储器。这种存储器在断电时将丢失其存储内容，故主要用于存储短时间使用的程序。 按照存储信息的不同，随机存储器又分为静态随机存储器（Static RAM，SRAM)和动态随机存储器（Dynamic RAM，DRAM)。

SRAMS

RAM（Static RAM，静态随机存储器），不需要刷新电路，数据不会丢失，而且，一般不是行列地址复用的。但是他集成度比较低，不适合做容量大的内存，一般是用在处理器的缓存里面。像 S3C2440 的 ARM9 处理器里面就有 4K 的 SRAM 用来做 CPU 启动时用的。

SRAM 其实是一种非常重要的存储器，它的用途广泛。SRAM 的速度非常快，在快速读取和刷新时能够保持数据完整性。SRAM 内部采用的是双稳态电路的形式来存储数据。所以 SRAM 的电路结构非常复杂。制造相同容量的 SRAM 比 DRAM 的成本高的多。正因为如此，才使其发展受到了限制。因此目前 SRAM 基本上只用于 CPU 内部的一级缓存以及内置的二级缓存。仅有少量的网络服务器以及路由器上能够使用 SRAM。

DRAM

Dynamic RAM，动态随机存取存储器，每隔一段时间就要刷新一次数据，才能保存数据。而且是行列地址复用的，许多都有页模式。SDRAM 是其中的一种。

SDRAM

SDRAM（Synchronous DRAM，同步动态随机存储器），即数据的读写需要时钟来同步。其存储单元不是按线性排列的，是分页的。

DRAM 和 SDRAM 由于实现工艺问题，容量较 SRAM 大。但是读写速度不如 SRAM。

一般的嵌入式产品里面的内存都是用的 SDRAM。电脑的内存也是用的这种 RAM，叫 DDR SDRAM，其集成度非常高，因为是动态的，所以必须有刷新电路，每隔一段时间必须得刷新数据。

ROM

Read-Only Memory，只读存储器的总称。

在微机的发展初期，BIOS 都存放在 ROM（Read Only Memory，只读存储器）中。ROM 内部的资料是在 ROM 的制造工序中，在工厂里用特殊的方法被烧录进去的，其中的内容只能读不能改，一旦烧录进去，用户只能验证写入的资料是否正确，不能再作任何修改。如果发现资料有任何错误，则只有舍弃不用， 重新订做一份。ROM 是在生产线上生产的，由于成本高，一般只用在大批量应用的场合。

PROM

可编程只读存储器，只能写一次，写错了就得报废，现在用得很少了，好像那些成本比较低的 OPT 单片机里面用的就是这种存储器吧。

EPROM

EPROM（Erasable Programmable ROM，可擦除可编程 ROM）芯片可重复擦除和写入，解决了 PROM 芯片只能写入一次的弊端。

EPROM 芯片有一个很明显的特征，在其正面的陶瓷封装上，开有一个玻璃窗口，透过该窗口，可以看到其内部的集成电路，紫外线透过该孔照射内部芯片就可以擦除其内的数据，完成芯片擦除的操作要用到 EPROM 擦除器。

EPROM 内资料的写入要用专用的编程器，并且往芯片中写内容时必须要加一定的编程电压（VPP=12—24V,随不同的芯片型号而定）。EPROM 的型号是以 27 开头的,如 27C020(8*256K)是一片 2M Bits 容量的 EPROM 芯片。EPROM 芯片在写入资料后，还要以不透光的贴纸或胶布把窗口封住，以免受到周围的紫外线照射而使资料受损。 EPROM 芯片在空白状态时（用紫外光线擦除后），内部的每一个存储单元的数据都为 1（高电平）。

EEPROMEEPROM (Electrically Erasable Programmable ROM，电可擦可编程只读存储器)，一种掉电后数据不丢失的存储芯片。EEPROM 是可用户更改的只读存储器，其可通过高于普通电压的作用来擦除和重编程（重写），即可以在电脑上或专用设备上擦除已有信息并重新编程。不像 EPROM 芯片，EEPROM 不需从计算机中取出即可修改，是现在用得比较多的存储器，比如 24CXX 系列的 EEPROM。

在一个 EEPROM 中，当计算机在使用的时候是可频繁地重编程的，EEPROM 的寿命是一个很重要的设计考虑参数。

EEPROM 的一种特殊形式是闪存，其应用通常是个人电脑中的电压来擦写和重编程。

EEPROM 一般用于即插即用（Plug & Play），常用在接口卡中，用来存放硬件设置数据，也常用在防止软件非法拷贝的"硬件锁"上面。

闪存（Flash）

闪存(FLASH)是一种非易失性存储器，即断电数据也不会丢失。因为闪存不像 RAM（随机存取存储器）一样以字节为单位改写数据，因此不能取代 RAM。

闪存卡（Flash Card）是利用闪存（Flash Memory）技术达到存储电子信息的存储器，一般应用在数码相机，掌上电脑，MP3 等小型数码产品中作为存储介质，所以样子小巧，有如一张卡片，所以称之为闪存卡。根据不同的生产厂商和不同的应用，闪存卡大概有 U 盘、SmartMedia（SM 卡）、Compact Flash（CF 卡）、MultiMediaCard（MMC 卡）、Secure Digital（SD 卡）、Memory Stick（记忆棒）、XD-Picture Card（XD 卡）和微硬盘（MICRODRIVE）。这些闪存卡虽然外观、规格不同，但是技术原理都是相同的。

NAND FLASH 和 NOR FLASH 都是现在用得比较多的非易失性闪存。

设计实现

采用的并行接口,有独立的地址线和数据线,性能特点更像内存,是芯片内执行(XIP，eXecute In Place)，这样应用程序可以直接在 flash 闪存内运行，不必再把代码读到系统 RAM 中。

NAND 采用的是串行的接口，地址线和数据线是共用的 I/O 线，类似电脑硬盘。CPU 从里面读取数据的速度很慢,所以一般用 NAND 做闪存的话就必须把 NAND 里面的数据先读到内存里面，然后 CPU 才能够执行。但是它的集成度很高，成本很低。还有就是它的擦除速度也的 NOR 要快。其实 NAND 型闪存在设计之初确实考虑了与硬盘的兼容性，小数据块操作速度很慢，而大数据块速度就很快，这种差异远比其他存储介质大的多。这种性能特点非常值得我们留意性能对比　　flash 闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何 flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下,在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。

由于擦除 NOR 器件时是以 64～128KB 的块进行的，执行一个写入/擦除操作的时间为 5s,与此相反,擦除 NAND 器件是以 8～32KB 的块进行的,执行相同的操作最多只需要 4ms。

执行擦除时块尺寸的不同进一步拉大了 NOR 和 NADN 之间的性能差距，统计表明，

对于给定的一套写入操作(尤其是更新小文件时更多的擦除操作必须在基于 NOR 的单元中进行。这样，当选择存储解决方案时，设计师必须权衡以下的各项因素。

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。
- NAND 的擦除单元更小，相应的擦除电路更少。

接口差别　NOR flash 带有 SRAM 接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。

NAND 器件使用复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。

NAND 读和写操作采用 512 字节的块，这一点有点像硬盘管理此类操作，很自然地，基于 NAND 的存储器就可以取代硬盘或其他块设备。

容量和成本

NAND flash 的单元尺寸几乎是 NOR 器件的一半，由于生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

NOR flash 占据了容量为 1～16MB 闪存市场的大部分，而 NAND flash 只是用在 8～128MB 的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于数据存储，NAND 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

可靠性和耐用性

采用 flahs 介质时一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF 的系统来说，Flash 是非常合适的存储方案。可以从寿命(耐用性)、位交换和坏块处理三个方面来比较 NOR 和 NAND 的可靠性。

寿命(耐用性)

在 NAND 闪存中每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次。NAND 存储器除了具有 10 比 1 的块擦除周期优势，典型的 NAND 块尺寸要比 NOR 器件小 8 倍，每个 NAND 存储器块在给定的时间内的删除次数要少一些。

位交换

所有 flash 器件都受位交换现象的困扰。在某些情况下(很少见，NAND 发生的次数要比 NOR 多)，一个比特位会发生反转或被报告反转了。

一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。

当然，如果这个位真的改变了，就必须采用错误探测/错误更正(EDC/ECC)算法。位反转的问题更多见于 NAND 闪存，NAND 的供应商建议使用 NAND 闪存的时候，同时使用 EDC/ECC 算法。

这个问题对于用 NAND 存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用 EDC/ECC 系统以确保可靠性。

坏块处理

NAND 器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。

NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

易于使用

可以非常直接地使用基于 NOR 的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。

由于需要 I/O 接口，NAND 要复杂得多。各种 NAND 器件的存取方法因厂家而异。

在使用 NAND 器件时，必须先写入驱动程序，才能继续执行其他操作。向 NAND 器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在 NAND 器件上自始至终都必须进行虚拟映射。

软件支持

当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。

在 NOR 器件上运行代码不需要任何的软件支持，在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。

使用 NOR 器件时所需要的 MTD 要相对少一些，许多厂商都提供用于 NOR 器件的更高级软件，这其中包括 M-System 的 TrueFFS 驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian 和 Intel 等厂商所采用。

驱动还用于对 DiskOnChip 产品进行仿真和 NAND 闪存的管理，包括纠错、坏块处理和损耗平衡。

应用环境

NOR 型闪存现在的容量一般在 2M 左右，比较适合频繁随机读写的场合，通常用于存储程序代码并直接在闪存内运行，手机就是使用 NOR 型闪存的大户，所以手机的“内存”容量通常不大。另外用在代码量小的嵌入式产品方面，可以把 LINUX 操作系统剪裁到 2M 以内在其上面直接运行。

NAND 型闪存主要用来存储资料，我们常用的闪存产品，如闪存盘、数码存储卡、U 盘、MP3 等。另外用在那些要跑大型的操作系统的嵌入式产品上面，比如 LINUX 啊，WINCE 啊。当然也可以把 LINUX 操作系统剪裁到 2M 以内在 NOR Flash 上运行。但是很多时候，一个嵌入式产品里面，操作系统占的存储空间只是一小部分，大部分都是给用户跑应用程序的。就像电脑，硬盘都是几百 G，可是 WINDOWNS 操作系统所占的空间也不过几 G 而已。

总结：

简单地说，在计算机中，RAM 、ROM 都是数据存储器。RAM 是随机存取存储器，它的特点是易挥发性，即掉电失忆。ROM 通常指固化存储器(一次写入，反复读取)，它的特点与 RAM 相反。

ROM 又分一次性固化（PROM)、光擦除（EPROM）和电擦除（EEPROM）重写几种类型。举个例子来说也就是，如果突然停电或者没有保存就关闭了文件，那么 ROM 可以随机保存之前没有储存的文件但是 RAM 会使之前没有保存的文件消失。

RAM 又分为静态随机存储器（SRAM)和动态随机存储器（DRAM)。

问与答

问题 1：什么是 DRAM、SRAM、SDRAM？答：名词解释如下 DRAM--------动态随即存取器，需要不断的刷新，才能保存数据，而且是行列地址复用的，许多都有页模式 SRAM--------静态的随机存储器，加电情况下，不需要刷新，数据不会丢失，而且一般不是行列地址复用的 SDRAM-------同步的 DRAM，即数据的读写需要时钟来同步

问题 2：为什么 DRAM 要刷新，SRAM 则不需要？答：这是由 RAM 的设计类型决定的，DRAM 用了一个 T 和一个 RC 电路，导致电容会漏电和缓慢放电，所以需要经常刷新来保存数据

问题 3：我们通常所说的内存用的是什么呢？这三个产品跟我们实际使用有什么关系？答：内存（即随机存贮器 RAM）可分为静态随机存储器 SRAM，和动态随机存储器 DRAM 两种。我们经常说的“内存”是指 DRAM。而 SRAM 大家却接触的很少。

问题 4：为什么使用 DRAM 比较多、而使用 SRAM 却很少？答：1）因为制造相同容量的 SRAM 比 DRAM 的成本高的多，正因为如此，才使其发展受到了限制。因此目前 SRAM 基本上只用于 CPU 内部的一级缓存以及内置的二级缓存。仅有少量的网络服务器以及路由器上能够使用 SRAM。2）存储单元结构不同导致了容量的不同：一个 DRAM 存储单元大约需要一个晶体管和一个电容（不包括行读出放大器等），而一个 SRAM 存储单元大约需要六个晶体管。DRAM 和 SDRAM 由于实现工艺问题，容量较 SRAM 大，但是读写速度不如 SRAM。

问题 5：用得最多的 DRAM 有什么特点呢？它的工艺是什么情况？（通常所说的内存就是 DRAM）答：1）DRAM 需要进行周期性的刷新操作，我们不应将 SRAM 与只读存储器（ROM）和 Flash Memory 相混淆，因为 SRAM 是一种易失性存储器，它只有在电源保持连续供应的情况下才能够保持数据。“随机访问”是指存储器的内容可以以任何顺序访问，而不管前一次访问的是哪一个位置。2）DRAM 和 SDRAM 由于实现工艺问题，容量较 SRAM 大。但是读写速度不如 SRAM，但是现在，SDRAM 的速度也已经很快了，时钟好像已经有 150 兆的了。那么就是读写周期小于 10ns 了。3）SDRAM 虽然工作频率高，但是实际吞吐率要打折扣。以 PC133 为例，它的时钟周期是 7.5ns，当 CAS latency=2 时，它需要 12 个周期完成 8 个突发读操作，10 个周期完成 8 个突发写操作。不过，如果以交替方式访问 Bank，SDRAM 可以在每个周期完成一个读写操作（当然除去刷新操作）。4）其实现在的主流高速存储器是 SSRAM（同步 SRAM）和 SDRAM（同步 DRAM）。目前可以方便买到的 SSRAM 最大容量是 8Mb/片，最大工作速度是 166MHz；可以方便买到的 SDRAM 最大容量是 128Mb/片，最大工作速度是 133MHz。

问题 6：用得比较少但速度很快，通常用于服务器 cache 的 SRAM 有什么特点呢？答：1）SRAM 是静态的，DRAM 或 SDRAM 是动态的，静态的是用的双稳态触发器来保存信息，而动态的是用电子，要不时的刷新来保持。SRAM 是 Static Random Access Memory 的缩写，中文含义为静态随机访问存储器，它是一种类型的半导体存储器。“静态”是指只要不掉电，存储在 SRAM 中的数据就不会丢失。2）SRAM 其实是一种非常重要的存储器，它的用途广泛。SRAM 的速度非常快，在快速读取和刷新时能够保持数据完整性。SRAM 内部采用的是双稳态电路的形式来存储数据。所以 SRAM 的电路结构非常复杂。3）从晶体管的类型分，SRAM 可以分为双极性与 CMOS 两种。从功能上分，SRAM 可以分为异步 SRAM 和同步 SRAM（SSRAM）。异步 SRAM 的访问独立于时钟，数据输入和输出都由地址的变化控制。同步 SRAM 的所有访问都在时钟的上升/下降沿启动。地址、数据输入和其它控制信号均于时钟信号相关。最后要说明的一点： SRAM 不应该与 SDRAM 相混淆，SDRAM 代表的是同步 DRAM（Synchronous DRAM），这与 SRAM 是完全不同的。SRAM 也不应该与 PSRAM 相混淆，PSRAM 是一种伪装成 SRAM 的 DRAM。

十二、链接地址，加载地址等区别：

搞 ARM 开发时，在连接目标代码会提到运行地址和加载地址。这两者有什么区别呢?其次，网上也有说链接地址和存储地址，那么这四个地址之间有什么区别?

1、运行地址<--->链接地址：他们两个是等价的，只是两种不同的说法。

2、加载地址<--->存储地址：他们两个是等价的，也是两种不同的说法。

运行地址：程序在 SRAM、SDRAM 中执行时的地址。就是执行这条指令时，PC 应该等于这个地址，换句话说，PC 等于这个地址时，这条指令应该保存在这个地址内。

加载地址：程序保存在 Nand flash 中的地址。

位置无关码：B、BL、MOV 都是位置位置无关码。

位置有关码：LDR PC,=LABEL 等类似的代码都是位置有关码。

下面我们来看看一个 Makefile 文件

```
sdram.bin : head.S   leds.c
    arm-Linux-gcc   -c -o head.o head.S
    arm-linux-gcc -c -o leds.o leds.c
    arm-linux-ld -Ttext 0x30000000 head.o leds.o -o sdram_elf
    arm-linux-objcopy -O binary -S sdram_elf sdram.bin
    arm-linux-objdump -D -m arm   sdram_elf > sdram.dis
clean:
 rm -f    sdram.dis sdram.bin sdram_elf *.o
```

我们可以看到 sdram_elf 的代码段是从 0x30000000 地址开始存放，这个地址我们称之为运行地址。为什么从这个地址开始存放，因为 SDRAM 的起始地址是 0x30000000.

当我们从 Nand flash 启动时，硬件会自动将 Nand flash 前 4kB 代码拷贝到片内 SRAM 中，然后 CPU 从 SRAM 的 0x00000000 地址处开始执行程序。在这里我想纠正一个错误的观点，网上很多人都说是 CPU 自动把 Nand flash 前 4kB 代码拷贝到片内 SRAM 中，其实不然，是 Nand flash 控制器完成的，这个过程中 CPU 根本就没有参与 。

通过上面的 Makefile 文件，我们可以知道 bl disable_watch_dog 这条指令的运行地址是 0x30000000，但是它现在保存在 SRAM 的 0x00000000 的地址处，那么这条指令能够正确执行吗？of course，why?

因为这条指令 bl disable_watch_dog 是位置无关码，虽然它的运行地址是在 SDRAM 中的 0x30000000，但是它可以在 Steppingstone 中执行。这条指令的功能是跳转到标号

disable_watch_dog 处执行，它是一个相对跳转，PC=当前 PC 的值+偏移量 OFFSET。其中当前 PC 的值等于下两条指令的地址，通过反汇编可以看到下两条指令的地址为 0x0000 0008，而不是 0x3000 0008.因为现在指令是保存在 SRAM 中。
这条指令的机器码为 eb00 0005，将机器码低 24 位按符号位扩展成 32 位得到 0x0000 00005.然后将 0x0000 0005 左移 2 位得到 0x0000 0014。这个值就是偏移量 OFFSET=0X0000 0014。所以 PC=0X0000 0008+0X0000 0014=0X0000 001c.那么 CPU 就会到 SRAM 中的 0x0000 001c 地址处执行程序。

可以发现 bl 指令依赖当前 PC 的值，这个特性使得 bl 指令不依赖指令的运行地址。所以接下来的 bl mensetup ,bl cope_steppingstone_to_sdram 都能够执行。

接下来的 ldr pc,=on_sdram 是一条位置有关码，经过反汇编可以看到，它是当前 pc 的值+偏移量，得到一个地址 Addr，然后从内存中的这个地址去取数据 Data 赋给 pc。现在我们计算一下 Addr 这个地址。
Addr=当前 PC 的值+144。当前 PC 的值等于下两条指令的地址 0x0000 0014,而不是 0x3000 0014,因为现在程序是保存在 SRAM 中。所以 Addr=0x0000 0014+144=0x0000 0014+0x0000 0090=0x0000 00a4.那么这个地址 0x0000 00a4 中保存了什么数据。通过反汇编可以看到 SRAM 中的 0x0000 00a4 这个地址保存的机器码为 30000010，所以 cpu 会跳转到 0x30000010 这个地址处执行程序。这个地址 0x30000010
是在 SDRAM 中，这样程序就跳到 SDRAM 中去了。因为我们前面的指令 bl cope_stepping_to_sdram 已经把 SRAM 中 4kB 的程序拷贝到了 SDRAM 中，所以现在 SDRAM 中有程序了。

但是如果在程序的开头放置一条这样的指令：ldr pc,=disable_watch_dog,那么整个程序就不能够正确执行了。why?

我们先来看看启动代码

```
@******************************************************************************
@ File：head.S
@ 功能：设置 SDRAM，将程序复制到 SDRAM，然后跳到 SDRAM 继续执行
@******************************************************************************

.equ    MEM_CTL_BASE,    0x48000000
.equ    SDRAM_BASE,      0x30000000
```

```
.text
.global _start
_start:
    @bl   disable_watch_dog             @ 关闭 WATCHDOG，否则 CPU 会不断重启
   ldr pc, =disable_watch_dog
    bl   memsetup                       @ 设置存储控制器
    bl   copy_steppingstone_to_sdram    @ 复制代码到 SDRAM 中
    ldr pc, =on_sdram                   @ 跳到 SDRAM 中继续执行
on_sdram:
    ldr sp, =0x34000000                 @ 设置堆栈
    bl   main
halt_loop:
    b    halt_loop

disable_watch_dog:
    @ 往 WATCHDOG 寄存器写 0 即可
    mov r1,     #0x53000000
    mov r2,     #0x0
    str r2,     [r1]
    mov pc,     lr      @ 返回

copy_steppingstone_to_sdram:
    @ 将 Steppingstone 的 4K 数据全部复制到 SDRAM 中去
    @ Steppingstone 起始地址为 0x00000000，SDRAM 中起始地址为 0x30000000

    mov r1, #0
    ldr r2, =SDRAM_BASE
    mov r3, #4*1024
1:
    ldr r4, [r1],#4     @ 从 Steppingstone 读取 4 字节的数据，并让源地址加 4
    str r4, [r2],#4     @ 将此 4 字节的数据复制到 SDRAM 中，并让目地地址加 4
    cmp r1, r3          @ 判断是否完成：源地址等于 Steppingstone 的未地址?
    bne 1b              @ 若没有复制完，继续
    mov pc,     lr      @ 返回
```

```
memsetup:
    @ 设置存储控制器以便使用 SDRAM 等外设

    mov r1,     #MEM_CTL_BASE       @ 存储控制器的 13 个寄存器的开始地址
    adrl    r2, mem_cfg_val         @ 这 13 个值的起始存储地址
    add r3,     r1, #52             @ 13*4 = 54
1:
    ldr r4,     [r2], #4            @ 读取设置值，并让 r2 加 4
    str r4,     [r1], #4            @ 将此值写入寄存器，并让 r1 加 4
    cmp r1,     r3                  @ 判断是否设置完所有 13 个寄存器
    bne 1b                          @ 若没有写成，继续
    mov pc,     lr                  @ 返回


.align 4
mem_cfg_val:
    @ 存储控制器 13 个寄存器的设置值
    .long   0x22011110      @ BWSCON
    .long   0x00000700      @ BANKCON0
    .long   0x00000700      @ BANKCON1
    .long   0x00000700      @ BANKCON2
    .long   0x00000700      @ BANKCON3
    .long   0x00000700      @ BANKCON4
    .long   0x00000700      @ BANKCON5
    .long   0x00018005      @ BANKCON6
    .long   0x00018005      @ BANKCON7
    .long   0x008C07A3      @ REFRESH
    .long   0x000000B1      @ BANKSIZE
    .long   0x00000030      @ MRSRB6
    .long   0x00000030      @ MRSRB7
```

对应的反汇编代码

```
sdram_elf:     file format elf32-littlearm
```

```
Disassembly of section .text:

30000000 <_start>:
30000000:    e59ff09c    ldr    pc, [pc, #156]    ; 300000a4 <mem_cfg_val+0x34>
30000004:    eb000010    bl     3000004c <memsetup>
30000008:    eb000007    bl     3000002c <copy_steppingstone_to_sdram>
3000000c:    e59ff094    ldr    pc, [pc, #148]    ; 300000a8 <mem_cfg_val+0x38>

30000010 <on_sdram>:
30000010:    e3a0d30d    mov    sp, #872415232    ; 0x34000000
30000014:    eb000033    bl     300000e8 <main>

30000018 <halt_loop>:
30000018:    eafffffe    b      30000018 <halt_loop>

3000001c <disable_watch_dog>:
3000001c:    e3a01453    mov    r1, #1392508928    ; 0x53000000
30000020:    e3a02000    mov    r2, #0
30000024:    e5812000    str    r2, [r1]
30000028:    e1a0f00e    mov    pc, lr

3000002c <copy_steppingstone_to_sdram>:
3000002c:    e3a01000    mov    r1, #0
30000030:    e3a02203    mov    r2, #805306368    ; 0x30000000
30000034:    e3a03a01    mov    r3, #4096    ; 0x1000
30000038:    e4914004    ldr    r4, [r1], #4
3000003c:    e4824004    str    r4, [r2], #4
30000040:    e1510003    cmp    r1, r3
30000044:    1afffffb    bne    30000038 <copy_steppingstone_to_sdram+0xc>
30000048:    e1a0f00e    mov    pc, lr

3000004c <memsetup>:
3000004c:    e3a01312    mov    r1, #1207959552    ; 0x48000000
```

```
30000050:    e28f2018      add     r2, pc, #24
30000054:    e1a00000      nop                ; (mov r0, r0)
30000058:    e2813034      add     r3, r1, #52     ; 0x34
3000005c:    e4924004      ldr     r4, [r2], #4
30000060:    e4814004      str     r4, [r1], #4
30000064:    e1510003      cmp     r1, r3
30000068:    1afffffb      bne     3000005c <memsetup+0x10>
3000006c:    e1a0f00e      mov     pc, lr

30000070 <mem_cfg_val>:
30000070:    22011110      andcs     r1, r1, #4
30000074:    00000700      andeq     r0, r0, r0, lsl #14
30000078:    00000700      andeq     r0, r0, r0, lsl #14
3000007c:    00000700      andeq     r0, r0, r0, lsl #14
30000080:    00000700      andeq     r0, r0, r0, lsl #14
30000084:    00000700      andeq     r0, r0, r0, lsl #14
30000088:    00000700      andeq     r0, r0, r0, lsl #14
3000008c:    00018005      andeq     r8, r1, r5
30000090:    00018005      andeq     r8, r1, r5
30000094:    008c07a3      addeq     r0, ip, r3, lsr #15
30000098:    000000b1      strheq     r0, [r0], -r1
3000009c:    00000030      andeq     r0, r0, r0, lsr r0
300000a0:    00000030      andeq     r0, r0, r0, lsr r0
300000a4:    3000001c      andcc     r0, r0, ip, lsl r0
300000a8:    30000010      andcc     r0, r0, r0, lsl r0
300000ac:    e1a00000      nop                ; (mov r0, r0)

300000b0 <wait>:
300000b0:    e52db004      push     {fp}          ; (str fp, [sp, #-4]!)
300000b4:    e28db000      add     fp, sp, #0
300000b8:    e24dd00c      sub     sp, sp, #12
300000bc:    e50b0008      str     r0, [fp, #-8]
300000c0:    ea000002      b     300000d0 <wait+0x20>
300000c4:    e51b3008      ldr     r3, [fp, #-8]
300000c8:    e2433001      sub     r3, r3, #1
```

```
300000cc:    e50b3008      str    r3, [fp, #-8]
300000d0:    e51b3008      ldr    r3, [fp, #-8]
300000d4:    e3530000      cmp    r3, #0
300000d8:    1afffff9      bne    300000c4 <wait+0x14>
300000dc:    e28bd000      add    sp, fp, #0
300000e0:    e8bd0800      pop    {fp}
300000e4:    e12fff1e      bx     lr

300000e8 <main>:
300000e8:    e92d4800      push   {fp, lr}
300000ec:    e28db004      add    fp, sp, #4
300000f0:    e24dd008      sub    sp, sp, #8
300000f4:    e3a03000      mov    r3, #0
300000f8:    e50b3008      str    r3, [fp, #-8]
300000fc:    e59f3030      ldr    r3, [pc, #48]    ; 30000134 <main+0x4c>
30000100:    e3a02b55      mov    r2, #87040    ; 0x15400
30000104:    e5832000      str    r2, [r3]
30000108:    e59f0028      ldr    r0, [pc, #40]    ; 30000138 <main+0x50>
3000010c:    ebffffe7      bl     300000b0 <wait>
30000110:    e59f3024      ldr    r3, [pc, #36]    ; 3000013c <main+0x54>
30000114:    e3a02000      mov    r2, #0
30000118:    e5832000      str    r2, [r3]
3000011c:    e59f0014      ldr    r0, [pc, #20]    ; 30000138 <main+0x50>
30000120:    ebffffe2      bl     300000b0 <wait>
30000124:    e59f3010      ldr    r3, [pc, #16]    ; 3000013c <main+0x54>
30000128:    e3a02e1e      mov    r2, #480    ; 0x1e0
3000012c:    e5832000      str    r2, [r3]
30000130:    eafffff4      b      30000108 <main+0x20>
30000134:    56000010      undefined instruction 0x56000010
30000138:    00007530      andeq  r7, r0, r0, lsr r5
3000013c:    56000014      undefined instruction 0x56000014

Disassembly of section .ARM.attributes:

00000000 <.ARM.attributes>:
```

```
   0:   00002541    andeq   r2, r0, r1, asr #10
   4:   61656100    cmnvs   r5, r0, lsl #2
   8:   01006962    tsteq   r0, r2, ror #18
   c:   0000001b    andeq   r0, r0, fp, lsl r0
  10:   00543405    subseq  r3, r4, r5, lsl #8
  14:   01080206    tsteq   r8, r6, lsl #4
  18:   04120109    ldreq   r0, [r2], #-265  ; 0x109
  1c:   01150114    tsteq   r5, r4, lsl r1
  20:   01180317    tsteq   r8, r7, lsl r3
  24:   Address 0x00000024 is out of bounds.


Disassembly of section .comment:

00000000 <.comment>:
   0:   3a434347    bcc     10d0d24 <SDRAM_BASE-0x2ef2f2dc>
   4:   74632820    strbtvc r2, [r3], #-2080 ; 0x820
   8:   312d676e    teqcc   sp, lr, ror #14
   c:   312e362e    teqcc   lr, lr, lsr #12
  10:   2e342029    cdpcs   0, 3, cr2, cr4, cr9, {1}
  14:   00332e34    eorseq  r2, r3, r4, lsr lr
```

通过反汇编代码我们可以看到：第一条指令
ldr pc,=disable_watch_dog 对应的反汇编代码为 30000000: e59ff09c ldr pc, [pc, #156] ; 300000a4 <mem_cfg_val+0x34>

其中 pc=下两条指令的地址=0x0000 0008，立即数 156 对应的 16 进制为为 0x0000 009c，0x0000 0008+0x0000 009c=0x0000 00a4.而 SRAM 中的 0x0000 00a4 这个地址中保存的机器码为 3000001c，所以执行完这条指令后 pc=0x3000 001c，0x3000 001c 这个地址是 SDRAM 中的，而现在 SDRAM 中什么都没有，所以程序不能正确执行。

搞 ARM 开发时，在连接目标代码会提到运行地址和加载地址。这两者有什么区别呢?其次，网上也有说链接地址和存储地址，那么这四个地址之间有什么区别?

1、运行地址<--->链接地址：他们两个是等价的，只是两种不同的说法。

2、加载地址<--->存储地址：他们两个是等价的，也是两种不同的说法。

运行地址：程序在 SRAM、SDRAM 中执行时的地址。就是执行这条指令时，PC 应该等于这个地址，换句话说，PC 等于这个地址时，这条指令应该保存在这个地址内。
加载地址：程序保存在 Nand flash 中的地址。

位置无关码：B、BL、MOV 都是位置位置无关码。
位置有关码：LDR PC,=LABEL 等类似的代码都是位置有关码。

下面我们来看看一个 Makefile 文件

```
sdram.bin : head.S   leds.c
    arm-Linux-gcc   -c -o head.o head.S
    arm-linux-gcc -c -o leds.o leds.c
    arm-linux-ld -Ttext 0x30000000 head.o leds.o -o sdram_elf
    arm-linux-objcopy -O binary -S sdram_elf sdram.bin
    arm-linux-objdump -D -m arm   sdram_elf > sdram.dis
clean:
 rm -f    sdram.dis sdram.bin sdram_elf *.o
```

我们可以看到 sdram_elf 的代码段是从 0x30000000 地址开始存放，这个地址我们称之为运行地址。为什么从这个地址开始存放，因为 SDRAM 的起始地址是 0x30000000.

下面来看看一个启动代码

```
@******************************************************************************
@ File：head.S
@ 功能：设置 SDRAM，将程序复制到 SDRAM，然后跳到 SDRAM 继续执行
@******************************************************************************
.equ        MEM_CTL_BASE,       0x48000000
.equ        SDRAM_BASE,         0x30000000
.text
.global _start
_start:
   bl   disable_watch_dog            @ 关闭 WATCHDOG，否则 CPU 会不断重启
    bl   memsetup                    @ 设置存储控制器
    bl   copy_steppingstone_to_sdram     @ 复制代码到 SDRAM 中
   ldr pc, =on_sdram                 @ 跳到 SDRAM 中继续执行
```

```
on_sdram:
    ldr sp, =0x34000000              @ 设置堆栈
    bl   main
halt_loop:
    b    halt_loop
disable_watch_dog:
    @ 往 WATCHDOG 寄存器写 0 即可
    mov r1,     #0x53000000
    mov r2,     #0x0
    str r2,     [r1]
    mov pc,     lr      @ 返回
copy_steppingstone_to_sdram:
    @ 将 Steppingstone 的 4K 数据全部复制到 SDRAM 中去
    @ Steppingstone 起始地址为 0x00000000，SDRAM 中起始地址为 0x30000000

    mov r1, #0
    ldr r2, =SDRAM_BASE
    mov r3, #4*1024
1:
    ldr r4, [r1],#4     @ 从 Steppingstone 读取 4 字节的数据，并让源地址加 4
    str r4, [r2],#4     @ 将此 4 字节的数据复制到 SDRAM 中，并让目地地址加 4
    cmp r1, r3          @ 判断是否完成：源地址等于 Steppingstone 的末地址?
    bne 1b              @ 若没有复制完，继续
    mov pc,     lr      @ 返回
memsetup:
    @ 设置存储控制器以便使用 SDRAM 等外设
    mov r1,     #MEM_CTL_BASE       @ 存储控制器的 13 个寄存器的开始地址
    adrl    r2, mem_cfg_val         @ 这 13 个值的起始存储地址
    add r3,     r1, #52             @ 13*4 = 54
1:
    ldr r4,     [r2], #4            @ 读取设置值，并让 r2 加 4
    str r4,     [r1], #4            @ 将此值写入寄存器，并让 r1 加 4
    cmp r1,     r3                  @ 判断是否设置完所有 13 个寄存器
    bne 1b                          @ 若没有写成，继续
    mov pc,     lr                  @ 返回
```

```
.align 4
mem_cfg_val:
    @ 存储控制器 13 个寄存器的设置值
    .long   0x22011110      @ BWSCON
    .long   0x00000700      @ BANKCON0
    .long   0x00000700      @ BANKCON1
    .long   0x00000700      @ BANKCON2
    .long   0x00000700      @ BANKCON3
    .long   0x00000700      @ BANKCON4
    .long   0x00000700      @ BANKCON5
    .long   0x00018005      @ BANKCON6
    .long   0x00018005      @ BANKCON7
    .long   0x008C07A3      @ REFRESH
    .long   0x000000B1      @ BANKSIZE
    .long   0x00000030      @ MRSRB6
    .long   0x00000030      @ MRSRB7
```

下面来看看反汇编代码

```
sdram_elf:     file format elf32-littlearm


Disassembly of section .text:

30000000 <_start>:
30000000:    eb000005    bl    3000001c <disable_watch_dog>
30000004:    eb000010    bl    3000004c <memsetup>
30000008:    eb000007    bl    3000002c <copy_steppingstone_to_sdram>
3000000c:    e59ff090    ldr    pc, [pc, #144]    ; 300000a4 <mem_cfg_val+0x34>

30000010 <on_sdram>:
30000010:    e3a0d30d    mov    sp, #872415232    ; 0x34000000
30000014:    eb000033    bl    300000e8 <main>
```

```
30000018 <halt_loop>:
30000018:    eafffffe    b    30000018 <halt_loop>

3000001c <disable_watch_dog>:
3000001c:    e3a01453    mov    r1, #1392508928    ; 0x53000000
30000020:    e3a02000    mov    r2, #0
30000024:    e5812000    str    r2, [r1]
30000028:    e1a0f00e    mov    pc, lr

3000002c <copy_steppingstone_to_sdram>:
3000002c:    e3a01000    mov    r1, #0
30000030:    e3a02203    mov    r2, #805306368    ; 0x30000000
30000034:    e3a03a01    mov    r3, #4096    ; 0x1000
30000038:    e4914004    ldr    r4, [r1], #4
3000003c:    e4824004    str    r4, [r2], #4
30000040:    e1510003    cmp    r1, r3
30000044:    1afffffb    bne    30000038 <copy_steppingstone_to_sdram+0xc>
30000048:    e1a0f00e    mov    pc, lr

3000004c <memsetup>:
3000004c:    e3a01312    mov    r1, #1207959552    ; 0x48000000
30000050:    e28f2018    add    r2, pc, #24
30000054:    e1a00000    nop            ; (mov r0, r0)
30000058:    e2813034    add    r3, r1, #52    ; 0x34
3000005c:    e4924004    ldr    r4, [r2], #4
30000060:    e4814004    str    r4, [r1], #4
30000064:    e1510003    cmp    r1, r3
30000068:    1afffffb    bne    3000005c <memsetup+0x10>
3000006c:    e1a0f00e    mov    pc, lr

30000070 <mem_cfg_val>:
30000070:    22011110    andcs    r1, r1, #4
30000074:    00000700    andeq    r0, r0, r0, lsl #14
30000078:    00000700    andeq    r0, r0, r0, lsl #14
3000007c:    00000700    andeq    r0, r0, r0, lsl #14
```

```
30000080:    00000700      andeq    r0, r0, r0, lsl #14
30000084:    00000700      andeq    r0, r0, r0, lsl #14
30000088:    00000700      andeq    r0, r0, r0, lsl #14
3000008c:    00018005      andeq    r8, r1, r5
30000090:    00018005      andeq    r8, r1, r5
30000094:    008c07a3      addeq    r0, ip, r3, lsr #15
30000098:    000000b1      strheq   r0, [r0], -r1
3000009c:    00000030      andeq    r0, r0, r0, lsr r0
300000a0:    00000030      andeq    r0, r0, r0, lsr r0
300000a4:    30000010      andcc    r0, r0, r0, lsl r0
300000a8:    e1a00000      nop              ; (mov r0, r0)
300000ac:    e1a00000      nop              ; (mov r0, r0)

300000b0 <wait>:
300000b0:    e52db004      push     {fp}          ; (str fp, [sp, #-4]!)
300000b4:    e28db000      add      fp, sp, #0
300000b8:    e24dd00c      sub      sp, sp, #12
300000bc:    e50b0008      str      r0, [fp, #-8]
300000c0:    ea000002      b        300000d0 <wait+0x20>
300000c4:    e51b3008      ldr      r3, [fp, #-8]
300000c8:    e2433001      sub      r3, r3, #1
300000cc:    e50b3008      str      r3, [fp, #-8]
300000d0:    e51b3008      ldr      r3, [fp, #-8]
300000d4:    e3530000      cmp      r3, #0
300000d8:    1afffff9      bne      300000c4 <wait+0x14>
300000dc:    e28bd000      add      sp, fp, #0
300000e0:    e8bd0800      pop      {fp}
300000e4:    e12fff1e      bx       lr

300000e8 <main>:
300000e8:    e92d4800      push     {fp, lr}
300000ec:    e28db004      add      fp, sp, #4
300000f0:    e24dd008      sub      sp, sp, #8
300000f4:    e3a03000      mov      r3, #0
300000f8:    e50b3008      str      r3, [fp, #-8]
```

```
300000fc:    e59f3030      ldr     r3, [pc, #48]     ; 30000134 <main+0x4c>
30000100:    e3a02b55      mov     r2, #87040     ; 0x15400
30000104:    e5832000      str     r2, [r3]
30000108:    e59f0028      ldr     r0, [pc, #40]     ; 30000138 <main+0x50>
3000010c:    ebfffe7       bl      300000b0 <wait>
30000110:    e59f3024      ldr     r3, [pc, #36]     ; 3000013c <main+0x54>
30000114:    e3a02000      mov     r2, #0
30000118:    e5832000      str     r2, [r3]
3000011c:    e59f0014      ldr     r0, [pc, #20]     ; 30000138 <main+0x50>
30000120:    ebffffe2      bl      300000b0 <wait>
30000124:    e59f3010      ldr     r3, [pc, #16]     ; 3000013c <main+0x54>
30000128:    e3a02e1e      mov     r2, #480     ; 0x1e0
3000012c:    e5832000      str     r2, [r3]
30000130:    eafffff4      b       30000108 <main+0x20>
30000134:    56000010      undefined instruction 0x56000010
30000138:    00007530      andeq   r7, r0, r0, lsr r5
3000013c:    56000014      undefined instruction 0x56000014
```

Disassembly of section .ARM.attributes:

```
00000000 <.ARM.attributes>:
   0:    00002541      andeq   r2, r0, r1, asr #10
   4:    61656100      cmnvs   r5, r0, lsl #2
   8:    01006962      tsteq   r0, r2, ror #18
   c:    0000001b      andeq   r0, r0, fp, lsl r0
  10:    00543405      subseq  r3, r4, r5, lsl #8
  14:    01080206      tsteq   r8, r6, lsl #4
  18:    04120109      ldreq   r0, [r2], #-265     ; 0x109
  1c:    01150114      tsteq   r5, r4, lsl r1
  20:    01180317      tsteq   r8, r7, lsl r3
  24:    Address 0x00000024 is out of bounds.
```

Disassembly of section .comment:

00000000 <.comment>:

```
   0:   3a434347    bcc     10d0d24 <SDRAM_BASE-0x2ef2f2dc>
   4:   74632820    strbtvc r2, [r3], #-2080    ; 0x820
   8:   312d676e    teqcc   sp, lr, ror #14
   c:   312e362e    teqcc   lr, lr, lsr #12
  10:   2e342029    cdpcs   0, 3, cr2, cr4, cr9, {1}
  14:   00332e34    eorseq  r2, r3, r4, lsr lr
```

当我们从 Nand flash 启动时，硬件会自动将 Nand flash 前 4kB 代码拷贝到片内 SRAM 中，然后 CPU 从 SRAM 的 0x00000000 地址处开始执行程序。在这里我想纠正一个错误的观点，网上很多人都说是 CPU 自动把 Nand flash 前 4kB 代码拷贝到片内 SRAM 中，其实不然，是 Nand flash 控制器完成的，这个过程中 CPU 根本就没有参与 。

通过上面的 Makefile 文件，我们可以知道 bl disable_watch_dog 这条指令的运行地址是 0x30000000，但是它现在保存在 SRAM 的 0x00000000 的地址处，那么这条指令能够正确执行吗？of course，why?

因为这条指令 bl disable_watch_dog 是位置无关码，虽然它的运行地址是在 SDRAM 中的 0x30000000，但是它可以在 Steppingstone 中执行。这条指令的功能是跳转到标号 disable_watch_dog 处执行，它是一个相对跳转，PC=当前 PC 的值+偏移量 OFFSET。其中当前 PC 的值等于下两条指令的地址，通过反汇编可以看到下两条指令的地址为 0x0000 0008，而不是 0x3000 0008.因为现在指令是保存在 SRAM 中。

这条指令的机器码为 eb00 0005，将机器码低 24 位按符号位扩展成 32 位得到 0x0000 00005.然后将 0x0000 0005 左移 2 位得到 0x0000 0014。这个值就是偏移量 OFFSET=0X0000 0014。所以 PC=0X0000 0008+0X0000 0014=0X0000 001c.那么 CPU 就会到 SRAM 中的 0x0000 001c 地址处执行程序。

可以发现 bl 指令依赖当前 PC 的值，这个特性使得 bl 指令不依赖指令的运行地址。所以接下来的 bl mensetup ,bl cope_steppingstone_to_sdram 都能够执行。

接下来的 ldr pc,=on_sdram 是一条位置有关码，经过反汇编可以看到，它是当前 pc 的值+偏移量，得到一个地址 Addr，然后从内存中的这个地址去取数据 Data 赋给 pc。现在我们计算一下 Addr 这个地址。

Addr=当前 PC 的值+144。当前 PC 的值等于下两条指令的地址 0x0000 0014,而不是 0x3000 0014,因为现在程序是保存在 SRAM 中。所以 Addr=0x0000 0014+144=0x0000 0014+0x0000 0090=0x0000 00a4.那么这个地址 0x0000 00a4 中保存了什么数据。通过反汇编可以看到 SRAM 中的 0x0000 00a4 这个地址保存的机器码为 30000010，所以 cpu 会跳转到 0x30000010 这个地址处执行程序。这个地址 0x30000010

是在 SDRAM 中，这样程序就跳到 SDRAM 中去了。因为我们前面的指令 bl cope_stepping_to_sdram 已经把 SRAM 中 4kB 的程序拷贝到了 SDRAM 中，所以现在 SDRAM 中有程序了。

但是如果在程序的开头放置一条这样的指令：ldr pc,=disable_watch_dog，那么整个程序就不能够正确执行了。why?

我们先来看看启动代码

```
@******************************************************************************
@ File：head.S
@ 功能：设置 SDRAM，将程序复制到 SDRAM，然后跳到 SDRAM 继续执行
@******************************************************************************

.equ       MEM_CTL_BASE,       0x48000000
.equ       SDRAM_BASE,         0x30000000

.text
.global _start
_start:
    @bl   disable_watch_dog             @ 关闭 WATCHDOG，否则 CPU 会不断重启
   ldr pc, =disable_watch_dog
    bl   memsetup                       @ 设置存储控制器
    bl   copy_steppingstone_to_sdram    @ 复制代码到 SDRAM 中
    ldr pc, =on_sdram                   @ 跳到 SDRAM 中继续执行
on_sdram:
    ldr sp, =0x34000000                 @ 设置堆栈
    bl   main
halt_loop:
    b    halt_loop

disable_watch_dog:
    @ 往 WATCHDOG 寄存器写 0 即可
    mov r1,     #0x53000000
    mov r2,     #0x0
    str r2,     [r1]
```

```
    mov pc,     lr      @ 返回

copy_steppingstone_to_sdram:
    @ 将 Steppingstone 的 4K 数据全部复制到 SDRAM 中去
    @ Steppingstone 起始地址为 0x00000000，SDRAM 中起始地址为 0x30000000

    mov r1, #0
    ldr r2, =SDRAM_BASE
    mov r3, #4*1024
1:
    ldr r4, [r1],#4     @ 从 Steppingstone 读取 4 字节的数据，并让源地址加 4
    str r4, [r2],#4     @ 将此 4 字节的数据复制到 SDRAM 中，并让目地地址加 4
    cmp r1, r3          @ 判断是否完成：源地址等于 Steppingstone 的未地址?
    bne 1b              @ 若没有复制完，继续
    mov pc,     lr      @ 返回

memsetup:
    @ 设置存储控制器以便使用 SDRAM 等外设

    mov r1,     #MEM_CTL_BASE       @ 存储控制器的 13 个寄存器的开始地址
    adrl    r2, mem_cfg_val         @ 这 13 个值的起始存储地址
    add r3,     r1, #52             @ 13*4 = 54
1:
    ldr r4,     [r2], #4            @ 读取设置值，并让 r2 加 4
    str r4,     [r1], #4            @ 将此值写入寄存器，并让 r1 加 4
    cmp r1,     r3                  @ 判断是否设置完所有 13 个寄存器
    bne 1b                          @ 若没有写成，继续
    mov pc,     lr                  @ 返回


.align 4
mem_cfg_val:
    @ 存储控制器 13 个寄存器的设置值
    .long   0x22011110      @ BWSCON
    .long   0x00000700      @ BANKCON0
```

```
    .long   0x00000700      @ BANKCON1
    .long   0x00000700      @ BANKCON2
    .long   0x00000700      @ BANKCON3
    .long   0x00000700      @ BANKCON4
    .long   0x00000700      @ BANKCON5
    .long   0x00018005      @ BANKCON6
    .long   0x00018005      @ BANKCON7
    .long   0x008C07A3      @ REFRESH
    .long   0x000000B1      @ BANKSIZE
    .long   0x00000030      @ MRSRB6
    .long   0x00000030      @ MRSRB7
```

对应的反汇编代码

```
sdram_elf:     file format elf32-littlearm


Disassembly of section .text:

30000000 <_start>:
30000000:    e59ff09c    ldr    pc, [pc, #156]    ; 300000a4 <mem_cfg_val+0x34>
30000004:    eb000010    bl     3000004c <memsetup>
30000008:    eb000007    bl     3000002c <copy_steppingstone_to_sdram>
3000000c:    e59ff094    ldr    pc, [pc, #148]    ; 300000a8 <mem_cfg_val+0x38>

30000010 <on_sdram>:
30000010:    e3a0d30d    mov    sp, #872415232    ; 0x34000000
30000014:    eb000033    bl     300000e8 <main>

30000018 <halt_loop>:
30000018:    eafffffe    b      30000018 <halt_loop>

3000001c <disable_watch_dog>:
3000001c:    e3a01453    mov    r1, #1392508928    ; 0x53000000
```

```
30000020:    e3a02000      mov     r2, #0
30000024:    e5812000      str     r2, [r1]
30000028:    e1a0f00e      mov     pc, lr

3000002c <copy_steppingstone_to_sdram>:
3000002c:    e3a01000      mov     r1, #0
30000030:    e3a02203      mov     r2, #805306368     ; 0x30000000
30000034:    e3a03a01      mov     r3, #4096     ; 0x1000
30000038:    e4914004      ldr     r4, [r1], #4
3000003c:    e4824004      str     r4, [r2], #4
30000040:    e1510003      cmp     r1, r3
30000044:    1afffffb      bne     30000038 <copy_steppingstone_to_sdram+0xc>
30000048:    e1a0f00e      mov     pc, lr

3000004c <memsetup>:
3000004c:    e3a01312      mov     r1, #1207959552     ; 0x48000000
30000050:    e28f2018      add     r2, pc, #24
30000054:    e1a00000      nop             ; (mov r0, r0)
30000058:    e2813034      add     r3, r1, #52     ; 0x34
3000005c:    e4924004      ldr     r4, [r2], #4
30000060:    e4814004      str     r4, [r1], #4
30000064:    e1510003      cmp     r1, r3
30000068:    1afffffb      bne     3000005c <memsetup+0x10>
3000006c:    e1a0f00e      mov     pc, lr

30000070 <mem_cfg_val>:
30000070:    22011110      andcs     r1, r1, #4
30000074:    00000700      andeq     r0, r0, r0, lsl #14
30000078:    00000700      andeq     r0, r0, r0, lsl #14
3000007c:    00000700      andeq     r0, r0, r0, lsl #14
30000080:    00000700      andeq     r0, r0, r0, lsl #14
30000084:    00000700      andeq     r0, r0, r0, lsl #14
30000088:    00000700      andeq     r0, r0, r0, lsl #14
3000008c:    00018005      andeq     r8, r1, r5
30000090:    00018005      andeq     r8, r1, r5
```

```
30000094:    008c07a3      addeq    r0, ip, r3, lsr #15
30000098:    000000b1      strheq    r0, [r0], -r1
3000009c:    00000030      andeq    r0, r0, r0, lsr r0
300000a0:    00000030      andeq    r0, r0, r0, lsr r0
300000a4:    3000001c      andcc    r0, r0, ip, lsl r0
300000a8:    30000010      andcc    r0, r0, r0, lsl r0
300000ac:    e1a00000      nop              ; (mov r0, r0)

300000b0 <wait>:
300000b0:    e52db004      push    {fp}          ; (str fp, [sp, #-4]!)
300000b4:    e28db000      add    fp, sp, #0
300000b8:    e24dd00c      sub    sp, sp, #12
300000bc:    e50b0008      str    r0, [fp, #-8]
300000c0:    ea000002      b    300000d0 <wait+0x20>
300000c4:    e51b3008      ldr    r3, [fp, #-8]
300000c8:    e2433001      sub    r3, r3, #1
300000cc:    e50b3008      str    r3, [fp, #-8]
300000d0:    e51b3008      ldr    r3, [fp, #-8]
300000d4:    e3530000      cmp    r3, #0
300000d8:    1afffff9      bne    300000c4 <wait+0x14>
300000dc:    e28bd000      add    sp, fp, #0
300000e0:    e8bd0800      pop    {fp}
300000e4:    e12fff1e      bx    lr

300000e8 <main>:
300000e8:    e92d4800      push    {fp, lr}
300000ec:    e28db004      add    fp, sp, #4
300000f0:    e24dd008      sub    sp, sp, #8
300000f4:    e3a03000      mov    r3, #0
300000f8:    e50b3008      str    r3, [fp, #-8]
300000fc:    e59f3030      ldr    r3, [pc, #48]    ; 30000134 <main+0x4c>
30000100:    e3a02b55      mov    r2, #87040    ; 0x15400
30000104:    e5832000      str    r2, [r3]
30000108:    e59f0028      ldr    r0, [pc, #40]    ; 30000138 <main+0x50>
3000010c:    ebffffe7      bl    300000b0 <wait>
```

```
30000110:    e59f3024      ldr     r3, [pc, #36]    ; 3000013c <main+0x54>
30000114:    e3a02000      mov     r2, #0
30000118:    e5832000      str     r2, [r3]
3000011c:    e59f0014      ldr     r0, [pc, #20]    ; 30000138 <main+0x50>
30000120:    ebffffe2      bl      300000b0 <wait>
30000124:    e59f3010      ldr     r3, [pc, #16]    ; 3000013c <main+0x54>
30000128:    e3a02e1e      mov     r2, #480    ; 0x1e0
3000012c:    e5832000      str     r2, [r3]
30000130:    eafffff4      b       30000108 <main+0x20>
30000134:    56000010      undefined instruction 0x56000010
30000138:    00007530      andeq   r7, r0, r0, lsr r5
3000013c:    56000014      undefined instruction 0x56000014
```

Disassembly of section .ARM.attributes:

```
00000000 <.ARM.attributes>:
   0:    00002541      andeq   r2, r0, r1, asr #10
   4:    61656100      cmnvs   r5, r0, lsl #2
   8:    01006962      tsteq   r0, r2, ror #18
   c:    0000001b      andeq   r0, r0, fp, lsl r0
  10:    00543405      subseq  r3, r4, r5, lsl #8
  14:    01080206      tsteq   r8, r6, lsl #4
  18:    04120109      ldreq   r0, [r2], #-265    ; 0x109
  1c:    01150114      tsteq   r5, r4, lsl r1
  20:    01180317      tsteq   r8, r7, lsl r3
  24:    Address 0x00000024 is out of bounds.
```

Disassembly of section .comment:

```
00000000 <.comment>:
   0:    3a434347      bcc     10d0d24 <SDRAM_BASE-0x2ef2f2dc>
   4:    74632820      strbtvc r2, [r3], #-2080    ; 0x820
   8:    312d676e      teqcc   sp, lr, ror #14
   c:    312e362e      teqcc   lr, lr, lsr #12
```

10: 2e342029 cdpcs 0, 3, cr2, cr4, cr9, {1}

14: 00332e34 eorseq r2, r3, r4, lsr lr

通过反汇编代码我们可以看到：第一条指令

ldr pc,=disable_watch_dog 对应的反汇编代码为 30000000: e59ff09c ldr pc, [pc, #156] ; 300000a4 <mem_cfg_val+0x34>

其中 pc=下两条指令的地址=0x0000 0008，立即数 156 对应的 16 进制为为 0x0000 009c，0x0000 0008+0x0000 009c=0x0000 00a4.而 SRAM 中的 0x0000 00a4 这个地址中保存的机器码为 3000001c，所以执行完这条指令后 pc=0x3000 001c，0x3000 001c 这个地址是 SDRAM 中的，而现在 SDRAM 中什么都没有，所以程序不能正确执行。

十三：阻塞与非阻塞的区别

阻塞”与"非阻塞"与"同步"与“异步"不能简单的从字面理解，提供一个从分布式系统角度的回答。

1.同步与异步

同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)

所谓同步，就是在发出一个*调用*时，在没有得到结果之前，该*调用*就不返回。但是一旦调用返回，就得到返回值了。

换句话说，就是由*调用者*主动等待这个*调用*的结果。

而异步则是相反，*调用*在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在*调用*发出后，*被调用者*通过状态、通知来通知调用者，或通过回调函数处理这个调用。

典型的异步编程模型比如 Node.js

举个通俗的例子：

你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下"，然后开始查啊查，等查好了（可能是 5 秒，也可能是一天）告

诉你结果（返回结果）。
而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

2. 阻塞与非阻塞
阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

还是上面的例子，
你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟 check 一下老板有没有返回结果。
在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

十四、poll 或 select 机制：

在编写驱动程序的过程当中我们可以使用 poll 机制来非阻塞的打开我们的设备文件，我们知道，在之前我们编写 CC1100 的驱动程序以及倒车雷达的驱动程序的时候，在 read 函数中都有用到过 wait_event_interruptible_timeout 这个函数，这个函数的主要作用就是采用非阻塞的 read，因为每一次我们 read 函数的时候，都会先判断是否有新的数据可以读，如果没有新的数据就会休眠等待有新的数据。同时我们这里也给休眠等待规定了之间限制，即如果在规定的时间里面如果没有新的数据的话，便会自己唤醒自己。当然如果说是我们在上层应用程序只需要打开一个驱动程序的时候，其实这个方式也还比较适用。但是 Linux 内核针对这种情况呢，自己采用了一种全新的方式，那就是 poll 机制。其实 poll 机制的实现原理与我们上面用到的方法也是一样的。

poll 机制的作用

poll 机制的作用主要是通过在用户空间调用 select()和 poll()系统调用查询是否可对设备进行无阻塞访问。顾名思义是用来查询该驱动设备是否是无阻塞的。既然要查询，首先 poll 机制其本身是非阻塞的，那如何实现其本身是非阻塞呢？我们肯定是要用到等待队列机制。即进入驱动程序的自定义 poll 函数之后，我们首先将该进程加入等待队列（这个等待队列头一定要是该驱动程序的 read 或者 write 函数使用的等待队列头），然后就进入休眠等待，这个休眠等待当然也是有 timeout 限制的（如果没有限制，就成了阻塞调用了），如果在 timeout 阶段，该等待因为有新的数据而被驱动程序唤醒（能够被唤醒的主要原因就是这个等待队列的队列头是与 read 和 wirte 队列头一致的），那么我们就认为该设备是可以非阻塞

调用的，反之如果该等待是被其自己通过其 timeout 机制而唤醒，那么就认为该设备是阻塞访问的。

当应用程序调用 poll、select 函数的时候，会调用到系统调用 do_sys_poll 函数，该函数最终调用 do_poll 函数，do_poll 函数中有一个死循环，在里面又会利用 do_pollfd 函数去调用驱动中的 poll 函数（fds 中每个成员的字符驱动程序都会被扫描到），驱动程序中的 Poll 函数的工作有两个，一个就是调用 poll_wait 函数，把进程挂到等待队列中去（这个是必须的，你要睡眠，必须要在一个等待队列上面，否则到哪里去唤醒你呢？？），另一个是确定相关的 fd 是否有内容可读，如果可读，就返回 1，否则返回 0，如果返回 1 ，do_poll 函数中的 count++， 然后 do_poll 函数然后判断三个条件（if (count ||!timeout || signal_pending(current))）如果成立就直接跳出，如果不成立，就睡眠 timeout 个 jiffes 这么长的时间（调用 schedule_timeout 实现睡眠），如果在这段时间内没有其他进程去唤醒它，那么第二次执行判断的时候就会跳出死循环。如果在这段时间内有其他进程唤醒它，那么也可以跳出死循环返回（例如我们可以利用中断处理函数去唤醒它，这样的话一有数据可读，就可以让它立即返回)。

十五、同步，异步，互斥，阻塞

（1）临界资源

在操作系统中，**进程是占有资源的最小单位**（线程可以访问其所在进程内的所有资源，但线程本身并不占有资源或仅仅占有一点必须资源)。但对于某些资源来说，其在同一时间只能被一个进程所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源。

（2）同步、互斥

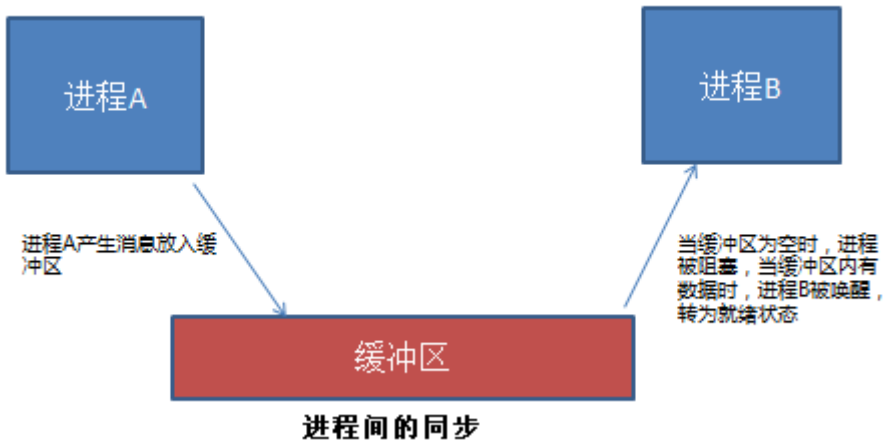
相交进程之间的关系主要有两种：**同步**与互斥（一定要记住：不是同步和异步)。所谓互斥，是指散布在不同进程之间的若干程序片断，当某个进程运行其中一个程序片段时，其它进程就不能运行它 们之中的任一程序片段，只能等到该进程运行完这个程序片段后才可以运行。所谓同步，是指散布在不同进程之间的若干程序片断，它们的运行必须严格按照规定的 某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。

显然，同步是一种更为复杂的互斥，而互斥是一种特殊的同步。

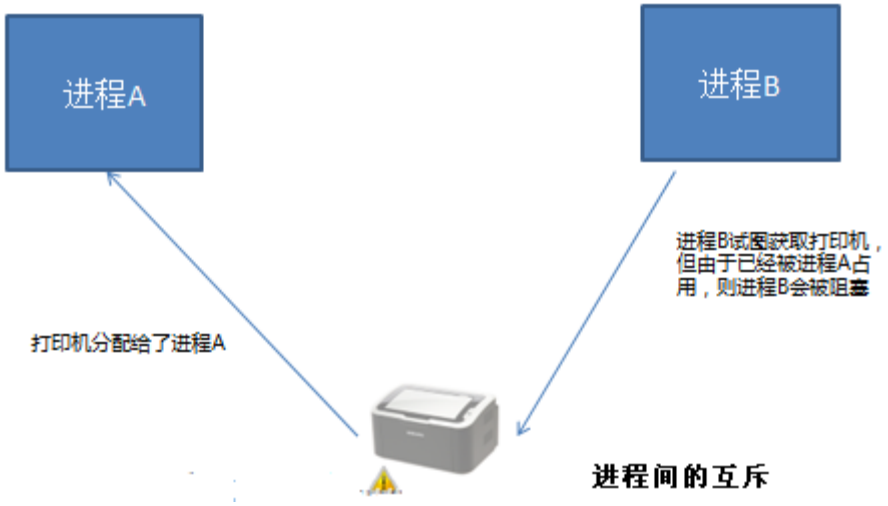
也就是说互斥是两个线程之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步也是不能同时运行，但他是必须要安照某种次序来运行相应的线程（也是一种互斥)！

互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。



进程间的同步



进程间的互斥

（3）互斥量、信号量

互斥量用于线程的互斥，信号量用于线程的同步。

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确、合理的使用公共资源。信号量(semaphore)是非负整型变量，除了初始化之外，它只能通过两个标准原子操作：wait(semap)，signal(semap)；来进行访问；信号量通过一个计数器控制对共享资源的访问，信号量的值是一个非负整数，所有通

过它的线程都会将该整数减一。如果计数器大于 0，则访问被允许，计数器减 1；如果为 0，则访问被禁止，所有试图通过它的线程都将处于等待状态。

计数器计算的结果是允许访问共享资源的通行证。因此，为了访问共享资源，线程必须从信号量得到通行证，如果该信号量的计数大于 0，则此线程获得一个通行证，这将导致信号量的计数递减，否则，此线程将阻塞直到获得一个通行证为止。当此线程不再需要访问共享资源时，它释放该通行证，这导致信号量的计数递增，如果另一个线程等待通行证，则那个线程将在那时获得通行证。

Semaphore 可以被抽象为五个操作：

1、 创建 Create

2、等待 Wait：线程等待信号量，如果值大于 0，则获得，值减一；如果只等于 0，则一直线程进入睡眠状态，知道信号量值大于 0 或者超时。 3、释放 Post：执行释放信号量，则值加一；如果此时有正在等待的线程，则唤醒该线程。

4、试图等待 TryWait：如果调用 TryWait，线程并不真正的去获得信号量，还是检查信号量是否能够被获得，如果信号量值大于 0，则 TryWait 返回成功；否则返回失败。

5、销毁 Destroy

信号量，是可以用来保护两个或多个关键代码段，这些关键代码段不能并发调用。在进入一个关键代码段之前，线程必须获取一个信号量。如果关键代码段中没有任何线程，那么线程会立即进入该框图中的那个部分。一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量，然后将 Acquire Semaphore VI 以及 Release Semaphore VI 分别放置在每个关键代码段的首末端。确认这些信号量 VI 引用的是初始创建的信号量。

（4）阻塞、非阻塞

首先来解释同步和异步的概念,这两个概念与**消息的通知机制**有关.

举个例子，比如我去银行办理业务，可能选择排队等候，也可能取一个小纸条上面有我的号码，等到排到我这一号时由柜台的人通知我轮到我去办理业务了。前者(排队等候)就是同步等待消息，而后者(等待别人通知)就是异步等待消息。在异步消息处理中，等待消息者(在这个例子中就是等待办理业务的人)往往**注册**一个回调机制，在所等待的事件被触发时由触发机制(在这里是柜台的人)通过某种机制(在这里是写在小纸条上的号码)找到等待该事件的人。

而在实际的程序中,同步消息处理就好比简单的 read/write 操作,它们需要**等待这两个操作成功才能返回**；而异步处理机制就是类似于 select/poll 之类的多路复用 IO 操作，**当**

所关注的消息被触发时，由消息触发机制通知触发对消息的处理。

其次再来解释一下阻塞和非阻塞,这两个概念与程序等待消息(无所谓同步或者异步)时的状态有关.

继续上面的那个例子,不论是排队还是使用号码等待通知,如果在这个等待的过程中,等待者除了等待消息之外不能做其它的事情，那么该机制就是阻塞的，表现在程序中，也就是该程序一直阻塞在该函数调用处不能继续往下执行。相反，有的人喜欢在银行办理这些业务的时候一边打打电话发发短信一边等待，这样的状态就是非阻塞的，因为他(等待者)没有阻塞在这个消息通知上，而是一边做自己的事情一边等待。但是需要注意了，第一种同步非阻塞形式实际上是效率低下的,想象一下你一边打着电话一边还需要抬头看到底队伍排到你了没有,如果把打电话和观察排队的位置看成是程序的两个操作的话，这个程序需要在这两种不同的行为之间来回的切换，效率可想而知是低下的。而后者,异步非阻塞形式却没有这样的问题，因为打电话是你(等待者)的事情,而通知你则是柜台(消息触发机制)的事情，程序没有在两种不同的操作中来回切换。

很多人会把同步和阻塞混淆，我想是因为很多时候同步操作会以阻塞的形式表现出来，比如很多人会写阻塞的 read/write 操作，但是别忘了可以对 fd 设置 O_NONBLOCK 标志位,这样就可以将同步操作变成非阻塞的了。同样的，很多人也会把异步和非阻塞混淆，因为异步操作一般都不会在真正的 IO 操作处被阻塞，比如如果用 select 函数，当 select 返回可读时再去 read 一般都不会被阻塞，就好比当你的号码排到时一般都是在你之前已经没有人了,所以你再去柜台办理业务就不会被阻塞。

同步和异步：上面提到过,同步和异步仅仅是关于所关注的消息如何通知的机制，而不是处理消息的机制。也就是说,同步的情况下,是由处理消息者自己去等待消息是否被触发，而异步的情况下是由触发机制来通知处理消息者，所以在异步机制中，处理消息者和触发机制之间就需要一个连接的桥梁，在我们举的例子中这个桥梁就是小纸条上面的号码，而在 select/poll 等 IO 多路复用机制中就是 fd，当消息被触发时，触发机制通过 fd 找到处理该 fd 的处理函数。

请注意理解消息通知和处理消息这两个概念，这是理解这个问题的关键所在.还是回到上面的例子，轮到你办理业务这个就是你关注的消息，而去办理业务就是对这个消息的处理，两者是有区别的.而在真实的 IO 操作时，所关注的消息就是该 fd 是否可读写,而对消息的处理就是对这个 fd 进行读写.同步/异步仅仅关注的是如何通知消息，它们对如何处理消息并不关心，好比说，银行的人仅仅通知你轮到你办理业务了,而如何办理业务他们是不知道的。

而很多人之所以把同步和阻塞混淆，我想也是因为没有区分这两个概念，比如阻塞的 read/write 操作中，其实是把消息通知和处理消息结合在了一起，在这里所关注的消息就是 fd 是否可读/写，而处理消息则是对 fd 读/写。当我们将这个 fd 设置为非阻塞的时候，read/write 操作就不会在等待消息通知这里阻塞,如果 fd 不可读/写则操作立即返回。

很多人又会问了，异步操作不会是阻塞的吧?已经通知了有消息可以处理了就一定不是阻塞的了吧?其实异步操作是可以被阻塞住的,只不过通常不是在处理消息时阻塞,而是在等待消息被触发时被阻塞.比如 select 函数,假如传入的最后一个 timeout 参数为 NULL,那么如果所关注的事件没有一个被触发,程序就会一直阻塞在这个 select 调用处.而如果使用异步非阻塞的情况,比如 aio_*组的操作,当我发起一个 aio_read 操作时,函数会马上返回不会被阻塞,当所关注的事件被触发时会调用之前注册的回调函数进行处理,具体可以参见我上面的连接给出的那篇文章.回到上面的例子中,如果在银行等待办理业务的人采用的是异步的方式去等待消息被触发,也就是领了一张小纸条,假如在这段时间里他不能离开银行做其它的事情,那么很显然,这个人被阻塞在了这个等待的操作上面;但是呢,这个人突然发觉自己烟瘾犯了,需要出去抽根烟,于是他告诉大堂经理说,排到我这个号码的时候麻烦到外面通知我一下(注册一个回调函数),那么他就没有被阻塞在这个等待的操作上面,自然这个就是异步+非阻塞的方式了。

韦东山关于此知识点讲解：

1. 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

常用原子操作函数举例：

```
atomic_t v = ATOMIC_INIT(0);          //定义原子变量 v 并初始化为 0
atomic_read(atomic_t *v);             //返回原子变量的值
void atomic_inc(atomic_t *v);         //原子变量增加 1
void atomic_dec(atomic_t *v);         //原子变量减少 1
int atomic_dec_and_test(atomic_t *v); //自减操作后测试其是否为 0，为 0 则返回 true，否则返回 false。
```

2. 信号量

信号量（semaphore）是用于保护临界区的一种常用方法，只有得到信号量的进程才能执行临界区代码。

当获取不到信号量时，进程进入休眠等待状态。

定义信号量

```
struct semaphore sem;
```

初始化信号量

```
void sema_init (struct semaphore *sem, int val);
void init_MUTEX(struct semaphore *sem);//初始化为 0

static DECLARE_MUTEX(button_lock);          //定义互斥锁
```

获得信号量

void down(struct semaphore * sem);

int down_interruptible(struct semaphore * sem);

int down_trylock(struct semaphore * sem);

释放信号量

void up(struct semaphore * sem);

3. 阻塞

阻塞操作

是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。

非阻塞操作

进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

fd = open("...", O_RDWR | O_NONBLOCK);

十六、RS422，RS485，RS232：

1，什么是串口？2，什么是 RS-232？3，什么是 RS-422？4，什么是 RS-485？5，什么是握手？1，什么是串口？

串口是计算机上一种非常通用设备通信的协议（不要与通用串行总线 Universal Serial Bus 或者 USB 混淆）。大多数计算机包含两个基于 RS232 的串口。串口同时也是仪器仪表设备通用的通信协议；很多 GPIB 兼容的设备也带有 RS- 232 口。同时，串口通信协议也可以用于获取远程采集设备的数据。

串口通信的概念非常简单，串口按位（bit）发送和接收字节。尽管比按字节（byte）的并行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。它很简单并且能够实现远距离通信。比如 IEEE488 定义并行通行状态时，规定设备线总常不得超过 20 米，并且任意两个设备间的长度不得超过 2 米；而对于串口而言，长度可达 1200 米。

典型地，串口用于 ASCII 码字符的传输。通信使用 3 根线完成：（1）地线，（2）发送，（3）接收。由于串口通信是异步的，端口能够在一根线上发送数据同时在另一根线上接收数据。其他线用于握手，但是不是必须的。串口通信最重要的参数是波特率、数据位、停止位和奇偶校验。对于两个进行通行的端口，这些参数必须匹配：

a，波特率：这是一个衡量通信速度的参数。它表示每秒钟传送的 bit 的个数。例如 300 波特表示每秒钟发送 300 个 bit。当我们提到时钟周期时，我们就是指波特率例如如果协议需要 4800 波特率，那么时钟是 4800Hz。这意味着串口通信在数据线上的采样率为 4800Hz。通常电话线的波特率为 14400，28800 和 36600。波特率可以远远大于这些值，但是波特率和距离成反比。高波特率常常用于放置的很近的仪器间的通信，典型的例子就是 GPIB 设备的通信。

b，数据位：这是衡量通信中实际数据位的参数。当计算机发送一个信息包，实际的数据不会是 8 位的，标准的值是 5、7 和 8 位。如何设置取决于你想传送的信息。比如，标准的 ASCII 码是 0～127（7 位）。扩展的 ASCII 码是 0～255（8 位）。如果数据使用简单的文本（标准 ASCII 码），那么每个数据包使用 7 位数据。每个包是指一个字节，包括开始/停止位，数据位和奇偶校验位。由于实际数据位取决于通信协议的选取，术语 “包”指任何通信的情况。

c，停止位：用于表示单个包的最后一位。典型的值为 1，1.5 和 2 位。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率同时也越慢。

d，奇偶校验位：在串口通信中一种简单的检错方式。有四种检错方式：偶、奇、高和低。当然没有校验位也是可以的。对于偶和奇校验的情况，串口会设置校验位（数据位后面的一位），用一个值确保传输的数据有偶个或者奇个逻辑高位。例如，如果数据是 011，那么对于偶校验，校验位为 0，保证逻辑高的位数是偶数个。如果是奇校验，校验位位 1，这样就有 3 个逻辑高位。高位和低位不真正的检查数据，简单置位逻辑高或者逻辑低校验。这样使得接收设备能够知道一个位的状态，有机会判断是否有噪声干扰了通信或者是否传输和接收数据是否不同步。

2，什么是 RS-232?

RS-232 （ANSI/EIA-232 标准）是 IBM-PC 及其兼容机上的串行连接标准。可用于许多用途，比如连接鼠标、打印机或者 Modem，同时也可以接工业仪器仪表。用于驱动和连线的改进，实际应用中 RS-232 的传输长度或者速度常常超过标准的值。RS-232 只限于 PC 串口和设备间点对点的通信。RS- 232 串口通信最远距离是 50 英尺。DB-9 针连接头

\ 1 2 3 4 5 /

\ 6 7 8 9 /

-------从计算机连出的线的截面。

RS-232 针脚的功能：

数据：

TXD（pin 3）：串口数据输出

RXD（pin 2）：串口数据输入

握手：

RTS（pin 7）：发送数据请求

CTS（pin 8）：清除发送

DSR（pin 6）：数据发送就绪

DCD（pin 1）：数据载波检测

DTR（pin 4）：数据终端就绪

地线：

GND（pin 5）：地线

其他

RI（pin 9）：铃声指示

3，什么是 RS-422?

RS -422（EIA RS-422-A Standard）是 Apple 的 Macintosh 计算机的串口连接标准。RS-422 使用差分信号，RS-232 使用非平衡参考地的信号。差分传输使用两根线发送和接收信号，对比 RS-232，它能更好的抗噪声和有更远的传输距离。在工业环境中更好的抗噪性和更远的传输距离是一个很大的优点。

4，什么是 RS-485?

RS -485（EIA-485 标准）是 RS-422 的改进，因为它增加了设备的个数，从 10 个增加到 32 个，同时定义了在最大设备个数情况下的电气特性，以保证足够的信号电压。有了多个设备的能力，你可以使用一个单个 RS-422 口建立设备网络。出色抗噪和多设备能力，在工业应用中建立连向 PC 机的分布式设备网络、其他数据收集控制器、HMI 或者其他操作时，串行连接会选择 RS-485。RS-485 是 RS-422 的超集，因此所有的 RS-422 设备可以被 RS-485 控制。RS-485 可以用超过 4000 英尺的线进行串行通行。

DB-9 引脚连接

\ 1 2 3 4 5 /

\ 6 7 8 9 /

一、什么是 RS-232-C 接口？采用 RS-232-C 接口有何特点？传输电缆长度如何考虑？

答： 计算机与计算机或计算机与终端之间的数据传送可以采用串行通讯和并行通讯二种方式。由于串行通讯方式具有使用线路少、成本低，特别是在远程传输时，避免了多条线路特性的不一致而被广泛采用。 在串行通讯时，要求通讯双方都采用一个标准接口，使不同的设备可以方便地连接起来进行通讯。 RS-232-C 接口（又称 EIA RS-232-C）是目前最常用的一种串行通讯接口。它是在 1970 年由美国电子工业协会（EIA）联合贝尔系统、 调制解调器厂家及计算机终端生产厂家共同制定的用于串行通讯的标 准。它的全名是“数据终端设备（DTE）和数据通讯设备（DCE）之间 串行二进制数据交换接口技术标准”该标准规定采用一个 25 个脚的 DB25 连接器，对连接器的每个引脚的信号内容加以规定，还对各种信 号的电平加以规定。

（1）接口的信号内容 实际上 RS-232-C 的 25 条引线中有许多是很少使用的，在计算机与终端通讯中一般只使用 3-9 条引线。RS-232-C 最常用的 9 条引线的信号内容见附表 1 所示

（2）接口的电气特性 在 RS-232-C 中任何一条信号线的电压均为负逻辑关系。即：逻 辑“1”，-5—-15V；逻辑“0” +5—+15V 。噪声容限为 2V。即 要求接收器能识别低至 +3V 的信号作为逻辑“0”，高到-3V 的信号 作为逻辑“1” 附表 1

此主题相关图片如下：

（3）接口的物理结构 RS-232-C 接口连接器一般使用型号为 DB-25 的 25 芯插头座,通常插头在 DCE 端,插座在 DTE 端. 一些设备与 PC 机连接的 RS-232-C 接口,因为不使用对方的传送控制信号,只需三条接口线,即“发送数据”、“接收数据”和“信号地”。所以采用 DB-9 的 9 芯插头座，传输线采用屏蔽双绞线。

二、什么是 RS-485 接口？它比 RS-232-C 接口相比有何特点？

答： 由于 RS-232-C 接口标准出现较早，难免有不足之处，主要有以下四点：

1、接口的信号电平值较高，易损坏接口电路的芯片，又因为与 TTL 电平不兼容故需使用电平转换电路方能与 TTL 电路连接。

2、传输速率较低，在异步传输时，波特率为 20Kbps。

3、接口使用一根信号线和一根信号返回线而构成共地的传输形式， 这种共地传输容易产生共模干扰，所以抗噪声干扰性弱。

4、传输距离有限，最大传输距离标准值为 50 英尺，实际上也只能 用在 50 米左右。

针对 RS-232-C 的不足，于是就不断出现了一些新的接口标准，RS-485 就是其中之一，它具有以下特点：

1、RS-485 的电气特性：逻辑“1”以两线间的电压差为+（2—6） V 表示；逻辑“0”以两线间的电压差为-（2—6）V 表示。接口信号电平比 RS-232-C 降低了，就不易损坏接口电路的芯片， 且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。

2、RS-485 的数据最高传输速率为 10Mbps

3、RS-485 接口是采用平衡驱动器和差分接收器的组合，抗共模干能力增强，即抗噪声干扰性好。

4、RS-485 接口的最大传输距离标准值为 4000 英尺，实际上可达 3000 米，另外 RS-232-C 接口在总线上只允许连接 1 个收发器， 即单站能力。而 RS-485 接口在总线上是允许连接多达 128 个收发器。即具有多站能力,这样用户可以利用单一的 RS-485 接口方便地建立起设备网络。

因 RS-485 接口具有良好的抗噪声干扰性，长的传输距离和多站能力等上述优点就使其成为首选的串行接口。 因为 RS485 接口组成的半双工网络，一般只需二根连线，所以 RS485 接口均采用屏蔽双绞线传输。 RS485 接口连接器采用 DB-9 的 9 芯插头座，与智能终端 RS485 接口采用 DB-9（孔），与键盘连接的键盘接口 RS485 采用 DB-9（针）。

十七、BSP 驱动概念：

1　BSP 概述

BSP 即 Board Support Package，板级支持包。它来源于嵌入式操作系统与硬件无关的设计思想，操作系统被设计为运行在虚拟的硬件平台上。对于具体的硬件平台，与硬件相关的代码都被封装在 BSP 中，由 BSP 向上提供虚拟的硬件平台，BSP 与操作系统通过定义好的接口进行交互。**BSP 是所有与硬件相关的代码体的集合**。

一个成熟的商用操作系统，其被广泛应用的必要条件之一就是能够支持众多的硬件平台，并实现应用程序的硬件无关性。**一般来说，这种无关性都是由操作系统实现的**。

但对于嵌入式系统来说，它没有像 PC 机那样具有广泛使用的各种工业标准、统一的硬件结构。各种嵌入式系统各不同的应用需求就决定了它一般都选用各自定制的硬件环境，每种嵌入式系统从核心的处理器到外部芯片在硬件结构上都有很大的不同。这种诸多变化的硬件环境就决定了无法完全由操作系统来实现上层软件与底层硬件之间的无关性。

因此各种商用实时操作系统，都采用了分层设计的方法，它将系统中与硬件直接相关的一层软件独立出来，称之为 Board Support Package，简称为 BSP。顾名思义，BSP 是针对某个特定的单板而设计的。如果没有单板支持软件包，则操作系统就不能在单板上运行。并且它对于用户（指开发者）也是开放的，用户可以根据不同的硬件需求对其作改动或二次开发。**BSP 在嵌入式系统中的角色，很相似于在 PC 系统中的 BIOS 和驱动程序的地位**。

BSP 的具体结构和组成根据不同的嵌入式操作系统而不同。BSP 的开发要求设计人员具备软硬件方面的综合知识。

BSP 软件与其他软件的最大区别在于 BSP 软件有一整套模板和格式，开发人员必须严格遵守，不允许任意发挥。在 BSP 软件中，绝大部分文件的文件名和所要完成的功能都是固定的。所以，BSP 软件的开发一般来说都是在一个基本成型的 BSP 软件上进行修改，以适应不同单板的需求。针对某类 CPU 的硬件单板，嵌入式操作系统(如 vxWorks)通常提供有其 DEMO 板的 BSP,这些程序位于指定的目录之下。也就是我们所说的最小系统 BSP。一般来说，我们在硬件系统设计好之后，都会先找到一个与自己系统相近的 DEMO 板 BSP（最起码是使用相同的 CPU）。并以此为基础,开发自己单板的 BSP。

定义

BSP 就是为软件操作系统正常运行提供最基本、最原始的硬件操作的软件模块，它和操作系统息息相关，但又不属于操作系统的一部分。BSP 可以分为三大部分：

1：系统上电时的硬件初始化。

2：为操作系统访问硬件驱动程序提供支持。

3：集成的硬件相关和硬件无关的操作系统所需的软件模块。

BSP 的表现形式

BSP 主要以两种形式来表现：

1：源代码（C 代码、汇编代码）、系统编译连接依靠文件。

2：二进制的目标代码和目标代码库。

BSP 在软件系统中的位置

BSP 在软件系统中的位置可以用下图来表示，BSP 为操作系统和硬件设备的互操作建了一个桥梁，操作系统通过 BSP 来完成对指定硬件的配置和管理。

BSP 向上层提供的接口有

与操作系统内核的接口（如报告 DRAM 大小、修改中断屏蔽级别等）

与操作系统的 I/O 系统的接口

与应用程序的接口

CPU 最小系统 BSP 的定义

广义上讲，单板中所有需要 CPU 控制的硬件的程序，都属于单板 BSP，但是，为了调试方便和软件的模块化，我们通常就将与此单板最小系统相关的程序简称为 BSP，而将其他程序称为驱动程序。

对于嵌入式系统来说，所谓最小系统就是一个包含：CPU, Bootrom, RAM,系统时钟，网口，串口的计算机运行环境。

这样，最小系统 BSP 就包含了 CPU 系统的初始化程序以及网口，串口，系统时钟等设备的驱动程序。

BSP 的主要功能

BSP 的主要功能在于配置系统硬件使其工作于正常的状态，完成硬件与软件之间的数据交互，为 OS 及上层应用程序提供一个与硬件无关的软件平台。因此从**执行角度**来说，其可以分为两大部分：

1）目标板启动时的硬件初始化及多任务环境的初始化

2）目标板上控制各个硬件设备正常运行的设备驱动程序，由它来完成硬件与软件之间的信息交互

通常我们认为 BSP 是**为 OS 服务**的，但实际上，BSP 软件包中的部分程序对 OS 也并不是必须的，从这个角度，又可以将 BSP 划分为两部分：

1）最小系统 BSP，即我们通常所称的 BSP

2) 设备驱动程序

开发 BSP 需要的条件

目标硬件：硬件调测完毕，经过必要的软件测试

必要的硬件设计文档：如地址空间的分布，CPU 和其他芯片的工作模式等。

操作系统

交叉开发工具：编译器、汇编器、链接器等

下载机制：bootrom 或仿真器等

2 BSP 实际开发的主要过程

1. 掌握开发中使用的操作系统，和在这种操作系统下开发 BSP 的要求。
2. 研读所选 CPU 的资料。
3. 研读硬件设计文挡。
4. 研读电路板中器件的资料。
5. 找一个 BSP 模板，熟悉它并在此基础上开发自己的 BSP。从头研制 BSP 工作量极大，也没有必要。
6. 利用仿真器进行调试，开发最小 BSP 系统。
7. 在最小 BSP 的基础上，利用 Tornado 集成开发环境，进一步调试外围设备，配置、完善系统。
8. 调试单板上的设备驱动程序。

3 BSP 的调试方法（最小系统的调试和设备驱动程序的调试）

3.1 仿真器调试方式:

在串口和网口初始化及发挥功能以前，用仿真器调试是一种相对来讲很方便的手段。BSP 软件的调试通常需要利用仿真器来进行。目前市场上的大多数仿真器都能支持 JTAG 接口。典型的仿真器调试环境如下图所示:

调试计算机通过 RS232 接口与仿真器相连，完成对仿真器的初始化配置工作，通常这项工作只在第一次使用仿真器时进行，配置结果一般会被仿真器存储起来。仿真器通过以太网口与调试计算机相连，通过 JTAG 接口与目标板相连，利用这条通路，仿真器就可将计算机上的程序下载到目标板上进行调试。

目标板上的串口和以太网口为被调试的对象，和调试计算机相连，主要是可以通过调试计算机检验目标板上的接口是否工作正常。

在仿真器环境下,既可以调试 vxWorks 映象,也可以调试 bootrom 映象。当调试 BootRom 映象时，需要修改 CONFIG.H 和 MAKEFILE 文件将这段代码定位到 RAM 中，然后通过仿真器下载到目标板上的 RAM 中进行调试。

目前常用的有两类仿真器，一是 JTAG 仿真器，二是全功能在线仿真器。前者是利用处理器中的调试模块的功能，通过其 JTAG 边界扫描口来与仿真器连接。这种方式的仿真器比较便宜，连接比较方便。但由于仅通过十几条线来调试，因而功能有局限。对于全功能在线仿真器来说，由于其仿真头完全取代目标板上的 CPU，因而功能非常强大。这类仿真器为了能够全速仿真时钟速度高于 100MHz 的处理器，通常必须采用极其复杂的设计和工艺，因而其价格比较昂贵。

3.2 “黑”调

在没有仿真器的情况下一般使用“黑”调，具体的方法是加“指示灯”、用示波器测量硬件信号等，目的是打通串口，达到宿主机与目标机的通信。这种调试方法无法跟踪软件的运行这种调试方法要求所使用的 BSP 模板与自己的单板基本一致。
“黑”调的工程步骤: BSP 完成

3.3 使用集成开发环境

在进行 VxWorks 下 BSP 开发时，如果最小系统的 BSP 已经能够正常运行，则可以使用 VxWorks 的集成开发环境 Tornado。Tornado II 工具提供一个高度可视化和自动化的开发环境，加快了基于 VxWorks 的应用开发。这样，不论对于初次使用还是有经验的开发者，使用 Tornado II 开发其应用是快速而方便的。

4 中断处理

4.1 采用中断处理方式的原因:

保证处理的实时性、减少 CPU 的消耗。

4.2 中断**的处理流程**

中断处理程序首先切换到中断堆栈，保存程序计数器和寄存器等中断现场状态，然后对中断进行处理，中断处理过程中必须要及时清除中断源，最后要恢复中断前的程序计数器和寄存器等现场状态，由中断处理程序返回。

Interrupt Service Code
实时系统中的中断处理非常重要，系统通常通过中断获取外部事件。为了尽可能块的响应中断，VxWorks 中的 ISRs 运行在特定的上下文（非任务上下文），中断处理无需任务的上下文切换。

我们可以使用除了 VxWorks 系统使用的之外的系统硬件中断，VxWorks 提供了例程 intConnect()用于将 C 程序与任何中断相连接。VxWorks 的 ISRs 运行在特定的上下文（x86 中断使用当前被中断掉的任务的堆栈，PPC 有单独的全局中断堆栈）因而中断处理没有任务的上下文切换。

4.3 中断的堆栈

大部分系统规定（如 PowerPC）：所有的中断使用同一个专用堆栈，这个堆栈在系统启动时根据特定的配置参数由系统来分配和初始化。要求堆栈足够大来处理最坏的中断嵌套。

然而有一些系统不允许有单独的中断堆栈（如 x86），在这种情况下，中断使用当前被中断掉的任务的堆栈。如果使用这种结构，必须给每个任务开足够大的任务堆栈来处理最坏的中断嵌套和调用嵌套。

可以在开发中使用 checkStack()来查看在栈空间中任务和中断的堆栈是如何分布的。

4.4 ISR 的一些限制

1、 ISR 要尽量的短，能在任务中完成的工作就不要放在 ISR 中。
2、 ISR 不能调用将会导致阻塞的子程序。
3、 ISR 不能 take 信号量，但是 ISR 可以 give 信号量。
4、 由于子程序 malloc（）、free（）使用了信号量，ISR 不能调用它们。
5、 ISR 不能通过 VxWorks 的驱动执行 I/O。
6、 ISR 不能调用使用了浮点协处理器的子程序。

在 ISR 中不能调用的函数列表参见：《VxWorks Programmer Guide》中 2.5.3 Special Limitations of ISRs。

4.5 中断服务程序与任务的通信

由于中断事件通常涉及到任务级代码，因此必须提供中断服务程序和一般任务的通信机制。VxWorks 提供的中断服务程序和一般任务的通讯机制有：

共享存储区和环形缓冲
信号量：中断服务程序能够释放信号量（不包括互斥信号量和 VxMP 共享信号量），任务能够等待该信号量。
消息队列，中断服务程序能够向消息队列发送消息，任务能够从消息队列里接收消息。
管道：中断服务程序可以向管道写数据，任务可以从管道读取数据。
信号灯：中断服务程序能够通过发信号通知任务，触发相应的信号处理程序的异步调度。

5 常用总线协议

HDLC、
High-level data link control (HDLC) is one of the most common protocols in the data link layer, layer 2 of the OSI model.
UART、
universal asynchronous receiver transmitter (UART) protocol is commonly used to send low-speed data between devices.
Ethernet/IEEE 802.3 、
ATM、
PCI、
I2C、
SPI(motorola：Serial Peripheral Interface)

exchange data between cpu and peripheral devices (such as EEPROMs, real-time clocks, A/D)converters, and ISDN devices.

5.1 PCI 总线简介

PCI：周边器件互联（Peripheral Component Interconnect）。
目前最新版本 2.2，但很多器件还只支持 2.1。

PCI 与器件的基本关系。

Host/PCI 北桥：连接主处理器总线到 PCI 总线
PCI/ISA 南桥：连接 PCI 总线到 ISA（或 EISA）总线，南桥通常含 IDE 控制器、中断
控制器、USB 主控制器、DMA 控制器。
PCI/PCI： 连接 PCI 总线到 PCI 总线

设备和功能：

支持 256 个 PCI 总线。
每条 PCI 总线上最多可以有 32 个设备（最多 5－6 个比较合适）。
每个设备可以有 1－8 个功能。

性能：

33M，32bit 传输速率 132MB
33M，64bit 传输速率 264MB
66M，32bit 传输速率 264MB
66M，64bit 传输速率 528MB

6 单板的硬件组成

BSP 与单板密切相关，要开发 BSP 就要了解单板的硬件组成，单板一般由 CPU 最小系统和一些外围硬件设备构成。

CPU 最小系统：

CPU、内存、内存控制器、调试串口、调试网口、系统时钟、桥片、外围芯片（包括）、实时时钟、定时器、FPGA、部分嵌入式系统也包括软硬盘控制器、显卡、键盘。

不同单板使用不同的专用设备芯片：

DMA 控制器、E1 传输芯片、光接口芯片、时隙交换、FLASH、host/pci 桥片 pci/pci 桥片、以太网口芯片（如 intel 的 82559、realtek 的 8139）、以太网口交换芯片（BCM5616）、CSM5000、CSM5500、看门狗、专用 FPGA 逻辑等等。

7．处理器：

我们目前经常使用的处理器：POWERPC、ARM、MIPS、x86 等系列处理器

POWERPC：Mpc860、Mpc8260、Mpc850、Mpc8250、Mpc755、Mpc8245、Mpc7450、IBM750

ARM ：IXP1200、IXC1100

X86： （PIII 处理器）

各处理器的特点（见硬件相关文档）：嵌入式系统使用的 CPU 一般具有低功耗、体

积小、集成度高等特点，而且一般内部集成了内存控制器、串口控制器、以太网口控制器等芯片。

例如：MPC8260 POWERQUICC Ⅱ是一个功能强大的嵌入式通讯处理器，它集成了一个 64-bit 高性能的 PowerPC 系列的 RISC 微处理器和一个 32-bit 的 RISC 通讯处理器。同 MPC860 一样，MPC8260 由三个主要功能块组成，但它具有更强大的功能：

一个 64-bit 的内核，是 PowerPC MPC603e 微处理器的变种。它的处理速度可达 100~200MHz，并支持 L2 cache。

一个系统接口单元（SIU）。它具有一个更加灵活的存储器控制器，可以与几乎所有类型的存储器接口。并支持 JTAG 控制器 IEEE 1149.1 测试端口（TAP）。

一个通信处理模块。它不仅包含了 MPC860 上的所用通信外围控制器，还增加了三个高性能通信通道（FCC）支持新的高速协议，两个多通道控制器（MCC）可支持 128 个串行全双工通道。

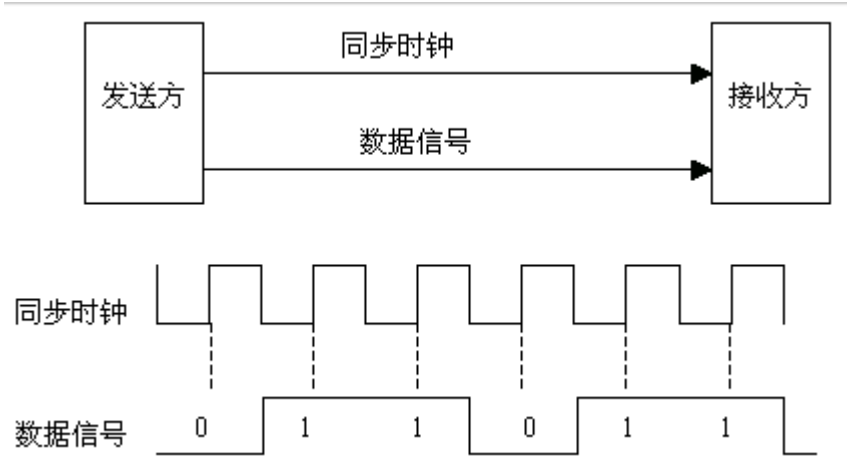
十八、串口驱动协议

串口的重要性不言而喻。我们可以通过串口来打印 debug 的信息，以此来定位代码的错误位置。

我们也可以通过串口来向内核传入命令，可以说它是开发人员常用的一个交互终端。当进行驱动开发时，总是需要用到串口来显示内核的打印信息。对于开发来说，很是重要。

串口：属于串行通信的一种，与之对应的是并行通信。串行通信就是数据的各位在同一根线上顺序依次按位传输。

同步通信：要求发生时钟和接收时钟保持严格同步。发送方先发送一个或两个特殊字符，该字符称为同步字符。当发送方和接收方达到同步后，就可以一个字符接一个字符地发送一大块数据，而不再需要用起始位和停止位了，这样可以明显地提高数据的传输速率。



异步通信：在异步通信中，数据通常是以字符或字节为单位组成数据帧进行传送的。收、发端各有一套彼此独立，互不同步的通信机构，由于收发数据的帧格式相同，因此可以相互识别接收到的数据信息.

串口：串行异步通信.

异步通信信息帧格式如图 9.4 所示。

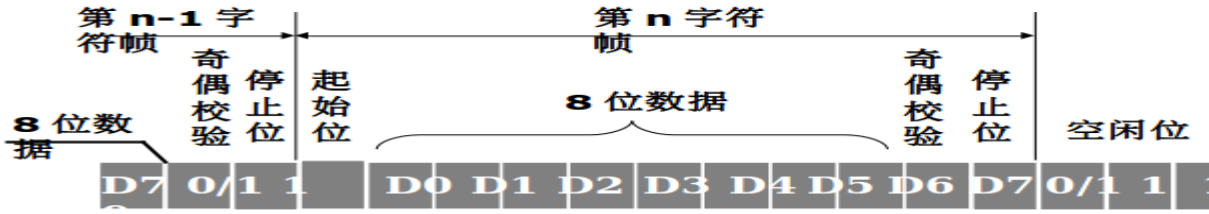


图 9.4　异步通信帧格式

(1) 起始位:

在没有数据传送时，通信线上处于逻辑“1”状态。当发送端要发送 1 个字符数据时，首先发送 1 个逻辑“0”信号，这个低电平便是帧格式的起始位。其作用是向接收端表示发送端开始发送一帧数据。接收端检测到这个低电平后，就准备接收数据信号。

(2) 数据位:

在起始位之后，发送端发出(或接收端接收)的是数据位，数据的位数没有严格的限制，5～8 位均可。由低位到高位逐位传送。

(3) 奇偶校验位:

数据位发送完(接收完)之后，可发送一位用来检验数据在传送过程中是否出错的奇偶校验位。奇偶校验是收发双方预先约定好的有限差错检验方式之一。有时也可不用奇偶校验。

(4) 停止位:

字符帧格式的最后部分是停止位，逻辑“1”电平有效，它可占 1/2 位、1 位或 2 位。停止位表示传送一帧信息的结束，也为发送下一帧信息作好准备。

串口通信的波特率:

波特率(Baud Rate)是串行通信中一个重要概念，它是指传输数据的速率，亦称比特率。波特率的定义是每秒传输二进制数码的位数。如:波特率为 1200bps 是指每秒钟能传输 1200 位二进制数码。

串行通信的制式:

1) 单工方式：数据只能从一个方向传送到另一个方向。

2) 半双工：数据可以双向传输，但是不能同时双向传输。当发送时，不能接收，当接收时，不能发送。

3) 全双工：数据可以双向传输，且可以同时进行接受和发送。

串口协议是全双工协议，可以同时发送和接收。两根线比较重要，TX 和 RX。TX 用来传输，RX 用来接收.

(5)校验位:

串行通信的目的不只是传送数据信息，更重要的是应确保准确无误地传送。因此必须考虑在通信过程中对数据差错进行校验，因为差错校验是保证准确无误地通信的关键。常用差错校验方法有奇偶校验、累加和校验以及循环冗余码校验等。

1) 奇偶校验:

奇偶校验的特点是按字符校验，即在发送每个字符数据之后都附加一位奇偶校验位(1 或 0),当设置为奇校验时，数据中 1 的个数与校验位 1 的个数之和应为奇数；反之则为偶校

验。收、发双方应具有一致的差错检验设置，当接收 1 帧字符时，对 1 的个数进行检验，若奇偶性(收、发双方)一致则说明传输正确。奇偶校验只能检测到那种影响奇偶位数的错误，比较低级且速度慢，一般只用在异步通信中。

2）累加和校验：

累加和校验是指发送方将所发送的数据块求和，并将“校验和”附加到数据块末尾。接收方接收数据时也是先对数据块求和，将所得结果与发送方的“校验和”进行比较，若两者相同，表示传送正确，若不同则表示传送出了差错。“校验和”的加法运算可用逻辑加，也可用算术加。累加和校验的缺点是无法检验出字节或位序的错误。

3）循环冗余码校验(CRC)：

循环冗余码校验的基本原理是将一个数据块看成一个位数很长的二进制数，然后用一个特定的数去除它，将余数作校验码附在数据块之后一起发送。接收端收到该数据块和校验码后，进行同样的运算来校验传送是否出错。目前 CRC 已广泛用于数据存储和数据通信中，并在国际上形成规范，市面上已有不少现成的 CRC 软件算法。

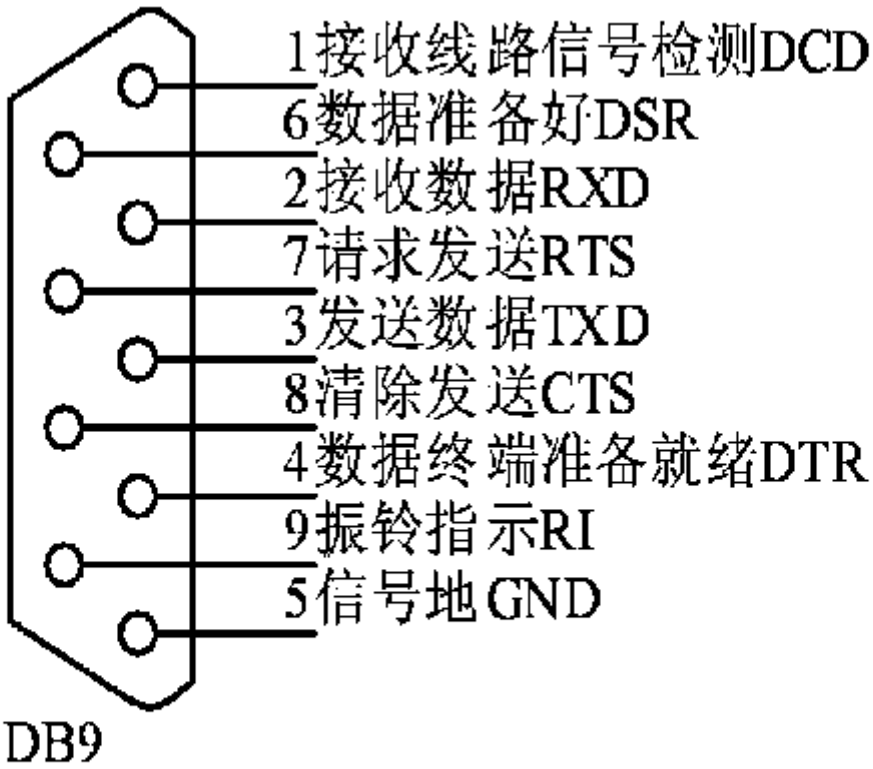


图 9.15　微机 9 针 D 形串口连接器

计算机串行通信中主要使用了如下信号：

(1) 数据传送信号:发送数据(TXD)；接收数据(RXD)。

(2) 调制解调器控制信号：请求发送(RTS)；清除发送(CTS)；数据通信设备准备就绪(DSR)；数据终端准备就绪(DTR)。

(3) 定位信号：接收时钟(RXC)；发送时钟(TXC)。

(4) 信号地 GND。

十九、typedef struct 用法：

typedef struct student {
char *name;
int age;
char *sex;
float height;
}stu;

此时 stu==struct student

当然 student 可以省略

typedef struct{
char *name;
int age;
char *sex;
float height;
}student;

student stu;可以直接定义

但不可以直接定义

student *stu 使用

因为该指针未初始化，未指向一个地方，称为野指针（结构体指针）。

可以先 student stu;

Student* stu1;

Stu1=&stu;

Typdef 实际上是为 struct 起了一个别名

十九、正则表达式

在电脑上查找文件：*.c “*”表示通配符，任意字符

用正则表达式 ：.表示用任意字符（换行符除外），*重复 0 次或更多次，

+ 重复 1 次或更多次：？重复 0 次或 1 次 【】表示这些字符里的某一个

二十、extern 的用法

1、声明一个全局（外部）变量

当用 extern 声明一个全局变量的时候，首先应明确一点：extern 的作用范围是整个工程，也就是说当我们在.h 文件中写了 extern int a;链接的时候链接器会去其他的.c 文件中找有没有 int a 的定义，如果没有，链接报错；当 extern int a;写在.c 文件中时，链接器会在这个.c 文件该声明语句之后找有没有 int a 的定义，然后去其他的.cpp 文件中找，如果都找不到，链接报错。值得注意的一点：当 extern 语句出现在头文件中时，不要将声明和定义在一条语句中给出，也就是不要在头文件中写类似于这样的语句：

extern int a = 1;这种写法，在 gcc 编译时会给出一个警告：warning: 'a' initialized and declared 'extern'

注意：external 声明和定义的区别,定义只有一次，但是声明没有限制

int a;　　--->定义，默认为 0（等效于 int a = 0;）

extern int a;　　--->声明外部变量

extern int a = 1;　　--->定义性声明（即在定义的同时声明为全局变量），一般不提倡

比如：

头文件 a.h:

```
extern int a = 1;
```

头文件 b.h:

```
#include "a.h"
.....
```

源文件 test.c:

```
#include "a.h"
#include "b.h"
.....
```

这样肯定会报错：会报错说变量 a 重复定义

所有一般（提倡）的做法是：

只在头文件中通过 extern 给出全局变量的声明（即 external int a; 而不要写成 external int a = 1;），并在源文件中给出定义（并且只能定义一次）

比如：

头文件 a.h:

```
.....
```

源文件 1.c:

```
#include "a.h"
int a = 0;　　--->定义
```

头文件 b.h:

```
extern int a;　　--->声明
```

源文件 2.c:

```
#include <stdio.h>
#include "2.h"
void print()
{
    printf("a = %d\n", a);
}
```

源文件 main.c

```
#include "1.h"
#include "2.h"
int main()
{
    print();
    return 0;
}
```

2、extern “C” { /*用 C 实现的内容（通常写在另外的.c 文件中）*/ }

C++完全兼容 C，当 extern 与“C”连用时，作用是告诉编译器用 C 的编译规则去解

析 extern “C”后面的内容。最常见的差别就是 C++支持函数重载，而标准 C 是不支持的。如果不指明 extern “C”，C++编译器会根据自己的规则在编译函数时为函数名加上特定的后缀以区别不同的重载版本，而如果是按 C 的标准来编译的话，则不需要。

3.extern 对于变量在不同 C 文件中的使用，必需加。对于函数则可以不加，但一般为了具有可读性，加上，告诉编译器和读者在别的文件中有定义。可以这样用，在某个 c 文件中定义的变量或函数，如 test.c 中 int a； 可在 test.h 中定义 external int a,不要初始化

static 和 external 定义的全局变量区别：

注意：用 static 定义的全局静态变量可以被模块内所有的函数访问，但不能被模块外其他函数访问。而用 external 定义的全局

1、static 修饰全局变量时，声明和定义是同时给出的；而 extern 一般是定义和声明分开，且定义只能一次

2、static 的全局作用域只是自身编译单元（即一个.c 文件以及这个.c 文件所包含的.h 文件）；而 extern 的全局作用域是整个工程（一个工程可以包含很多个.h 和.c 文件）。即区别就在于“全局”的范围是整个工程，还是自身编译单元。

二十一、套接字类型：

（1）SOCK_STREAM：流式套接字，提供面向连接、可靠的数据传输服务，数据按字节流、按顺序收发，保证在传输过程中无丢失、无冗余。TCP 协议支持该套接字。

（2）SOCK_DGRAM：数据报套接字，提供面向无连接的服务，数据收发无序，不能保证数据的准确到达。UDP 协议支持该套接字。

（3）SOCK_RAW：原始套接字。允许对低于传输层的协议或物理网络直接访问，例如可以接收和发送 ICMP 报文。常用于检测新的协议。

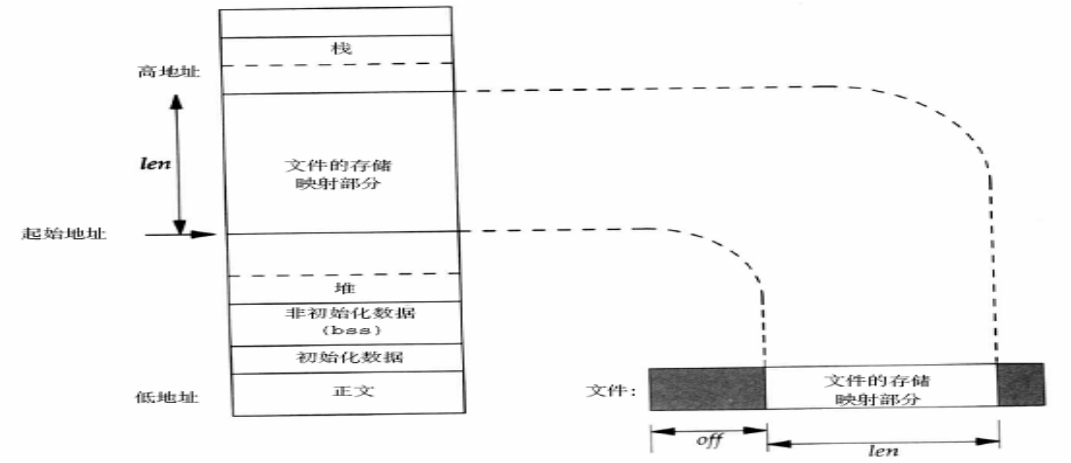
二十二、内核有哪些分配内存函数

__get_free_pages	直接对页框进行操作	4MB	适用于分配较大量的连续物理内存
kmem_cache_alloc	基于 slab 机制实现	128KB	适合需要频繁申请释放相同大小内存块时使用
Kmalloc	基于 kmem_cache_alloc 实现	128KB	最常见的分配方式，需要小于页框大小的内存时可以使用
vmalloc	建立非连续物理内存到虚拟地址的映射		物理不连续，适合需要大内存，但是对地址连续性没有要求的场合

dma_alloc_coherent	基于__alloc_pages 实现	4MB	适用于 DMA 操作
ioremap	实现已知物理地址到虚拟地址的映射		适用于物理地址已知的场合，如设备驱动
alloc_bootmem	在启动 kernel 时，预留一段内存，内核看不见		小于物理内存大小，内存管理要求较高
xzalloc			
kzalloc			

二十三、mmap 和 ioremap

void* mmap (void * addr , size_t len , int prot , int flags ,int fd , off_t offset) 内存映射函数 mmap，负责把文件内容映射到进程的虚拟内存空间，通过对这段内存的读取和修改，来实现对文件的读取和修改,而不需要再调用 read，write 等操作。

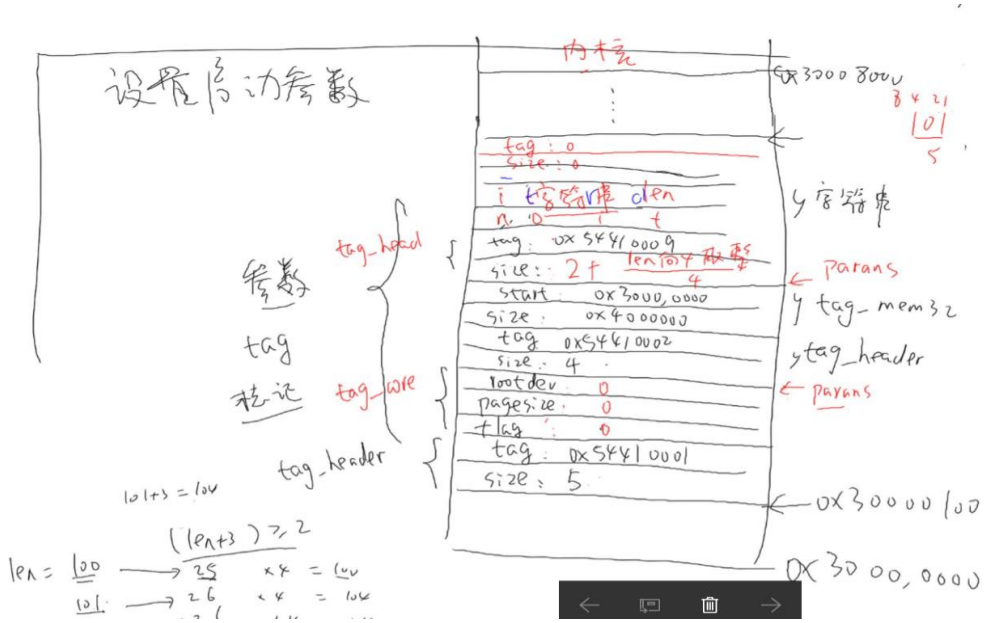


addr：
指定映射的起始地址，通常设为 NULL，由系统指定。
v length:
映射到内存的文件长度。
v prot：
映射区的保护方式，可以是:
PROT_EXEC: 映射区可被执行
PROT_READ: 映射区可被读取
PROT_WRITE: 映射区可被写入
flags: 映射区的特性，可以是:
vMAP_SHARED:写入映射区的数据会复制回文件，且允许其他映射该文件的进程共享。
vMAP_PRIVATE:对映射区的写入操作会产生一个映射区的复制(copy-on-write)，对此区域所做的修改不会写回原文件。
fd:由 open 返回的文件描述符，代表要映射的文件。

offset:以文件开始处的偏移量，必须是分页大小的整数倍，通常为 0，表示从文件头开始映射。
解除映射
int munmap(void *start,size_t length)功能：取消参数 start 所指向的映射内存，参数 length 表示欲取消的内存大小。返回值：解除成功返回 0，否则返回－1，错误原因存于 errno 中。
虚拟内存区域：虚拟内存区域是进程的虚拟地址空间中的一个同质区间，即具有同样特性的连续地址范围。一个进程的内存映象由下面几部分组成：程序代码、数据、BSS 和栈区域，以及内存映射的区域。

二十四、bootloader：

Bootloader 目标：启动内核
最简单的 bootloader 的编写步骤：
1. 初始化硬件：关看门狗、设置时钟、设置 SDRAM、初始化 NAND FLASH
2. 如果 bootloader 比较大，要把它重定位到 SDRAM
3. 把内核从 NAND FLASH 读到 SDRAM
4. 设置"要传给内核的参数",双方约定，以什么样的格式，存在什么地方
TAG；0x30000100 存放参数
Main 函数
启动参数 setup_start_tag
内存参数 setup_memory_tags
命令参数 setup_commandline_tag("noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0");
结束参数 setup_end_tag
因为单位为 4 字节，所以后面需设置 4 字节对齐，+3/4



5. 跳转执行内核

改进：
1. 提高 CPU 频率, 200MHZ ==> 400MHZ
2. 启动 ICACHE
二十五、汇编指令：
Adr：取得地址，位置无关
Ldr r0，=0x043324523;若立即数简单，则直接 mov r0, 0x043324523.若比较复杂,则把数存在某个地方，去那个地方读数据

二十五、U-boot：

从 make 100ask24x0_config 分析　config.mk，再分析 Makefile 得到编译流程：书 257

分析 start.S，发现 1.设为管理模式；2.关看门狗；3.屏蔽中断；4.初始化 SDRAM；5.设置栈（为了 C 函数）；6.设置时钟；7.代码重定位（flash ->SDRAM）；8.清 bss 端（静态变量和全局变量）；
以上可以称为第一阶段，硬件初始化（汇编）。
第二阶段：网卡，烧写 flash，usb，串口等开发功能（c 语言）
9.调用 start_armboot(调到第二阶段)

Warning: unable to open an initial console.
Kernel panic - not syncing: No init found.　Try passing init= option to kernel.的解决:
Nand erase，可能是内核问题，可以试着 set bootargs init=/linuxrc,若还是不行，可以尝试用旧版本内核。
Busybox :ls ,cd 的组合 ： 执行 ls 等价于执行/bin/ls -> busybox

Init 程序：1.读取配置文件；2.解析配置文件；3.执行用户程序
移植：
1、下载、建立 source insight 工程、编译、烧写、如果无运行分析原因
tar xjf u-boot-2012.04.01.tar.bz2
cd u-boot-2012.04.01
make smdk2410_config
make
2. 分析 u-boot: 通过链接命令分析组成文件、阅读代码分析启动过程

a. 初始化硬件：关看门狗、设置时钟、设置 SDRAM、初始化 NAND FLASH
b. 如果 bootloader 比较大，要把它重定位到 SDRAM
c. 把内核从 NAND FLASH 读到 SDRAM
d. 设置"要传给内核的参数"
e. 跳转执行内核

2.1 set the cpu to SVC32 mode
2.2 turn off the watchdog
2.3 mask all IRQs by setting all bits in the INTMR
2.4 设置时钟比例

2.5 设置内存控制器
2.6 设置栈，调用 C 函数 board_init_f
2.7 调用函数数组 init_sequence 里的各个函数
2.7.1 board_early_init_f：设置系统时钟、设置 GPIO
.....
2.8 重定位代码:
2.8.1 从 NOR FLASH 把代码复制到 SDRAM (because of the big program \
and Nor flash cannot random write)
2.8.2 程序的链接地址是 0，访问全局变量、静态变量、调用函数时是使"基于 0 地址编译得到的地址"
现在把程序复制到了 SDRAM
需要修改代码，把"基于 0 地址编译得到的地址"改为新地址 -pie (in reland dynsym section)
2.8.3 程序里有些地址在链接时不能确定，要到运行前才能确定：fixabs
2.9 clear_bss
2.10 调用 C 函数 board_init_r：第 2 阶段的代码

分析“重定位之修改代码为新地址”

```
#ifndef CONFIG_SPL_BUILD
    /*
     * fix .rel.dyn relocations
     */
    ldr   r0, _TEXT_BASE          /* r0 <- Text base */
                                  //ldr r0=0,code base address
    sub   r9, r6, r0              /* r9 <- relocation offset */
                                                //r9=r6-r0= 0x33ff4100-0=0x33ff4100
    ldr   r10, _dynsym_start_ofs  /* r10 <- sym table ofs */
                                                // r10=00073608
    add   r10, r10, r0            /* r10 <- sym table in FLASH */
                                                // r10=00073608+0
    ldr   r2, _rel_dyn_start_ofs  /* r2 <- rel dyn start ofs */
                                                //r2=0x0006b568
    add   r2, r2, r0              /* r2 <- rel dyn start in FLASH */
                                                //r2=r2+r0=0x0006b568
    ldr   r3, _rel_dyn_end_ofs    /* r3 <- rel dyn end ofs */
                                                //r3=0x00073608

    add   r3, r3, r0              /* r3 <- rel dyn end in FLASH */
                                                //r3=r3+r0=r3=0x00073608
fixloop:
    ldr   r0, [r2]                /* r0 <- location to fix up, IN FLASH! */
                          //r0=00000020
    add   r0, r0, r9              /* r0 <- location to fix up in RAM */
                                      //1.r0=r0+r9=00000020+0x33ff4100=0x33ff4120
```

```
        ldr    r1, [r2, #4]
                                    //1.r1=[r2+4]=00000017
        and  r7, r1, #0xff
                        //1.r7=r1&0xff=00000017
        cmp r7, #23                 /* relative fixup? */
        r7==0x17
        beq  fixrel
        cmp r7, #2                  /* absolute fixup? */
        beq  fixabs
        /* ignore unknown type of fixup */
        b      fixnext
fixabs:
        /* absolute fix: set location to (offset) symbol value */
        mov r1, r1, LSR #4          /* r1 <- symbol index in .dynsym */
        add  r1, r10, r1         /* r1 <- address of symbol in table */
        ldr    r1, [r1, #4]             /* r1 <- symbol value */
        add  r1, r1, r9          /* r1 <- relocated sym addr */
        b      fixnext
fixrel:
        /* relative fix: increase location by offset */
        ldr    r1, [r0]
        1.r1=[00000020]=0x000001e0

        add  r1, r1, r9
        1.r1=r1+r9=0x000001e0+0x33ff4100=33f411e0
fixnext:
        str    r1, [r0]
        1.[0x33ff4120]=33f411e0
        add  r2, r2, #8         /* each rel.dyn entry is 8 bytes */
        1.r2=r2+8=0x0006b568+8=0x0006b570
        cmp r2, r3
        blo   fixloop
#endif

clear_bss:
#ifndef CONFIG_SPL_BUILD
        ldr    r0, _bss_start_ofs
        ldr    r1, _bss_end_ofs
        mov r4, r6                  /* reloc addr */
        add  r0, r0, r4
        add  r1, r1, r4
        mov r2, #0x00000000             /* clear                        */

clbss_l:str r2, [r0]            /* clear loop...                */
```

```
    add  r0, r0, #4
    cmp r0, r1
    bne  clbss_l

    bl coloured_LED_init
    bl red_led_on
#endif
```

可以修改配置定义 CONFIG_S3C2440

3. 修改 U-BOOT 代码

3.1 建一个单板

```
cd board/samsung/
cp smdk2410 smdk2440 -rf
cd ../../include/configs/
cp smdk2410.h smdk2440.h
```

修改 boards.cfg:

仿照

```
smdk2410                arm         arm920t      -              samsung
s3c24x0
```

添加:

```
smdk2440                arm         arm920t      -              samsung
s3c24x0
```

3.2 烧写看结果

3.3 调试:

a. 阅读代码发现不足: UBOOT 里先以 60MHZ 的时钟计算参数来设置内存控制器，但是 MPLL 还未设置

处理措施: 把 MPLL 的设置放到 start.S 里，取消 board_early_init_f 里对 MPLL 的设置

```
    //writel(0xFFFFFF, &clk_power->locktime);

    /* configure MPLL */
    //writel((M_MDIV << 12) + (M_PDIV << 4) + M_SDIV,
          //    &clk_power->mpllcon);
```

编译出来的 uboot 非常大，可以先烧写主光盘里的 u-boot.bin 到 nor，然后用这个 uboot 来烧写新的 uboot

3.4 乱码，查看串口波特率的设置，发现在 get_HCLK 里没有定义 CONFIG_S3C2440

处理措施: include/configs/smdk2440.h: 去掉 CONFIG_S3C2410

```
#define CONFIG_S3C2440
//#define CONFIG_CMD_NAND
```

3.5 修改 UBOOT 支持 NAND 启动

原来的代码在链接时加了"-pie"选项，使得 u-boot.bin 里多了"*(.rel*)", "*(.dynsym)"

使得程序非常大，不利于从 NAND 启动(重定位之前的启动代码应该少于 4K)

3.5.1 去掉 "-pie"选项

arch/arm/config.mk:75:LDFLAGS_u-boot += -pie 去掉这行

u-boot.bin binary file not include bss section

3.5.2 参考"毕业班第 1 课"的 start.S, init.c 来修改代码

把 init.c 放入 board/samsung/smdk2440 目录, 修改 Makefile

修改 CONFIG_SYS_TEXT_BASE 为 0x33f80000

修改 start.S

3.5.3 修改 board_init_f, 把 relocate_code 去掉

3.5.4 修改链接脚本: 把 start.S, init.c, lowlevel.S 等文件放在最前面

./arch/arm/cpu/u-boot.lds:

board/samsung/smdk2440/libsmdk2440.o

自己编译出现 undefined reference to "nand_info"

编译出现 undefined reference to nand_info

解决: 直接让&nand_info[0]变为 0（暂时）

第二次移植发现没有这个问题，应该是第一次某个文件弄错了

u-boot.dis 需要用 arm-linux-objdump -D u-boot > u-boot.dis 自己去生成

3.6 修改 UBOOT 支持 NOR FLASH

drivers\mtd\jedec_flash.c 加上新的型号

#define CONFIG_SYS_MAX_FLASH_SECT (128)

修复了重定时留下来的 BUG: SP 要重新设置

3.7 修改 UBOOT 支持 NAND FLASH

修改: include/configs/smdk2440.h: #define CONFIG_CMD_NAND

把 drivers\mtd\nand\s3c2410_nand.c 复制为 s3c2440_nand.c

分析过程:

```
nand_init
    nand_init_chip
        board_nand_init
            设置 nand_chip 结构体, 提供底层的操作函数
        nand_scan
            nand_scan_ident
                nand_set_defaults
                    chip->select_chip = nand_select_chip;
                    chip->cmdfunc = nand_command;
                    chip->read_byte = busw ? nand_read_byte16 : nand_read_byte;
```

```
nand_get_flash_type
    chip->select_chip
    chip->cmdfunc(mtd, NAND_CMD_RESET, -1, -1);
        nand_command()   // 即可以用来发命令，也可以用来发列地址(页内地址)、行地址(哪一页)
            chip->cmd_ctrl
                s3c2440_hwcontrol

    chip->cmdfunc(mtd, NAND_CMD_READID, 0x00, -1);
    *maf_id = chip->read_byte(mtd);
    *dev_id = chip->read_byte(mtd);
```

3.8 修改 UBOOT 支持 DM9000 网卡

```
eth_initialize
    board_eth_init
        cs8900_initialize
```

*** ERROR: `ethaddr' not set
set ipaddr 192.168.1.17
set ethaddr 00:0c:29:4d:e4:f4
set serverip 192.168.1.103

4. 易用性修裁剪及制作补丁

修改 smdk2440，从头往后看，哪些是不需要的

内核打印出来的分区信息

0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
question:

undefined reference to `get_mtd_device_nm'
reason:driver/mtdcore.c not compiled
open certain Makefile to find the Macro and add in smdk2440.h

nand erase 60000 200000
nand write 30000000 60000 200000

```
tftp 30000000 uImage
nand erase.part kernel
nand write 30000000 kernel
```

烧写 JFFS2

```
tftp 30000000 fs_mini_mdev.jffs2
nand erase.part rootfs
nand write.jffs2 30000000 0x00260000 5b89a8

set bootargs console=ttySAC0 root=/dev/mtdblock3 rootfstype=jffs2
```

烧写 YAFFS

```
tftp 30000000 fs_mini_mdev.yaffs2
nand erase.part root
nand write.yaffs 30000000 260000    889bc0
```

更新 UBOOT:

```
tftp 30000000 u-boot.bin;protect off all;erase 0 3ffff ;cp.b 30000000 0 40000; reset
```

制作补丁:

```
diff -urN u-boot-2012.04.01 u-boot-2012.04.01_100ask > u-boot-2012.04.01_100ask.patch
```

分析"重定位之修改代码为新地址":

```
#ifndef CONFIG_SPL_BUILD
    /*
     * fix .rel.dyn relocations
     */
    ldr    r0, _TEXT_BASE          /* r0 <- Text base */
    // r0=0, 代码基地址

    sub   r9, r6, r0          /* r9 <- relocation offset */
    // r9 = r6-r0 = 0x33f41000 - 0 = 0x33f41000

    ldr    r10, _dynsym_start_ofs    /* r10 <- sym table ofs */
    // r10 = 00073608

    add   r10, r10, r0              /* r10 <- sym table in FLASH */
    // r10 = 00073608 + 0 = 00073608

    ldr    r2, _rel_dyn_start_ofs      /* r2 <- rel dyn start ofs */
    // r2=0006b568
```

```
        add  r2, r2, r0         /* r2 <- rel dyn start in FLASH */
        // r2=r2+r0=0006b568

        ldr    r3, _rel_dyn_end_ofs       /* r3 <- rel dyn end ofs */
        // r3=00073608

        add  r3, r3, r0         /* r3 <- rel dyn end in FLASH */
        // r3=r3+r0=00073608

fixloop:
        ldr    r0, [r2]             /* r0 <- location to fix up, IN FLASH! */
        1. r0=[0006b568]=00000020

        add  r0, r0, r9         /* r0 <- location to fix up in RAM */
        1. r0=r0+r9=00000020 + 0x33f41000 = 0x33f41020

        ldr    r1, [r2, #4]
        1. r1=[0006b568+4]=00000017

        and  r7, r1, #0xff
        1. r7=r1&0xff=00000017

        cmp r7, #23                 /* relative fixup? */
        1. r7 == 23(0x17)

        beq  fixrel
        cmp r7, #2                  /* absolute fixup? */

        beq  fixabs
        /* ignore unknown type of fixup */
        b      fixnext
fixabs:
        /* absolute fix: set location to (offset) symbol value */
        mov r1, r1, LSR #4          /* r1 <- symbol index in .dynsym */

        add  r1, r10, r1        /* r1 <- address of symbol in table */

        ldr    r1, [r1, #4]             /* r1 <- symbol value */

        add  r1, r1, r9         /* r1 <- relocated sym addr */

        b      fixnext
fixrel:
        /* relative fix: increase location by offset */
```

```
        ldr    r1, [r0]
        1. r1=[00000020]=000001e0

        add   r1, r1, r9
        1. r1=r1+r9=000001e0 + 0x33f41000 = 33F411E0

fixnext:
        str    r1, [r0]
        1. [0x33f41020] = 33F411E0

        add   r2, r2, #8         /* each rel.dyn entry is 8 bytes */
        1. r2=r2+8=0006b568+8=6B570

        cmp r2, r3
        1.

        blo   fixloop
#endif
```

二十六、Makefile 的使用

Makefile 的规则:在讲述这个 Makefile 之前，还是让我们先来粗略地看一看 Makefile 的规则。

target... : prerequisites ...

command

example： edit : main.o command.o display.o

```
        cc -o edit main.o command.o display.o

  main.o : main.c defs.h

        cc -c main.c

  command.o : command.c defs.h command.h

        cc -c command.c

  display.o : display.c defs.h buffer.h

        cc -c display.c

  clean :

        rm edit main.o command.o display.o \
```

target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签(Label)，对于标签这种特性，在后续的“伪目标”章节中会有叙述。 prerequisites 就是，要生成那个 target 所需要的文件或是目标。command 也就是 make 需要执行的命令。(任意的 Shell 命令)，在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键作为开头。记住，make 并不管命令是怎么工作的，他只管执行所定义的命令。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。 有一点要说明的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 lable 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

1.3 make 是如何工作的

在默认的方式下，也就是我们只输入 make 命令。那么，

1. make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。
2. 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。
3. 如果 edit 文件不存在，或是 edit 所依赖的后面的 .o 文件的文件修改时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件。
4. 如果 edit 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。(这有点像一个堆栈的过程)
5. 当然，你的 C 文件和 H 文件是存在的啦，于是 make 会生成 .o 文件，然后再用 .o 文件声明 make 的终极任务，也就是执行文件 edit 了。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么 make 就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像 clean 这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要 make 执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。 于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 file.c，那么根据我们的依赖性，我们的目标 file.o 会被重编译（也就是在这个依性关系后面所定义的命令），于是 file.o 的文件也是最新的啦，于是 file.o 的文件修改时间要比 edit 要新，所以 edit 也会被重新链接了（详见 edit 目标文件后定义的命令）。而如果我们改变了“command.h”，那么，kdb.o、command.o 和 files.o 都会被重编译，并且，edit 会被重链接。

1.4 makefile 中使用变量

在上面的例子中，先让我们看看 edit 的规则：

```
edit : main.o command.o display.o

	cc -o edit main.o command.o display.o
```

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在 clean 中）。当然，我们

的 makefile 并不复杂，所以在两个地方加也不累，但如果 makefile 变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了 makefile 的易维护，在 makefile 中我们可以使用变量。makefile 的变量也就是一个字符串，理解成 c 语言中的宏可能会更好。比如，我们声明一个变量，叫 objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么啦，只要能够表示 obj 文件就行了。我们在 makefile 一开始就这样定义：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

于是，我们就可以很方便地在我们的 makefile 中以“$(objects)”的方式来使用这个变量了，于是我们的改良版 makefile 就变成下面这个样子：

```
objects = main.o kbd.o command.o display.o \
          insert.osearch.o files.o utils.o
edit : $(objects)
       cc -o edit $(objects)
main.o : main.c defs.h
       cc -c main.c
kbd.o : kbd.c defs.h command.h
       cc -c kbd.c
command.o : command.c defs.h command.h
       cc -c command.c
display.o : display.c defs.h buffer.h
       cc -c display.c
insert.o : insert.c defs.h buffer.h
       cc -c insert.c
search.o : search.c defs.h buffer.h
       cc -c search.c
files.o : files.c defs.h buffer.h command.h
       cc -c files.c
utils.o : utils.c defs.h
       cc -c utils.c
clean :
       rm edit $(objects)
```

于是如果有新的 .o 文件加入，我们只需简单地修改一下 objects 变量就可以了。
关于变量更多的话题，我会在后续给你一一道来。

1.5 让 make 自动推导

GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的 make 会自动识别，并自己推导命令。

只要 make 看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来，于是，我们的 makefile 再也不用写得这么复杂。我们的是新的 makefile 又出炉了。

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
```

```
	cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
	rm edit $(objects)
```

这种方法，也就是 make 的“隐晦规则”。上面文件内容中，“.PHONY”表示，clean 是个伪目标文件。

关于更为详细的“隐晦规则”和“伪目标文件”，我会在后续给你一一道来。

1.6 另类风格的 makefile

即然我们的 make 可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来，好吧，没有问题，这个对于 make 来说很容易，谁叫它提供了自动推导命令和文件的功能呢？来看看最新风格的 makefile 吧。

```
objects = main.o kbd.o command.o display.o \
	insert.o search.o files.o utils.o

edit : $(objects)
	cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h

.PHONY : clean
clean :
	rm edit $(objects)
```

这种风格，让我们的 makefile 变得很简单，但我们的文件依赖关系就显得有点凌乱了。鱼和熊掌不可兼得。还看你的喜好了。我是不喜欢这种风格的，一是文件的依赖关系看不清楚，二是如果文件一多，要加入几个新的.o 文件，那就理不清楚了。

1.7 清空目标文件的规则

每个 Makefile 中都应该写一个清空目标文件（.o 和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个“修养”（呵呵，还记得我的《编程修养》吗）。一般的风格都是：

```
clean:
	rm edit $(objects)
```

更为稳健的做法是：

```
.PHONY : clean
clean :
	-rm edit $(objects)
```

前面说过，.PHONY 意思表示 clean 是一个“伪目标”,。而在 rm 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean 的规则不要放在文件的开头，不然，这就会变成 make 的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean 从来都是放在文件的最后”。

上面就是一个 makefile 的概貌，也是 makefile 的基础，下面还有很多 makefile 的相关细节，准备好了吗？准备好了就来。

2 Makefile 总述

2.1 Makefile 里有什么？

Makefile 里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

1. 显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由 Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
2. 隐晦规则。由于我们的 make 有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写 Makefile，这是由 make 所支持的。
3. 变量的定义。在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点你 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。
4. 文件指示。其包括了三个部分，一个是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样；另一个是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。
5. 注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用“#”字符，这个就像 C/C++中的“//”一样。如果你要在你的 Makefile 中使用“#”字符，可以用反斜框进行转义，如：“\#”。

最后，还值得一提的是，在 Makefile 中的命令，必须要以[Tab]键开始。

2.2Makefile 的文件名

默认的情况下，make 命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用“GNUmakefile”，这个文件是 GNU 的 make 识别的。有另外一些 make 只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的 make 都支持“makefile”和“Makefile”这两种默认文件名。

当然，你可以使用别的文件名来书写 Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的 Makefile，你可以使用 make 的“-f”和“--file”参数，如：make -f Make.Linux 或 make --file Make.AIX。

2.3 引用其它的 Makefile

在 Makefile 使用 include 关键字可以把别的 Makefile 包含进来，这很像 C 语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include 的语法是：

include<filename>filename 可以是当前操作系统 Shell 的文件模式（可以保含路径和通配

符）

在 include 前面可以有一些空字符，但是绝不能是[Tab]键开始。include 和可以用一个或多个空格隔开。举个例子，你有这样几个 Makefile：a.mk、b.mk、c.mk，还有一个文件叫 foo.make，以及一个变量$(bar)，其包含了 e.mk 和 f.mk，那么，下面的语句：

include foo.make *.mk $(bar)

等价于：

include foo.make a.mk b.mk c.mk e.mk f.mk

make 命令开始时，会把找寻 include 所指出的其它 Makefile，并把其内容安置在当前的位置。就好像 C/C++的#include 指令一样。如果文件都没有指定绝对路径或是相对路径的话，make 会在当前目录下首先寻找，如果当前目录下没有找到，那么，make 还会在下面的几个目录下找：

1.如果 make 执行时，有“-I”或“--include-dir”参数，那么 make 就会在这个参数所指定的目录下去寻找。

2.如果目录/include（一般是：/usr/local/bin 或/usr/include）存在的话，make 也会去找。

如果有文件没有找到的话，make 会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成 makefile 的读取，make 会再重试这些没有找到，或是不能读取的文件，如果还是不行，make 才会出现一条致命信息。如果你想让 make 不理那些无法读取的文件，而继续执行，你可以在 include 前加一个减号“-”。如：

-include<filename>

其表示，无论 include 过程中出现什么错误，都不要报错继续执行。和其它版本 make 兼容的相关命令是 sinclude，其作用和这一个是一样的。

2.4 环境变量 MAKEFILES

如果你的当前环境中定义了环境变量 MAKEFILES，那么，make 会把这个变量中的值做一个类似于 include 的动作。这个变量中的值是其它的 Makefile，用空格分隔。只是，它和 include 不同的是，从这个环境变中引入的 Makefile 的“目标”不会起作用，如果环境变量中定义的文件发现错误，make 也会不理。

但是在这里我还是建议不要使用这个环境变量，因为只要这个变量一被定义，那么当你使用 make 时，所有的 Makefile 都会受到它的影响，这绝不是你想看到的。在这里提这个事，只是为了告诉大家，也许有时候你的 Makefile 出现了怪事，那么你可以看看当前环境中有没有定义这个变量。

2.5 make 的工作方式

GNU 的 make 工作时的执行步骤入下：（想来其它的 make 也是类似）

1. 读入所有的 Makefile。
2. 读入被 include 的其它 Makefile。
3. 初始化文件中的变量。
4. 推导隐晦规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

1-5 步为第一个阶段，6-7 为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make 会把其展开在使用的位置。但 make 并不会完全马上展开，make 使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部

展开。
当然，这个工作方式你不一定要清楚，但是知道这个方式你也会对 make 更为熟悉。有了这个基础，后续部分也就容易看懂了。
3 Makefile 书写规则
规则包含两个部分，一个是依赖关系，一个是生成目标的方法。
在 Makefile 中，规则的顺序是很重要的，因为，Makefile 中只应该有一个最终目标，其它的目标都是被这个目标所连带出来的，所以一定要让 make 知道你的最终目标是什么。一般来说，定义在 Makefile 中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make 所完成的也就是这个目标。
好了，还是让我们来看一看如何书写规则。
3.1 规则举例
```
foo.o: foo.c defs.h       # foo 模块
        cc -c -g foo.c
```
看到这个例子，各位应该不是很陌生了，前面也已说过，foo.o 是我们的目标，foo.c 和 defs.h 是目标所依赖的源文件，而只有一个命令“cc -c -g foo.c”（以 Tab 键开头）。这个规则告诉我们两件事：
1. 文件的依赖关系，foo.o 依赖于 foo.c 和 defs.h 的文件，如果 foo.c 和 defs.h 的文件日期要比 foo.o 文件日期要新，或是 foo.o 不存在，那么依赖关系发生。
2. 如果生成（或更新）foo.o 文件。也就是那个 cc 命令，其说明了，如何生成 foo.o 这个文件。（当然 foo.c 文件 include 了 defs.h 文件）
3.2 规则的语法
```
    targets : prerequisites
     command
     ...
```
或是这样：
```
    targets : prerequisites ; command
        command
        ...
```
targets 是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。
command 是命令行，如果其不与“target:prerequisites”在一行，那么，必须以[Tab 键]开头，如果和 prerequisites 在一行，那么可以用分号做为分隔。（见上）
prerequisites 也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。这个在前面已经讲过了。
如果命令太长，你可以使用反斜框（‘\’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make 两件事，文件的依赖关系和如何成成目标文件。
一般来说，make 会以 UNIX 的标准 Shell，也就是/bin/sh 来执行命令。
3.3 在规则中使用通配符
如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make 支持三各通配符：“*”，“?”和“[...]”。这是和 Unix 的 B-Shell 是相同的。

"~"
波浪号（“~”）字符在文件名中也有比较特殊的用途。如果是“~/test”，这就表示当前用

户的$HOME 目录下的 test 目录。而“~hchen/test”则表示用户 hchen 的宿主目录下的 test 目录。（这些都是 Unix 下的小知识了，make 也支持）而在 Windows 或是 MS-DOS 下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量“HOME”而定。

"*"

通配符代替了你一系列的文件，如“*.c”表示所以后缀为 c 的文件。一个需要我们注意的是，如果我们的文件名中有通配符，如：“*”，那么可以用转义字符“\”，如“*”来表示真实的“*”字符，而不是任意长度的字符串。

好吧，还是先来看几个例子吧：

```
clean:
    rm -f *.o
```

上面这个例子我不不多说了，这是操作系统 Shell 所支持的通配符。这是在命令中的通配符。

```
print: *.c
    lpr -p $?
    touch print
```

上面这个例子说明了通配符也可以在我们的规则中，目标 print 依赖于所有的[.c]文件。其中的“$?”是一个自动化变量，我会在后面给你讲述。

```
objects = *.o
```

上面这个例子，表示了通配符同样可以用在变量中。并不是说[*.o]会展开，不！objects 的值就是“*.o”。Makefile 中的变量其实就是 C/C++中的宏。如果你要让通配符在变量中展开，也就是让 objects 的值是所有[.o]的文件名的集合，那么，你可以这样：

```
objects := $(wildcard *.o)
```

这种用法由关键字“wildcard”指出，关于 Makefile 的关键字，我们将在后面讨论。

3.4 文件搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当 make 需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉 make，让 make 在自动去找。

Makefile 文件中的特殊变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make 只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make 就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

```
VPATH = src:../headers
```

上面的的定义指定两个目录，“src”和“../headers”，make 会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用 make 的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个 make 的关键字，这和上面提到的那个 VPATH 变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1. vpath < pattern> < directories> 为符合模式< pattern>的文件指定搜索目录<directories>。
2. vpath < pattern> 清除符合模式< pattern>的文件的搜索目录。
3. vpath 清除所有已被设置好了的文件搜索目录。

vapth 使用方法中的< pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符，例如，“%.h”表示所有以“.h”结尾的文件。< pattern>指定了要搜索的文件集，而< directories>则指定了的文件集的搜索的目录。例如：

vpath %.h ../headers

该语句表示，要求 make 在“../headers”目录下搜索所有以“.h”结尾的文件。(如果某文件在当前目录没有找到的话)

我们可以连续地使用 vpath 语句，以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的< pattern>，或是被重复了的< pattern>，那么，make 会按照 vpath 语句的先后顺序来执行搜索。如：

vpath %.c foo

vpath % blish

vpath %.c bar

其表示“.c”结尾的文件，先在“foo”目录，然后是“blish”，最后是“bar”目录。

vpath %.c foo:bar

vpath % blish

而上面的语句则表示“.c”结尾的文件，先在“foo”目录，然后是“bar”目录，最后才是“blish”目录。

3.5 伪目标

最早先的一个例子中，我们提到过一个“clean”的目标，这是一个“伪目标”，

```
clean:
        rm *.o temp
```

正像我们前面例子中的“clean”一样，即然我们生成了许多文件编译文件，我们也应该提供一个清除它们的“目标”以备完整地重编译而用。 (以“make clean”来使用该目标)

因为，我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以 make 无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向 make 说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。于是整个过程可以这样写：

```
.PHONY: clean
clean:
        rm *.o temp
```

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。一个示例就是，如果你的 Makefile 需要一口气生成若干个可执行文件，但你只想简单地敲一个 make 完事，并且，所有的目标文件都写在一个 Makefile 中，那么你可以使用“伪目标”这个特性：

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
```

```
        cc -o prog2 prog2.o

  prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile 中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如“all”这个目标新。所以，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。“.PHONY : all”声明了“all”这个目标为“伪目标”。

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例子：

```
  .PHONY: cleanall cleanobj cleandiff

  cleanall : cleanobj cleandiff
        rm program

  cleanobj :
        rm *.o

  cleandiff :
        rm *.diff
```

“makeclean”将清除所有要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思。我们可以输入“makecleanall”和“make cleanobj”和“makecleandiff”命令来达到清除不同种类文件的目的

3.6 多目标

Makefile 的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会可我们带来麻烦，不过好在我们的可以使用一个自动化变量“$@”（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

```
  bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

上述规则等价于：

```
  bigoutput : text.g
        generate text.g -big > bigoutput
  littleoutput : text.g
        generate text.g -little > littleoutput
```

其中，-$(subst output,,$@)中的“$”表示执行一个 Makefile 的函数，函数名为 subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“$@”表示目标的集合，就像一个数组，“$@”依次取出目标，并执于命令。

3.7 静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

```
<targets...>: <target-pattern>: <prereq-patterns ...>
```

<commands>
...
targets 定义了一系列的目标文件，可以有通配符。是目标的一个集合。
target-parrtern 是指明了 targets 的模式，也就是目标集模式。
prereq-parrterns 是目标的依赖模式，它对 target-parrtern 形成的模式再进行一次依赖目标的定义。
这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的<target-parrtern>定义成“%.o”，意思是我们的集合中都是以“.o”结尾的，而如果我们的<prereq-parrterns>定义成“%.c”，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。
所以，我们的“目标模式”或是“依赖模式”中都应该有“%”这个字符，如果你的文件名中有“%”那么你可以使用反斜杠“\”进行转义，来标明真实的“%”字符。
看一个例子：

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
```

上面的例子中，指明了我们的目标从$object 中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$object 集合的模式，而依赖模式“%.c”则取模式“%.o”的“%”，也就是“foobar”，并为其加下“.c”的后缀，于是，我们的依赖目标就是“foo.cbar.c”。而命令中的“$<”和“$@”则是自动化变量，“$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“$@”表示目标集（foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

```
foo.o : foo.c
	$(CC) -c $(CFLAGS) foo.c -o foo.o
bar.o : bar.c
	$(CC) -c $(CFLAGS) bar.c -o bar.o
```

试想，如果我们的“%.o”有几百个，那种我们只要用这种很简单的“静态模式规则”就可以写完一堆规则，实在是太有效率了。“静态模式规则”的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
	emacs -f batch-byte-compile $<
```

$(filter%.o,$(files))表示调用 Makefile 的 filter 函数，过滤“$filter”集，只要其中模式为“%.o”的内容。其的它内容，我就不用多说了吧。这个例字展示了 Makefile 中更大的弹性。
3.8 自动生成依赖性

在 Makefile 中，我们的依赖关系可能会需要包含一系列的头文件，比如，如果我们的 main.c 中有一句“#include "defs.h"”，那么我们的依赖关系应该是：

```
main.o : main.c defs.h
```

但是，如果是一个比较大型的工程，你必需清楚哪些 C 文件包含了哪些头文件，并且，你在加入或删除头文件时，也需要小心地修改 Makefile，这是一个很没有维护性的工作。为了避免这种繁重而又容易出错的事情，我们可以使用 C/C++编译的一个功能。大多数的 C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：

```
cc -M main.c
```

其输出是：

```
main.o : main.c defs.h
```

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用 GNU 的 C/C++编译器，你得用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。

gcc-M main.c 的输出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \
    /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \
    /usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
    /usr/include/bits/sched.h /usr/include/libio.h \
    /usr/include/_G_config.h /usr/include/wchar.h \
    /usr/include/bits/wchar.h /usr/include/gconv.h \
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \
    /usr/include/bits/stdio_lim.h
```

gcc-MM main.c 的输出则是：

```
main.o: main.c defs.h
```

那么，编译器的这个功能如何与我们的 Makefile 联系在一起呢。因为这样一来，我们的 Makefile 也要根据这些源文件重新生成，让 Makefile 自己依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。GNU 组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个“name.c”的文件都生成一个“name.d”的 Makefile 文件，[.d]文件中就存放对应[.c]文件的依赖关系。

于是，我们可以写出[.c]文件和[.d]文件的依赖关系，并让 make 自动更新或自成[.d]文件，并把其包含在我们的主 Makefile 中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

```
%.d: %.c
    @set -e; rm -f $@; \
     $(CC) -M $(CPPFLAGS) $< > $@.
; \
    sed 's,$*\.o[ :]*,\1.o $@ : ,g' < $@.
> $@; \
    rm -f $@.
```

这个规则的意思是，所有的[.d]文件依赖于[.c]文件，“rm-f $@”的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件“$<”，也就是[.c]文件生成依赖文件，

“$@”表示模式“%.d”文件，如果有一个 C 文件是 name.c，那么“%”就是“name”，“”意为一个随机编号，第二行生成的文件有可能是“name.d.12345”，第三行使用 sed 命令做了一个替换，关于 sed 命令的用法请参看相关的使用文档。第四行就是删除临时文件。
总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入[.d]文件的依赖，即把依赖关系：

```
main.o : main.c defs.h
```

转成：

```
main.o main.d : main.c defs.h
```

于是，我们的[.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主 Makefile 中。我们可以使用 Makefile 的“include”命令，来引入别的 Makefile 文件（前面讲过），例如：

```
sources = foo.c bar.c


include $(sources:.c=.d)
```

上述语句中的“$(sources:.c=.d)”中的“.c=.d”的意思是做一个替换，把变量$(sources)所有[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为 include 是按次来载入文件，最先载入的[.d]文件中的目标会成为默认目标
4 Makefile 书写命令

每条规则中的命令和操作系统 Shell 的命令行是一致的。make 会一按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以 Tab 键开头的，那么 make 会认为其是一个空命令。
我们在 UNIX 下可能会使用不同的 Shell，但是 make 的命令默认是被“/bin/sh”——UNIX 的标准 Shell 解释执行的。除非你特别指定一个其它的 Shell。Makefile 中，“#”是注释符，很像 C/C++中的“//”，其后的本行字符都被注释。
4.1 显示命令
通常，make 会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被 make 显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

```
@echo 正在编译 XXX 模块.....
```

当 make 执行时，会输出“正在编译 XXX 模块.....”字串，但不会输出命令，如果没有“@”，那么，make 将输出：

```
echo 正在编译 XXX 模块.....
正在编译 XXX 模块.....
```

如果 make 执行时，带入 make 参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的 Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。
而 make 参数“-s”或“--slient”则是全面禁止命令的显示。
4.2 命令执行
当依赖目标新于目标时，也就是当规则的目标需要被更新时，make 会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是 cd 命令，你希望第二条命令得在 cd 之后的基础

上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

示例一：

```
exec:
    cd /home/hchen
    pwd
```

示例二：

```
exec:
    cd /home/hchen; pwd
```

当我们执行“make exec”时，第一个例子中的 cd 没有作用，pwd 会打印出当前的 Makefile 目录，而第二个例子中，cd 就起作用了，pwd 会打印出“/home/hchen”。

make 一般是使用环境变量 SHELL 中所定义的系统 Shell 来执行命令，默认情况下使用 UNIX 的标准 Shell——/bin/sh 来执行命令。但在 MS-DOS 下有点特殊，因为 MS-DOS 下没有 SHELL 环境变量，当然你也可以指定。如果你指定了 UNIX 风格的目录形式，首先，make 会在 SHELL 所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在 PATH 环境变量中所定义的所有路径中寻找。MS-DOS 中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

4.3 命令出错

每当命令运行完后，make 会检测每个命令的返回码，如果命令返回成功，那么 make 会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么 make 就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如 mkdir 命令，我们一定需要建立一个目录，如果目录不存在，那么 mkdir 就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用 mkdir 的意思就是一定要有这样的一个目录，于是我们就不希望 mkdir 出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在 Makefile 的命令行前加一个减号“-”（在 Tab 键之后），标记为不管命令出不出错都认为是成功的。如：

```
clean:
    -rm -f *.o
```

还有一个全局的办法是，给 make 加上“-i”或是“--ignore-errors”参数，那么，Makefile 中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的 make 的参数的是“-k”或是“--keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终目该规则的执行，但继续执行其它规则。

4.4 嵌套执行 make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的 Makefile，这有利于让我们的 Makefile 变得更加地简洁，而不至于把所有的东西全部写在一个 Makefile 中，这样会很难维护我们的 Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫 subdir，这个目录下有个 Makefile 文件，来指明了这个目录

下文件的编译规则。那么我们总控的 Makefile 可以这样书写：

```
subsystem:
        cd subdir && $(MAKE)
```

其等价于：

```
subsystem:
        $(MAKE) -C subdir
```

定义$(MAKE)宏变量的意思是，也许我们的 make 需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行 make 命令。

我们把这个 Makefile 叫做“总控 Makefile”，总控 Makefile 的变量可以传递到下级的 Makefile 中（如果你显示的声明），但是不会覆盖下层的 Makefile 中所定义的变量，除非指定了“-e”参数。

如果你要传递变量到下级 Makefile 中，那么你可以使用这样的声明：

export<variable ...>

如果你不想让某些变量传递到下级 Makefile 中，那么你可以这样声明：

unexport<variable ...>

如：

示例一：

```
export variable = value
```

其等价于：

```
variable = value
export variable
```

其等价于：

```
export variable := value
```

其等价于：

```
variable := value
export variable
```

示例二：

```
export variable += value
```

其等价于：

```
variable += value
export variable
```

如果你要传递所有的变量，那么，只要一个 export 就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是 SHELL，一个是 MAKEFLAGS，这两个变量不管你是否 export，其总是要传递到下层 Makefile 中，特别是 MAKEFILES 变量，其中包含了 make 的参数信息，如果我们执行“总控 Makefile”时有 make 参数或是在上层 Makefile 中定义了这个变量，那么 MAKEFILES 变量将会是这些参数，并会传递到下层 Makefile 中，这是一个系统

级的环境变量。
但是 make 命令中的有几个参数并不往下传递，它们是“-C”，“-f”，“-h” “-o”和“-W”（有关 Makefile 参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

```
subsystem:
        cd subdir && $(MAKE) MAKEFLAGS=
```

如果你定义了环境变量 MAKEFLAGS，那么你得确信其中的选项是大家都会用到的，如果其中有“-t”，“-n”,和“-q”参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。
还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在 make 的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级 make 目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们会看到：

```
make: Entering directory `/home/hchen/gnu/make'.
```

而在完成下层 make 后离开目录时，我们会看到：

```
make: Leaving directory `/home/hchen/gnu/make'
```

当你使用“-C”参数来指定 make 下层 Makefile 时，“-w”会被自动打开的。如果参数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。
4.5 定义命令包
如果Makefile中出现一些相同命令序列,那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束，如：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

这里，“run-yacc”是这个命令包的名字，其不要和 Makefile 中的变量重名。在“define”和“endef”中的两行就是命令序列。这个命令包中的第一个命令是运行 Yacc 程序，因为 Yacc 程序总是生成“y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

```
foo.c : foo.y
        $(run-yacc)
```

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“run-yacc”中的“$^”就是“foo.y”，“$@”就是“foo.c”（有关这种以“$”开头的特殊变量，我们会在后面介绍），make 在执行命令包时，命令包中的每个命令会被依次独立执行。
使用变量——在 Makefile 中的定义的变量，就像是 C/C++语言中的宏一样，他代表了一个文本字串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++所不同的是，你可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在“目标”，“依赖目标”，“命令”或是 Makefile 的其它部分中。变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“:”、“#”、“=”或是空字符（空格、回车等）。变量是大小写敏感的，“foo”、“Foo”和“FOO”是三个不同的变量名。传统的 Makefile 的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。有一些变量是很奇怪字串，如“$<”、“$@”

等，这些是自动化变量，我会在后面介绍。

一、变量的基础

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个例子：

objects = program.o foo.o utils.o

program : $(objects)

cc -o program $(objects)

$(objects) : defs.h

变量会在使用它的地方精确地展开，就像 C/C++中的宏一样，例如：

foo = c

prog.o : prog.$(foo)

(foo)(foo) -$(foo) prog.$(foo)

展开后得到： prog.o : prog.c

cc -c prog.c 当然，千万不要在你的 Makefile 中这样干，这里只是举个例子来表明 Makefile 中的变量在使用处展开的真实样子。可见其就是一个“替代”的原理。另外，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，但我还是强烈建议你给变量加上括号。

二、变量中的变量

在定义变量的值时，我们可以使用其它变量来构造变量的值，在 Makefile 中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的使用“=”号，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后面定义的值。如：

foo = $(bar)

bar = $(ugh)

ugh = Huh?

all:

echo $(foo)

我们执行“make all”将会打出变量$(foo)的值是“Huh?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可见，变量是可以使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们可以把变量的真实值推到后面来定义，如：

CFLAGS = $(include_dirs) -O

include_dirs = -Ifoo -Ibar

当“CFLAGS”在命令中被展开时，会是“-Ifoo -Ibar -O”。但这种形式也有不好的地方，那就是递归定义，如：

CFLAGS = $(CFLAGS) -O

或：

A = $(B)

B = $(A)这会让 make 陷入无限的变量展开过程中去，当然，我们的 make 是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的 make 运行时非常慢，更糟糕的是，他会使用得两个 make 的函数“wildcard”和“shell”发生不可预

知的错误。因为你不会知道这两个函数会被调用多少次。
为了避免上面的这种方法，我们可以使用 make 中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如：
x := foo
y := $(x) bar
x := later
其等价于：
y := foo bar
x := later
值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：
y := $(x) bar
x := foo
那么，y 的值是“bar”，而不是“foo bar”。
上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了 make 的函数、条件表达式和一个系统变量“MAKELEVEL”的使用：ifeq (0,${MAKELEVEL})
cur-dir := $(shell pwd)
whoami := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
关于条件表达式和函数，我们在后面再说，对于系统变量“MAKELEVEL”，其意思是，如果我们的 make 有一个嵌套执行的动作（参见前面的“嵌套使用 make”），那么，这个变量会记录了我们的当前 Makefile 的调用层数。
下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个空格，那么我们可以这样来：nullstring :=
space := $(nullstring) # end of the line
nullstring 是一个 Empty 变量，其中什么也没有，而我们的 space 的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个 Empty 变量来标明变量的值开始了，而后面采用“#”注释符来表示变量定义的终止，这样，我们可以定义出其值是一个空格的变量。请注意这里关于“#”的使用，注释符“#”的这种特性值得我们注意，如果我们这样定义一个变量：
dir := /foo/bar # directory to put the frobs in
dir 这个变量的值是“/foo/bar”，后面还跟了 4 个空格，如果我们这样使用这样变量来指定别的目录——“$(dir)/file”那么就完蛋了。
还有一个比较有用的操作符是“?=”，先看示例：
FOO ?= bar
其含义是，如果 FOO 没有被定义过，那么变量 FOO 的值就是“bar”，如果 FOO 先前被定义过，那么这条语将什么也不做，其等价于：
ifeq ($(origin FOO), undefined)
FOO = bar
endif

四、追加变量值

我们可以使用“+=”操作符给变量追加值，如：
objects = main.o foo.o bar.o utils.o
objects += another.o
于是，我们的$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o 被追加进去了）
使用“+=”操作符，可以模拟为下面的这种例子：
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
所不同的是，用“+=”更为简洁。
如果变量之前没有定义过，那么，“+=”会自动变成“=”，如果前面有变量定义，那么“+=”会继承于前次操作的赋值符。如果前一次的是“:=”，那么“+=”会以“:=”作为其赋值符，如：
variable := value
variable += more
等价于：
variable := value
variable := $(variable) more
但如果是这种情况：
variable = value
variable += more
由于前次的赋值符是“=”，所以“+=”也会以“=”来做为赋值，那么岂不会发生变量的递补归定义，这是很不好的，所以 make 会自动为我们解决这个问题，我们不必担心这个问题。

五、多行变量
还有一种设置变量值的方法是使用 define 关键字。使用 define 关键字设置变量的值可以有换行，这有利于定义一系列的命令（前面我们讲过“命令包”的技术就是利用这个关键字）。define 指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以 endef 关键字结束。其工作方式和“=”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用 define 定义的命令变量中没有以[Tab]键开头，那么 make 就不会把其认为是命令。
下面的这个示例展示了 define 的用法：
define two-lines
echo foo
echo $(bar)
endef

七、环境变量

make 运行时的系统环境变量可以在 make 开始运行时被载入到 Makefile 文件中，但是如果 Makefile 中已定义了这个变量，或是这个变量由 make 命令行带入，那么系统的环境变量的值将被覆盖。（如果 make 指定了“-e”参数，那么，系统环境变量将覆盖 Makefile 中定义的变量）

因此，如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的Makefile 中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果 Makefile 中定义了 CFLAGS，那么则会使用 Makefile 中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。 当 make 嵌套调用时（参见前面的“嵌套调用”章节），上层 Makefile 中定义的变量会以系统环境变量的方式传递到下层的 Makefile 中。当然，默认情况下，只有通过命令行设置的变量会被传递。而定义在文件中的变量，如果要向下层 Makefile 传递，则需要使用 exprot 关键字来声明。（参见前面章节）当然，我并不推荐把许多的变量都定义在系统环境中，这样，在我们执行不用的 Makefile 时，拥有的是同一套系统变量，这可能会带来更多的麻烦。

八、目标变量

前面我们所讲的在 Makefile 中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量。当然，“自动化变量”除外，如“$<”等这种类量的自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，我样同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。其语法是：

<target ...> : <variable-assignment>

<target ...> : overide <variable-assignment>

<variable-assignment>可以是前面讲过的各种赋值表达式，如“=”、“:=”、“+=”或是“?=”。第二个语法是针对于 make 命令行带入的变量，或是系统环境变量。

这个特性非常的有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。如：

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o
$(CC) $(CFLAGS) prog.o foo.o bar.o
prog.o : prog.c
$(CC) $(CFLAGS) prog.c
foo.o : foo.c
$(CC) $(CFLAGS) foo.c
bar.o : bar.c
$(CC) $(CFLAGS) bar.c
```

在这个示例中，不管全局的$(CFLAGS)的值是什么，在 prog 目标，以及其所引发的所有规则中（prog.o foo.o bar.o 的规则），$(CFLAGS)的值都是“-g”

九、模式变量

在 GNU 的 make 中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。

我们知道，make 的“模式”一般是至少含有一个“%”的，所以，我们可以以如下方式给所有以[.o]结尾的目标定义目标变量：

```
%.o : CFLAGS = -O
```

同样，模式变量的语法和“目标变量”一样：

<pattern ...> : <variable-assignment>
<pattern ...> : override <variable-assignment>
override 同样是针对于系统环境传入的变量，或是 make 命令行指定的变量。
使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

一、示例
下面的例子，判断$(CC)变量是否“gcc”，如果是的话，则使用 GNU 函数编译目标。

```
libs_for_gcc = -lgnu
normal_libs =
foo: $(objects)
ifeq ($(CC),gcc)
$(CC) -o foo $(objects) $(libs_for_gcc)
else
$(CC) -o foo $(objects) $(normal_libs)
endif
```

可见，在上面示例的这个规则中，目标“foo”可以根据变量“$(CC)”值来选取不同的函数库来编译程序。
我们可以从上面的示例中看到三个关键字：ifeq、else 和 endif。ifeq 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else 表示条件表达式为假的情况。endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。
当我们的变量$(CC)值是“gcc”时，目标 foo 的规则是：

```
foo: $(objects)
$(CC) -o foo $(objects) $(libs_for_gcc)
```

而当我们的变量$(CC)值不是“gcc”时（比如“cc”），目标 foo 的规则是：

```
foo: $(objects)
$(CC) -o foo $(objects) $(normal_libs)
```

当然，我们还可以把上面的那个例子写得更简洁一些：

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
libs=$(libs_for_gcc)
else
libs=$(normal_libs)
endif
```

```
foo: $(objects)
$(CC) -o foo $(objects) $(libs)
```

二、语法

条件表达式的语法为：

```
<conditional-directive>
<text-if-true>
endif
```

以及：

```
<conditional-directive>
<text-if-true>
else
<text-if-false>
endif
```

其中<conditional-directive>表示条件关键字，如“ifeq”。这个关键字有四个。

第一个是我们前面所见过的“ifeq”

```
ifeq (<arg1>, <arg2> )
ifeq '<arg1>' '<arg2>'
ifeq "<arg1>" "<arg2>"
ifeq "<arg1>" '<arg2>'
ifeq '<arg1>' "<arg2>"
```

比较参数“arg1”和“arg2”的值是否相同。当然，参数中我们还可以使用 make 的函数。如：

```
ifeq ($(strip $(foo)),)
<text-if-empty>
endif
```

这个示例中使用了“strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。第二个条件关键字是“ifneq”。语法是：

```
ifneq (<arg1>, <arg2> )
ifneq '<arg1>' '<arg2>'
ifneq "<arg1>" "<arg2>"
ifneq "<arg1>" '<arg2>'
ifneq '<arg1>' "<arg2>"
```

其比较参数“arg1”和“arg2”的值是否相同，如果不同，则为真。和“ifeq”类似。

第三个条件关键字是“ifdef”。语法是：ifdef <variable-name>

如果变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然，<variable-name>同样可以是一个函数的返回值。注意，ifdef 只是测试一个变量是否有值，其并不会把变量扩展到当前位置。还是来看两个例子：

示例一：
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif

示例二：
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
第一个例子中，“$(frobozz)”值是“yes”，第二个则是“no”。

第四个条件关键字是“ifndef”。其语法是：

ifndef <variable-name>

这个我就不多说了，和“ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“#”同样也是安全的。“else”和“endif”也
一样，只要不是以[Tab]键开始就行了。

特别注意的是，make 是在读取 Makefile 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“$@”等）放入条件表达式中，因为自动化变量是在运行时才有的。

而且，为了避免混乱，make 不允许把整个条件语句分成两部分放在不同的文件中。

一、make 的退出码

make 命令执行后有三个退出码：

0 —— 表示成功执行。

1 —— 如果 make 运行时出现任何错误，其返回 1。

2 —— 如果你使用了 make 的“-q”选项，并且 make 使得一些目标不需要更新，那么返回 2。

Make 的相关参数我们会在后续章节中讲述。

二、指定 Makefile

前面我们说过，GNU make 找寻默认的 Makefile 的规则是在当前目录下依次找三个文件——“GNUmakefile”、“makefile”和“Makefile”。其按顺序找这三个文件，一旦找到，就开始读取这个文件并执行。

当前，我们也可以给 make 命令指定一个特殊名字的 Makefile。要达到这个功能，我们要使用 make 的“-f”或是“--file”参数（“-- makefile”参数也行）。例如，我们有个 mak efile 的名字是“hchen.mk”，那么，我们可以这样来让 make 来执行这个文件：

make －f hchen.mk

如果在 make 的命令行是，你不只一次地使用了“-f”参数，那么，所有指定的 makefile 将会被连在一起传递给 make 执行。

三、指定目标

一般来说，make 的最终目标是 makefile 中的第一个目标，而其它目标一般是由这个目标连带出来的。这是 make 的默认行为。当然，一般来说，你的 makefile 中的第一个目标是由许多个目标组成，你可以指示 make，让其完成你所指定的目标。要达到这一目的很简单，需在 make 命令后直接跟目标的名字就可以完成（如前面提到的“make clean”形式）任何在 makefile 中的目标都可以被指定成终极目标，但是除了以“-”打头，或是包含了“=”的目标，因为有这些字符的目标，会被解析成命令行参数或是变量。甚至没有被我们明确写出来的目标也可以成为 make 的终极目标，也就是说，只要 make 可以找到其隐含规则推导规则，那么这个隐含目标同样可以被指定成终极目标。

有一个 make 的环境变量叫“MAKECMDGOALS”，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值。这个变量可以让你使用在一些比较特殊的情形下。比如下面的例子：

```
sources = foo.c bar.c
ifneq ( $(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

基于上面的这个例子，只要我们输入的命令不是“make clean”，那么 makefile 会自动包含“foo.d”和“bar.d”这两个 makefile。

使用指定终极目标的方法可以很方便地让我们编译我们的程序，例如下面这个例子：

```
.PHONY: all
all: prog1 prog2 prog3 prog4
```

从这个例子中，我们可以看到，这个 makefile 中有四个需要编译的程序——“prog1”，“prog2”，“prog3”和“prog4”，我们可以使用“make all”命令来编译所有的目标
（如果把 all 置成第一个目标，那么只需执行“make”），我们也可以使用“make prog2”来单独编译目标“prog2”。

即然 make 可以指定所有 makefile 中的目标，那么也包括“伪目标”，于是我们可以根据这种性质来让我们的 makefile 根据指定的不同的目标来完成不同的事。在 Unix 世界中，软件发布时，特别是 GNU 这种开源软件的发布时，其 makefile 都包含了编译、安装、打包等功能。我们可以参照这种规则来书写我们的 makefile 中的目标。

“all”　　这个伪目标是所有目标的目标，其功能一般是编译所有的目标。
“clean”　　这个伪目标功能是删除所有被 make 创建的文件。
“install”　　这个伪目标功能是安装已编译好的程序，其实就是把目标执行文件拷贝到指定的目标中去。
“print”　　这个伪目标的功能是例出改变过的源文件。
“tar”　　这个伪目标功能是把源程序打包备份。也就是一个 tar 文件。
“dist”　　这个伪目标功能是创建一个压缩文件，一般是把 tar 文件压成 Z 文件。或是 gz 文件。
“TAGS”　　这个伪目标功能是更新所有的目标，以备完整地重编译使用。
“check”和“test”　　这两个伪目标一般用来测试 makefile 的流程。

当然一个项目的 makefile 中也不一定要书写这样的目标，这些东西都是 GNU 的东西，但是我想，GNU 搞出这些东西一定有其可取之处（等你的 UNIX 下的程序文件一多时你就会发现这些功能很有用了），这里只不过是说明了，如果你要书写这种功能，最好使用这种名字命名你的目标，这样规范一些，规范的好处就是——不用解释，大家都明白。而且如果你的 makefile 中有这些功能，一是很实用，二是可以显得你的 makefile 很专业（不是那种初学者的作品）。

五、定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有"%"字符。"%"的意思是表示一个或多个任意字符。在依赖目标中同样可以使用"%"，只是依赖目标中的"%"的取值，取决于其目标。

有一点需要注意的是，"%"的展开发生在变量和函数的展开之后，变量和函数的展开发生在 make 载入 Makefile 时，而模式规则中的"%"则发生在运行时。

1、模式规则介绍

模式规则中，至少在规则的目标定义中要包含"%"，否则，就是一般的规则。目标中的"%"定义表示对文件名的匹配，"%"表示长度任意的非空字符串。例如："%.c"表示以".c"结尾的文件名（文件名的长度至少为 3），而"s.%.c"则表示以"s."开头，".c"结尾的文件名（文件名的长度至少为 5）。

如果"%"定义在目标中，那么，目标中的"%"的值决定了依赖目标中的"%"的值，也就是说，目标中的模式的"%"决定了依赖目标中"%"的样子。例如有一个模式规则如下：

%.o : %.c ; <command .....>

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是"a.o b.o"，那么"%c"就是"a.c b.c"。

一旦依赖目标中的"%"模式被确定，那么，make 会被要求去匹配当前目录下所有的文件名，一旦找到，make 就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make 就会产生所有的模式目标，此时，make 关心的是依赖的文件名和生成目标的命令这两件事。

2、模式规则示例

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.

%.o : %.c

$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

其中，"$@"表示所有的目标的挨个值，"$<"表示了所有依赖目标的挨个值。这些奇怪的变量我们叫"自动化变量"，后面会详细讲述。

下面的这个例子中有两个目标是模式的：

%.tab.c %.tab.h: %.y

bison -d $<

这条规则告诉 make 把所有的[.y]文件都以"bison -d <n>.y"执行，然后生成"<n>.tab.c"和"<n>.tab.h"文件。（其中，"<n>" 表示一个任意字符串）。如果我们的执行程序"foo"依赖于文件"parse.tab.o"和"scan.o"，并且文件"scan.o"依赖于文件"parse.tab.h"，如果"parse.y"文件被更新了，那么根据上述的规则，"bison -d parse.y"就会被执行一次，于是，"parse.tab.o"和"scan.o"的依赖文件就齐了。（假设，"parse.tab.o" 由"parse.tab.c"生成，和"scan.o"由"scan.c"生成，而"foo"由"parse.tab.o"和"scan.o"链接生成，

而且 foo 和其[.o]文件的依赖关系也写好，那么，所有的目标都会得到满足）

3、自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@
表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。
$%
仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix 下是[.a]，Windows 下是[.lib]），那么，其值为空。
$<
依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。
$?
所有比目标新的依赖目标的集合。以空格分隔。
$^
所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。
$+
这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。
$*
这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$*"的值就是"dir /a.foo"。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么"$*"也就不能被推导出，但是，如果目标文件的后缀是 make 所识别的，那么"$*"就是除了后缀的那一部分。例如：如果目标是"foo.c"，因为".c"是 make 所能识别的后缀名，所以，"$*"的值就是"foo"。这个特性是 GNU make 的，很有可能不兼容于其它版本的 make，所以，你应该尽量避免使用"$*"，除非是在隐含规则或是静态模式中。如果目标中的后缀是 make 所不能识别的，那么"$*"就是空值。

当你希望只对更新过的依赖文件进行操作时，"$?"在显式规则中很有用，例如，假设有一个函数库文件叫"lib"，其由其它几个 object 文件更新。那么把 object 文件打包的比较有效率的 Makefile 规则是：

```
lib : foo.o bar.o lose.o win.o
ar r lib $?
```

在上述所列出来的自动量变量中。四个变量（$@、$<、$%、$*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上"D"或"F"字样。这是 GNU make 中老版本的特性，在新版本中，我们使用函数"dir"或"notdir"就可以做到了。"D"的含义就是 Directory，就是目录，"F"的含义就是 File，就是文件。

下面是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)
表示"$@"的目录部分（不以斜杠作为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

$(@F)
表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"。

"$(*D)"
"$(*F)"
和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(*D)"返回"dir"，而"$(*F)"返回"foo"

"$(%D)"
"$(%F)"
分别表示了函数包文件成员的目录部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包含了不同的目录很有用。

"$(<D)"
"$(<F)"
分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"
分别表示所有依赖文件的目录部分和文件部分。(无相同的)

"$(+D)"
"$(+F)"
分别表示所有依赖文件的目录部分和文件部分。(可以有相同的)

"$(?D)"
"$(?F)"

分别表示被更新的依赖文件的目录部分和文件部分。

最后想提醒一下的是，对于"$<"，为了避免产生不必要的麻烦，我们最好给$后面的那个特定字符都加上圆括号，比如，"$(<)"就要比"$<"要好一些。

还得要注意的是，这些变量只使用在规则的命令中，而且一般都是"显式规则"和"静态模式规则"（参见前面"书写规则"一章)。其在隐含规则中并没有意义。

4、模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的"%"，或是没有前后缀，直接就是一个"%"。因为"%"代表一个或多个字符，所以在定义好了的模式中，我们把"%"所匹配的内容叫做"茎",例如"%.c"所匹配的文件"test.c"中"test"就是"茎"。因为在目标和依赖目标中同时有"%"时，依赖目标的"茎"会传给目标，当做目标中的"茎"。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行"茎"的传递时，我们需要知道这个步骤。例如有一个模式"e%t"，文件"src/eat" 匹配于该模式，于是"src/a"就是其"茎"，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式"c%r"，那么，目标就是"src/car"。("茎"被传递)

你可以重载内建的隐含规则（或是定义一个全新的)，例如你可以重新构造和内建隐含规则不同的命令，如:

%.o : %.c
$(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)

你可以取消内建的隐含规则，只要不在后面写命令就行。如：
%.o : %.s
同样，你也可以重新定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

二十七、shell 编程

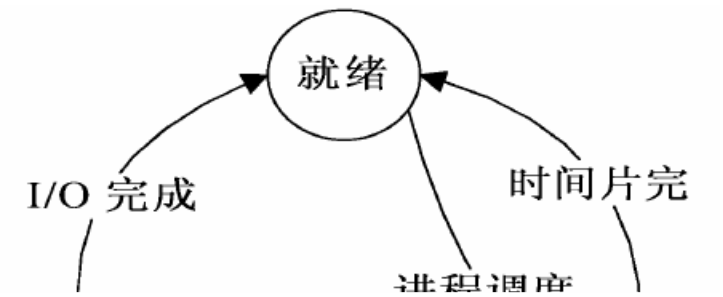
什么是 shell：
简单地讲，就是命令解析器，将用户输入的指令转换为相应的机器能够运行的程序。
种类：
vBourne shell (sh)
vKorn shell (ksh)
vBourne Again shell (bash) vC shell (包括 csh and tcsh) vTENEX/TOPS C shell (tcsh)
Shell 脚本是一个包含一系列命令序
列的文本文件。当运行这个脚本文件时，文件中包含的命令序列将得
到执行。（展示、运行 hello.sh）
Shell 脚本的第一行必须是如下格式：
v#!/bin/sh
符号#!用来指定该脚本文件的解析程序。在上面例子中使用/bin/sh 来解析该脚本。当编辑好脚本后，如果要执行该脚本，还必须使其具有可执行属性。
在 shell 编程中，所有的变量都由字符串组成，并且不需要预先对变量进行声明,例 s1:
有时候变量名很容易与其他文字混淆，比如：
S13:
num=2
echo "this is the $numnd“
思考：输出？Why?
num=2
echo "this is the $numnd"
这并不会打印出"this is the 2nd"，而仅仅打印"this is the "，因为 shell 会去搜索变量 numnd 的值，但是这个变量时没有值的。可以使用花括号来告诉 shell 我们要打印的是 num 变量：
num=2 echo "this is the ${num}nd"这将打印： this is the 2nd
$# :传入脚本的命令行参数个数
v$0 :命令本身（shell 文件名） v$1 :第一个命令行参数
v$2 :第二个命令行参数
$* :所有命令行参数值，在各个参数值之间留有空格
注意：
1. 变量赋值时，“=”左右两边都不能有空格
2. BASH 中的语句结尾不需要分号
If 语句： if-then
fi;
if then
else
fi

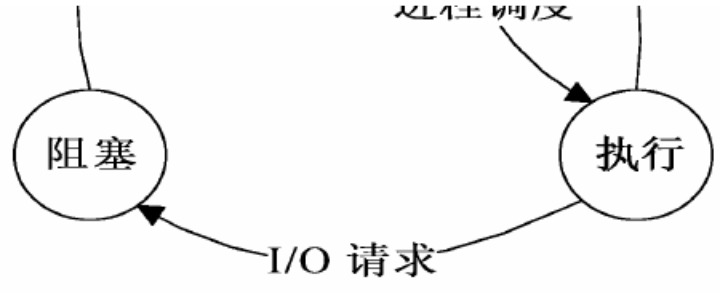
if　then　elif then (注意都要加 then)
vS7:
#!/bin/bash
for day in Sun Mon Tue Wed Thu Fri Sat
do
while 循环的基本结构是:
while [condition]
do
#code block
Done
Case
Easc

二十八、文件编程

在 Linux 系统中，所有打开的文件都对应一个
文件描述符。文件描述符的本质是一个非负整数。当打开一个文件时，该整数由系统来分配。文件描述符的范围是 0 -OPEN_MAX 。早期的 UNIX 版本 OPEN_MAX =19，即允许每个进程同时打开 20 个文件，现在很多系统则将其增加至 1024。
常见的打开标志:
O_RDONLY 只读；O_WRONLY 只写；O_RDWR，读写方式;
O_APPEND：追加方式；O_CREAT：如果当下没有打开的这个文件，就创建
当我们操作完文件以后,需要关闭文
件:
int close(int fd)
fd: 文件描述符，来源?
答：从 open 返回的整数值产生 fd=open（），因此一般必须先打开文件，才能后续操作，符合实际情况
int read(int fd, const void *buf, size_t length)功能： 从文件描述符 fd 所指定的文件中读取
length 个字节到 buf 所指向的缓冲区中，返回值为实际读取的字节数
int write(int fd, const void *buf, size_t length)功能:
把 length 个字节从 buf 指向的缓冲区中写到文件描述符 fd 所指向的文件中，返回值为实际写入的字节数。

二十九、进程调度算法





非抢占：绝对的按顺序。

抢占：

1.先来先服务调度算法

2.短进程优先调度算法

3.高优先级优先调度算法

4.时间片轮转法(都运行固定的时间)

进程间通信：

原因：1.数据传输：一个进程需要将它的数据发送给另一个进程

2.资源共享：多个进程间共享同样的资源

3.通知事件：一个进程需要向另一个或一组进程发送消息，通知它们发生了某种事件

4.进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有操作，并能够及时知道它的状态改变

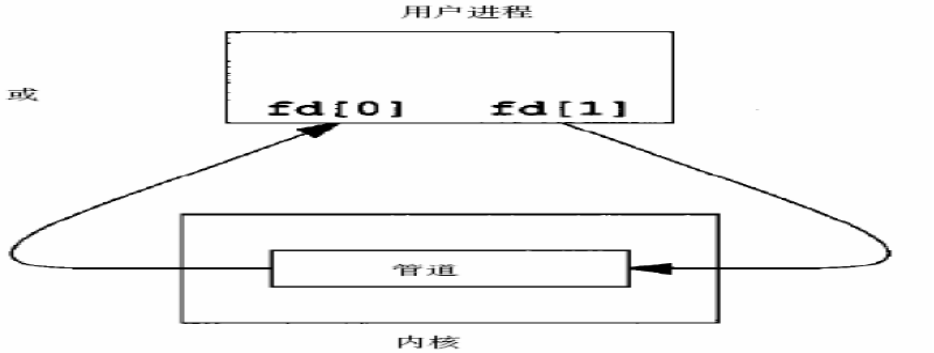
管道通信：管道是单向的、先进先出的，它把一个进程的输出和另一个进程的输入连接在一起。一个进程（写进程）在管道的尾部写入数据，另一个进程（读进程）从管道的头部读出数据。数据被一个进程读出后，将被从管道中删除，其它读进程将不能再读到这些数据。管道提供了简单的流控制机制，进程试图读空管道时，进程将阻塞。同样，管道已经满时，进程再试图向管道写入数据，进程将阻塞

管道包括无名管道和有名管道两种，前者用于父进程和子进程间的通信，后者可用于运行于同一系统中的任意两个进程间的通信。

无名管道由 pipe（）函数创建：

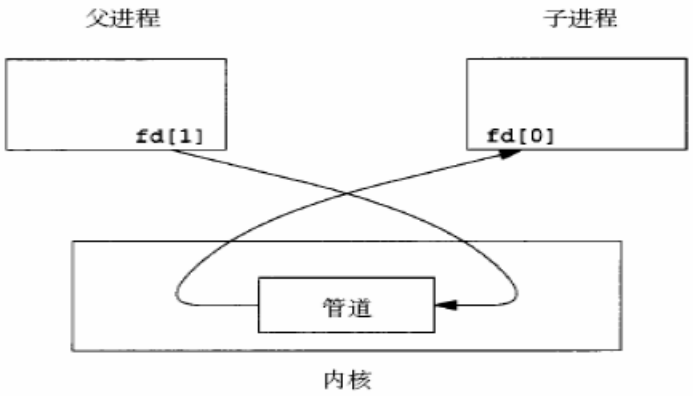
int pipe(int filedis[2]);

当一个管道建立时，它会创建两个文件描述符：filedis[0] 用于读管道，filedis[1] 用于写管道。



关闭管道只需将这两个文件描述符关闭即可，可以使用普通的 close 函数逐个关闭

管道用于不同进程间通信。通常先创建一个管道，再通过 fork 函数创建一个子进程，该子进程会继承父进程所创建的管道。



必须在系统调用 fork()前调用 pipe()，否则子进程将不会继承文件描述符。

为什么有了进程还要有线程：

1.和进程相比，它是一种非常“节俭”的多任务操作方式。在 Linux 系统下，启动一个新的进程

必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。

2.运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的 30 倍左右

3. 线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。

三十、FIQ 和 IRQ 的区别：

一般的中断控制器里我们可以配置与控制器相连的某个中断输入是 FIQ 还是 IRQ,所以一个中断是可以指定为 FIQ 或者 IRQ 的，为了合理，要求系统更快响应，自身处理所耗时间也很短的中断设置为 FIQ，否则就设置了 IRQ。

如果该中断设置为了 IRQ，那么当该中断产生的时候，中断处理器通过 IRQ 请求线告诉 ARM，ARM 就知道有个 IRQ 中断来了，然后 ARM 切换到 IRQ 模式运行。类似的如果该中断设置为 FIQ，那么当该中断产生的时候，中断处理器通过 FIQ 请求线告诉 ARM，ARM 就知道有个 FIQ 中断来了，然后切换到 FIQ 模式运行。

简单的对比的话就是 FIQ 比 IRQ 快，为什么快呢?

1：ARM 的 FIQ 模式提供了更多的 banked 寄存器，r8 到 r14 还有 SPSR，而 IRQ 模式就没有那么多，R8,R9,R10,R11,R12 对应的 banked 的寄存器就没有，这就意味着在 ARM 的 IRQ

模式下，中断处理程序自己要保存 R8 到 R12 这几个寄存器，然后退出中断处理时程序要恢复这几个寄存器，而 FIQ 模式由于这几个寄存器都有 banked 寄存器，模式切换时 CPU 自动保存这些值到 banked 寄存器，退出 FIQ 模式时自动恢复，所以这个过程 FIQ 比 IRQ 快。

2：FIQ 比 IRQ 有更高优先级，如果 FIQ 和 IRQ 同时产生，那么 FIQ 先处理。

3：FIQ 的中断向量地址在 0x0000001C，而 IRQ 的在 0x00000018。(也有的在 FFFF001C 以及 FFFF0018)，写过完整汇编系统的都比较明白这点的差别，18 只能放一条指令，为了不与 1C 处的 FIQ 冲突，这个地方只能跳转，而 FIQ 不一样，1C 以后没有任何中断向量表了，这样可以直接在 1C 处放 FIQ 的中断处理程序，由于跳转的范围限制，至少少了一条跳转指令。

4：IRQ 和 FIQ 的响应延迟有区别

IRQ 的响应并不及时，从 Verilog 仿真来看，IRQ 会延迟几个指令周期才跳转到中断向量处，看起来像是在等预取的指令执行完。FIQ 的响应不清楚，也许比 IRQ 快。

中断延迟:从外部中断请求信号发出到执行对应的中断服务程序 ISR 的第一条指令所需要的时间。通过软件程序设计来缩短中断延迟的方法有：中断优先级和中断嵌套。

三十一、中断的上半部分和下半部分

上半部是不能中断的，下半部是可以中断的。对于适时要求高的,必须放在上半部下半部的实现主要是通过软中断 ,tasklet,和工作队列来实现的.首先需要了解一下中断的概念：一个“中断”仅仅是一个信号，当硬件需要获得处理器对它的关注时，就可以发送这个信号。内核维护了一个中断信号线的注册表，该注册表类似于 I/O 端口的注册表。模块在使用中断前要先请求一个中断通道（或中断请求 IRQ），然后在使用后释放该通道。用到的 API 就是 request_irq()以及 free_irq()。注意在调用 request_irq()和 free_irq()的时机最好是在设备第一次打开和最后一次关闭之后。

对于中断处理例程来讲，它的一个典型的任务就是：如果中断通知进程所等待的事件已经发生，比如新的数据到达就会唤醒在该设备上休眠的进程。无论是快速还是慢速处理例程，程序员都应该编写执行事件尽可能短的处理例程。如果需要执行一个长时间的计算任务，做好的办法就是使用上下半部处理机制，以便让工作在更安全的时间里调度计算任务。上半部的功能是响应中断。当中断发生时，它就把设备驱动程序中中断处理例程的下

半部挂到设备的下半部执行队列中去，然后继续等待新的中断到来。这样一来，上半部的执行速度就会很快，它就可以接受更多它负责的设备所产生的中断了。上半部之所以快，是因为它是完全屏蔽中断的，如果它没有执行完，其他中断就不能及时地处理，只能等到这个中断处理程序执行完毕以后。所以要尽可能多的对设备产生的中断进行服务和处理，中断处理程序就一定要快。

下半部的功能是处理比较复杂的过程。下半部和上半部最大的区别是可中断，而上半部却不可中断。下半部几乎完成了中断处理程序所有的事情，因为上半部只是将下半部排到了它们所负责的设备中断的处理队列中去，然后就不做其它的处理了。下半部所负责的工作一般是查看设备以获得产生中断的事件信息，并根据这些信息（一般通过读设备上的寄存器得来）进行相应的处理。下半部是可中断的，所以在运行期间，如果其它设备产生了中断，这个下半部可以暂时的中断掉，等到那个设备的上半部运行完了，再回头运行这个下半部。

这上面大多数都是我们在查阅资料和听老师讲课获取的知识内容但是具体是如何实现的呢?

也许我们也知道 tasklet 和 work_queue。知道这两个是完成下半部的。在具体的过程中我们将如何实现呢?

此处就是通过创建工作队列的方式我们把初始化的操作都放在 irq 之前完成，在 irq_handler()中将具体的工作提交到工作队列中。queue_work 可以完成这个工作任务.

进程的概念:
① 原子量的引入：在修改某个变量的过程中让其不可改变
②互斥锁：保证同一时刻，只能有一个线程访问该对象
③自旋锁：与互斥锁类似，只是在调度机制上略有不同，对于互斥锁，若资源已经占用，自愿申请者只能进入休眠状态，但是自旋锁不会引起调用者休眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在哪里查看是否该自旋锁的保持着已经放了锁（一般非阻塞式自旋）
④使用并发的原因:
1.模块划分：在编程时划分模块，相关的代码写到一起，不相关的代码分开，并发把不同功能的代码分开，即使这些操作可能会同时执行

2.提高性能：⑴任务并行：将一个认为分解为多个任务，并发执行
⑵数据并行：每个线程进行相同的算法，但处理的数据不同

三十二、数据结构：

线性表特点：
①存在唯一的一个被称作“第一个”的数据元素
②存在唯一的一个被称作“最后一个”的数据元素
③除第一个外，集合中的每个数据元素均只有一个前驱
④除第一个外，集合中的每个数据元素均只有一个前驱
线性表包括顺序表和链表：
线性表：用一组地址连续的存储单元存放一个线性表
特点：逻辑上相邻-物理地址相邻
可实现随机存取任意元素
存储空间使用紧凑
缺点：插入，删除需要移动大量的元素
实现随机存取：插入线性表时所有的元素都需要后移
三十三、4 字节对齐的原因：
Arm 结构为 32 位处理系统

```
01.struct A {
02.    char c;
03.    int i;
04.};
05.struct A a;
```

假设变量 a 存放在内存中的起始地址为 0x00，那么其成员变量 c 的起始地址为 0x00，成员变量 i 的起始地址为 0x01，变量 a 一共占用了 5 个字节。当 CPU 要对成员变量 c 进行访问时，只需要一个读周期即可。而如若要对成员变量 i 进行访问，那么情况就变得有点复杂了，首先 CPU 用了一个读周期，从 0x00 处读取了 4 个字节(注意由于是 32 位架构)，然后将 0x01-0x03 的 3 个字节暂存，接着又花费了一个读周期读取了从 0x04-0x07 的 4 字节数据，将 0x04 这个字节与刚刚暂存的 3 个字节进行拼接从而读取到成员变量 i 的值。为了读取这个成员变量 i，CPU 花费了整整 2 个读周期。试想一下，如果数据成员 i 的起始地址被放在了 0x04 处，那么读取其所花费的周期就变成了 1，显然引入字节对齐可以避免读取效率的下降，但这同时也浪费了 3 个字节的空间(0x01-0x03)。

所分配的内存是 4 的整数倍

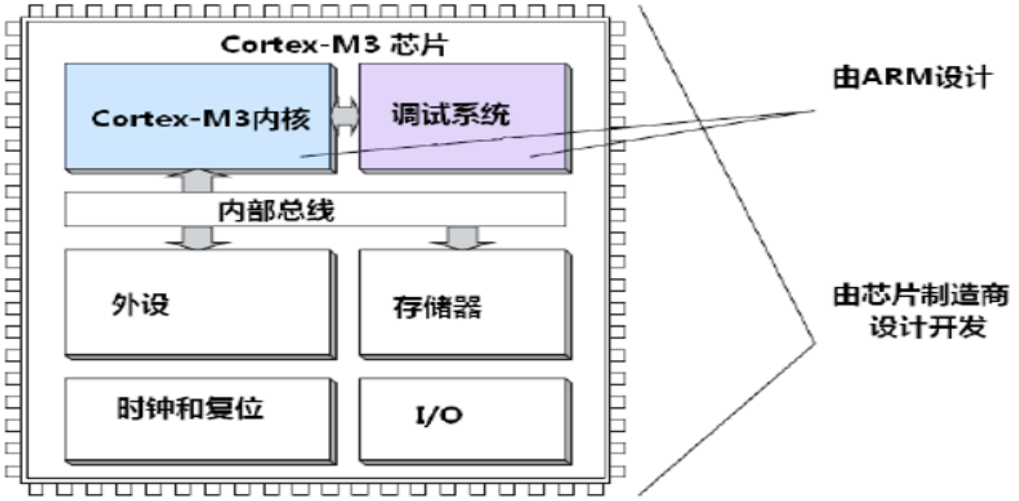
三十三、while（1）和 for（；；）哪一个更好

一般 for(;;)性能更优，这两个;; 空语句，编译器一般会优掉的，直接进入死循环

每循环一次都要判断常量 1 是不是等于零，在这里 while 比 for 多做了这点事。不过从汇编的角度来说，都是一样的代码。

三十四、ARM 架构：

1.架构框图



2. 每个部件有独立时钟控制，上电时仅 SRAM 和 FLASH 时钟有效，适合低功耗应用
3.R0 – R12，通用寄存器，绝大多数 Thumb 指令仅能访问 R0 - R7
32 位 Thumb-2 指令可访问所有寄存器
R13 用作堆栈指针(双堆栈指针)
MSP: 复位后默认，用于 OS 内核、异常程序
PSP: 用于用户应用程序
堆栈只能进行 32 位字访问，且要求 4 字节对齐，CM3 中堆栈采用“向下生长”机制
凡打断程序顺序执行的事件，均称作异常（Exception），包括中断、指令执行错误等
R14（LR），连接寄存器，用于存储程序返回地址，避免了堆栈操作，提高了速度
R15（PC），程序计数寄存器，指示当前正执行指令地址
ARM 7 种模式:

1.用户模式
2.快速中断模式（FIQ）
有 7 个备份寄存器 R8-R14，使得进入中断时，只要不改变 r0-r7，则不需要保存任何寄存器。
3.中断模式（irq）
4.管理模式（svc）
5.数据访问种植模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护
6.系统模式（sys）：运行具有特权的操作系统任务
7.未定义指令中止模式（und）：当未定义的指令执行时进入该模式
除用户模式，其他都属于特权模式。
除用户和系统模式外，其他都是异常模式

三十五、TCP/Udp 协议详解：

计算机网络之 TCP 协议与 UDP 协议

分类：

运输层向它上面应用层提供通信服务，它属于面向通信部分的最高层，同时也是用户功能中的最底层。

两个主机进行通信**实际上就是两个主机中的应用进程互相通信**。应用进程之间的通信又称为**端到端的通信**。

应用层不同进程的报文**通过不同的端口**向下交到运输层，再往下就**共用**网络层提供的服务。

运输层为应用进程之间提供端到端的逻辑通信（但网络层是为主机之间提供逻辑通信）。运输层还要对收到的报文进行差错检测。

运输层需要有两种不同的运输协议：

（1）**用户数据报协议 UDP** (User Datagram Protocol)

UDP 在传送数据之前**不需要先建立连接**。对方的运输层在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 是一种最有效的工作方式。

（2）**传输控制协议 TCP** (Transmission Control Protocol)

TCP 则提供**面向连接**的服务。TCP 不提供广播或多播服务。由于 TCP 要提供**可靠的、面向连接的运输服**务，因此不可避免地增加了许多的开销。

端口号

为了使运行不同操作系统的计算机的应用进程能够互相通信，就必须用统一的方法对 TCP/IP 体系的应用进程进行标志。

解决这个问题的方法就是在运输层使用**协议端口号**(protocol port number)，或通常简称为**端口**(port)。

虽然通信的终点是应用进程，但我们可以把端口想象是通信的终点，因为我们只要把要传送的报文交到目的主机的某一个合适的目的端口，剩下的工作（即最后交付目的进程）就由 TCP 来完成。

端口用一个 16 位端口号进行标志。

端口号只具有本地意义，即端口号只是为了标志本计算机应用层中的各进程。在因特网中不同计算机的相同端口号是没有联系的。

端口号分为两类：

（1）服务端使用的端口号。

这里又分为两类，最重要的一类叫做**熟知端口**（well-known port number），数值一般为 0~1023。管理机构 IANA 把这些端口指派给 TCP/IP 最重要的一些应用程序，让所有用户都知道。

应用程序	FTP	TELNET	SMTP	DNS	TFTP	HTTP	SNMP	SNMP (trap)
熟知端口号	21	23	25	53	69	80	161	162

另一类叫**登记端口号**，

数值为 1024~49151，为没有熟知端口号的应用程序使用的。使用这个范围的端口号必须在 IANA 登记，以防止重复。

（2）客户端使用的端口号

数值为 49152~65535，留给客户进程选择暂时使用，仅在客户进程运行时彩动态选择，因此又叫**短暂端口号**。当服务器进程收到客户进程的报文时，就知道了客户进程所使用的动态端口号。通信结束后，这个端口号可供其他客户进程以后使用。

用户数据报协议 UDP

UDP 只在 IP 的数据报服务之上增加了很少一点的功能，即端口的功能和差错检测的功能。

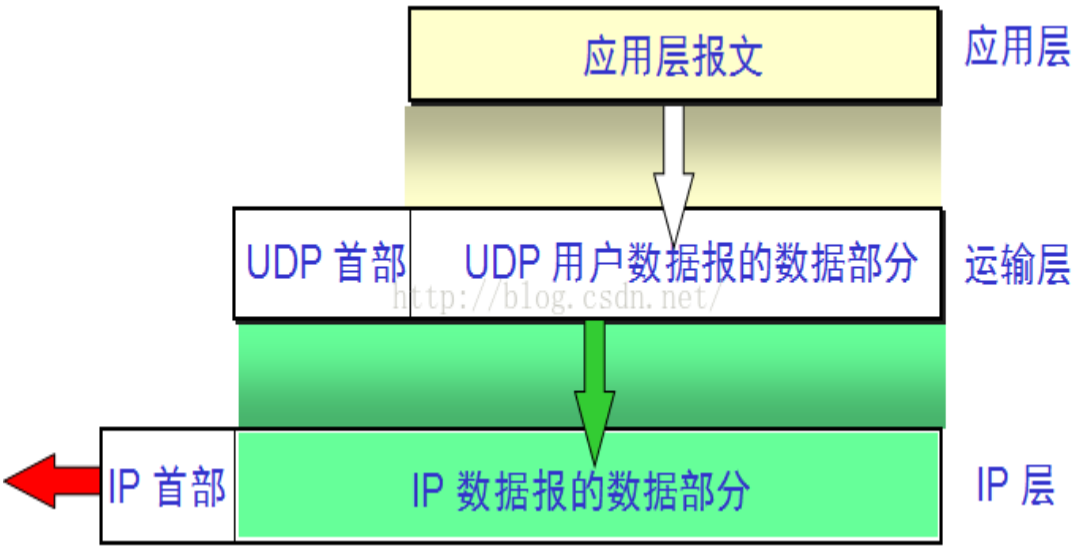
UDP 是无连接的，即**发送数据之前不需要建立连接**。

UDP 使用**尽最大努力交付**，即不保证可靠交付，同时也不使用拥塞控制。

UDP 是面向报文的。UDP 没有拥塞控制，很适合多媒体通信的要求。

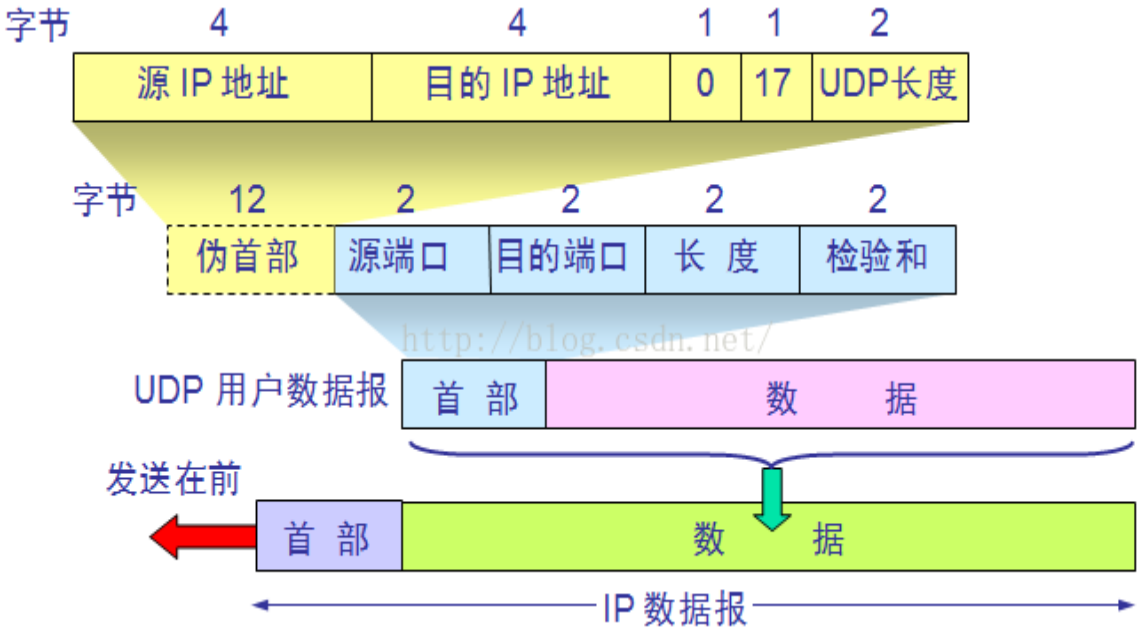
UDP 支持一对一、一对多、多对一和多对多的交互通信。

UDP 的首部开销小，只有 **8 个字节**。



发送方 UDP 对应用程序交下来的报文，在添加首部后就向下交付 IP 层。UDP 对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。应用层交给 UDP 多长的报文，UDP 就照样发送，即一次发送一个报文。接收方 UDP 对 IP 层交上来的 UDP 用户数据报，在去除首部后就原封不动地交付上层的应用进程，一次交付一个完整的报文。所以应用程序必须选择合适大小的报文。

UPD 首部格式如下：



传输控制协议 TCP

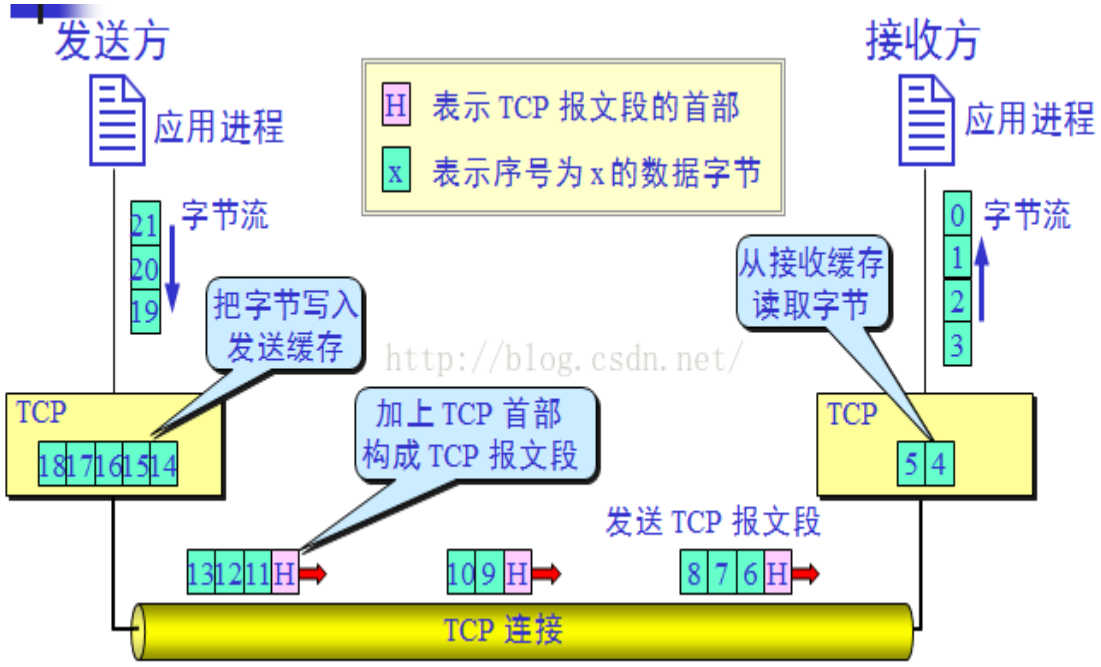
TCP 是面向连接的运输层协议，就是说应用程序在使用 TCP 协议之前，必须先建立 TCP 连接，传送完数据之后，必须释放连接。

每一条 TCP 连接只能有两个端点(endpoint), 每一条 TCP 连接**只能是点对点的（一对一）的**。

TCP 提供**可靠交付的服务**，保证数据无差错、不丢失、不重复、按序到达。

TCP 提供全双工通信。

TCP 是面向字节流，下图展示了字节流的发送、接受过程：



TCP 连接是一条虚连接而不是一条真正的物理连接。TCP 对应用进程一次把多长的报文发送到 TCP 的缓存中是不关心的。TCP 根据对方给出的窗口值和当前网络拥塞的程度来决定一个报文段应包含多少个字节（UDP 发送的报文长度是应用进程给出的）。TCP 可把太长的数据块划分短一些再传送。TCP 也可等待积累有足够多的字节后再构成报文段发送出去。

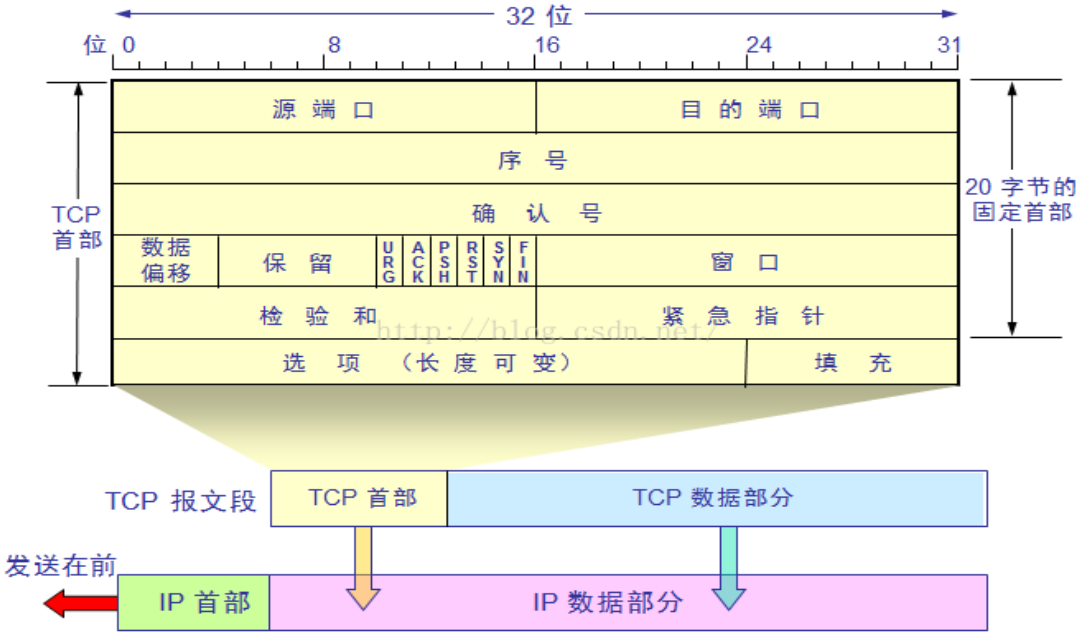
TCP 连接的端点不是主机，不是主机的 IP 地址，不是应用进程，也不是运输层的协议端口。TCP 连接的端点叫做**套接字**(socket)或插口。
端口号拼接到 IP 地址即构成了套接字。

套接字 socket = (IP 地址: 端口号)

每一条 TCP 连接唯一地被通信两端的两个端点（即两个套接字）所确定。即：

TCP 连接 ::= {socket1, socket2}
={(IP1: port1), (IP2: port2)}

TCP 虽然是面向字节流的，但 TCP 传送的数据单元却是报文段。一个 TCP 报文段分为首部和数据两部分。
TCP 报文段首部的前 20 个字节是固定的，后面的字节是根据需要增加的。首部格式如下：



TCP 的连接控制

运输连接就有三个阶段，即：**连接建立**、**数据传送**和**连接释放**。运输连接的管理就是使运输连接的建立和释放都能正常地进行。

连接建立过程中要解决以下三个问题：

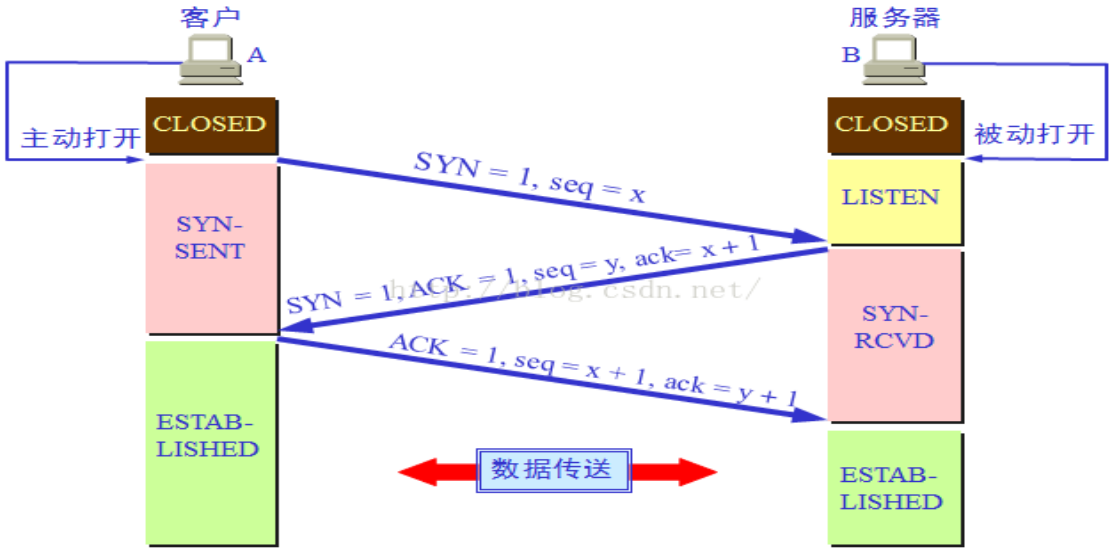
（1）要使每一方能够确知对方的存在。

（2）要允许双方协商一些参数（如最大报文段长度，最大窗口大小，服务质量等）。

（3）能够对运输实体资源（如缓存大小，连接表中的项目等）进行分配。

TCP 连接的建立都是采用**客户服务器方式**。主动发起连接建立的应用进程叫做客户(client)。被动等待连接建立的应用进程叫做服务器(server)。

TCP 的连接建立（三次握手）



上图中，B 的服务器进程先创建**传输控制块 TCB**，准备接受客户进程的连接请求。然后服务器进程就处于 **LISTEN**（监听）状态，等待客户的连接请求。

A 的 TCP 客户进程也是首先创建**传输控制块 TCB**，然后向 B 发出连接请求报文段，这时首部中的同步位 SYN=1，同时选择一个初始序号 seq=x。TCP 规定，SYN 报文段不能携带数据，但要消耗掉一个序号。这时，TCP 客户进程进入 **SYN-SENT**（同步已发送）状态。

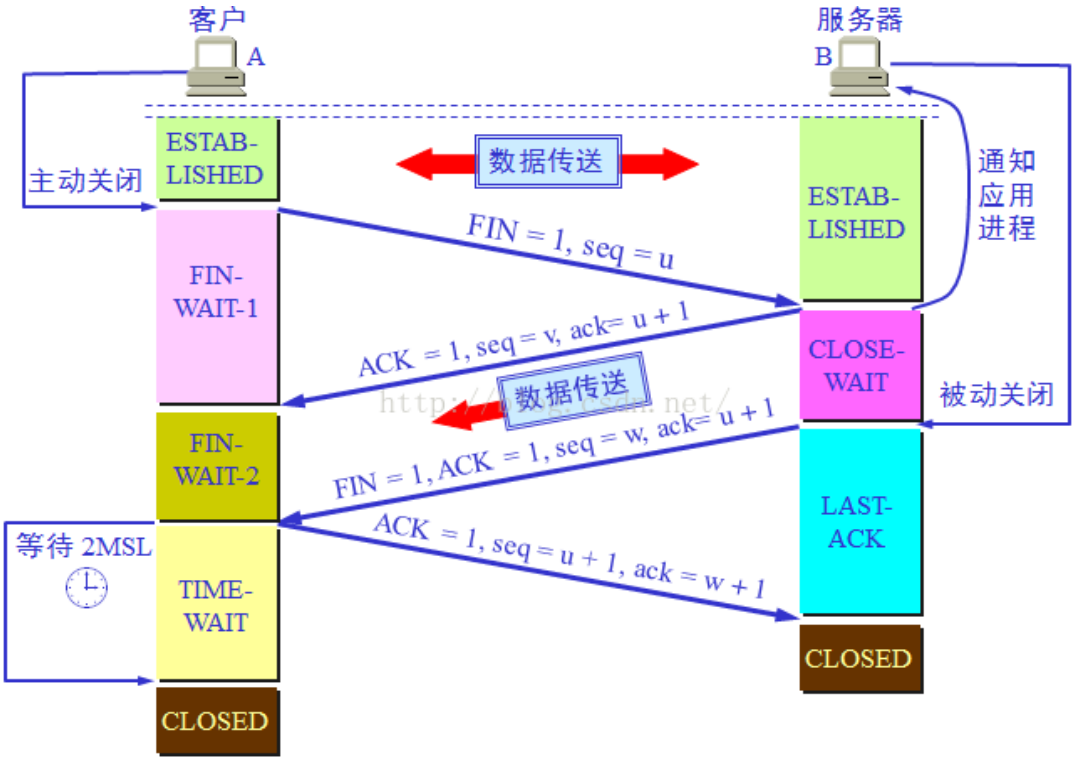
B 收到连接请求报文段后，如同意建立连接，则向 A 发送确认。在确认报文段中应把 SYN 位和 ACK 为都置 1，确认好 ack=x+1，同时也为自己选择一个初始序号 seq=y。这个报文段也不能携带数据，但同样要消耗一个序号。这时 TCP 服务器进程进入 **SYN-RCVD**（同步收到）状态。

TCP 客户进程收到 B 的确认后，还要向 B 给出确认。确认报文段的 ACK 置 1.确认号 ack=y+1，而自己的序号 seq=x+1。TCP 的标准规定，ACK 报文段可以携带数据，但如果不携带数据则不消耗序号。这时，TCP 连接已建立，A 进入 **ESTABLISHEN**（已建立连接）状态。

当 B 收到 A 的确认后，也进入 **ESTABLISHEN** 状态。

上面的过程称为**三次握手**。

TCP 的连接释放（四次挥手）



数据传输结束后，A 和 B 处于 **ESTABLISHEN** 状态。A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接。A 把连接释放报文段首部的 FIN 置 1，其序号 seq=u，等于前面已经传送过的数据的最后一个字节的加 1.这时 A 进入 **FIN-WAIT-1**（终止等待 1）状态，等待 B 的确认。

B 收到连接释放报文段后即发出确认，确认号是 ack=u+1，而这个报文段自己的序号是 v。等于 B 前面已经传送过的数据的最后一个字节的序号加 1。然后 B 进入 **CLOSE-WAIT**（关闭等待）状态。TCP 服务器进程这时应通知高层应用进程，因而从 A 到 B 这个方向的连接就释放了，这时的 TCP 连接处于**半关闭状态**。即 A 已经没有数据要发送了，但 B 若发送数据，A 仍要接受。

A 收到来自 B 的确认后，就进入 **FIN-WAIT-2**（终止等待 2）状态，等待 B 发出的连接释放报文段。

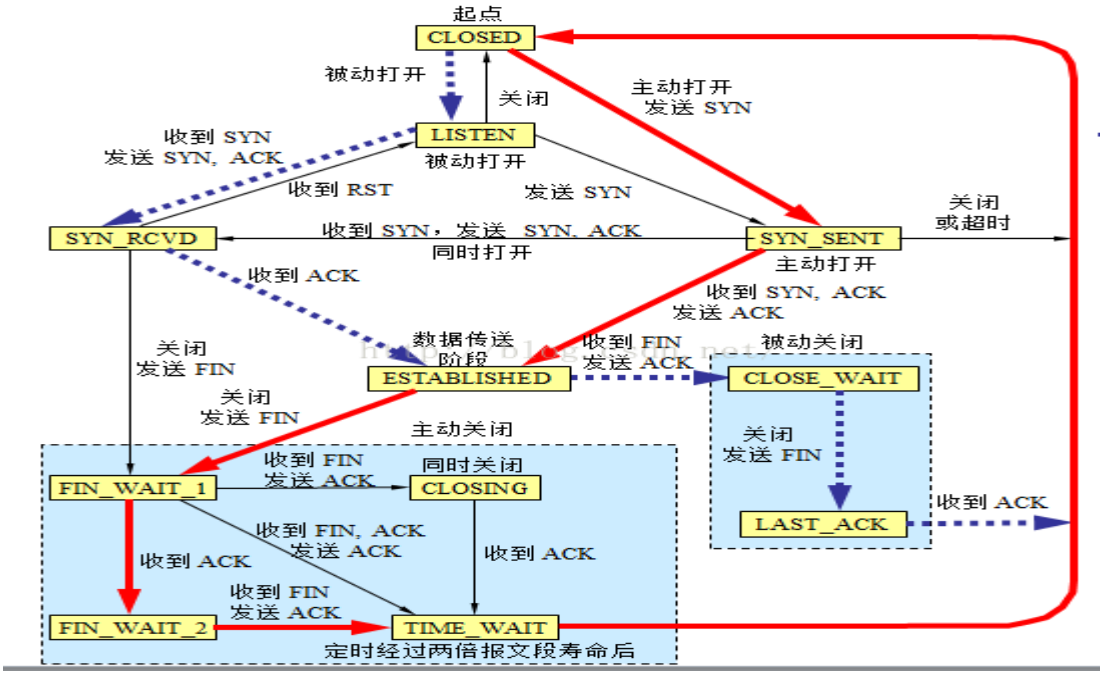
若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放链接。这时 B 发出的报文段必须使 FIN=1。现假定 B 的序号为 w（在半关闭状态 B 可能由发送了一些数据）。B 还必须重复上次已经发送过的确认号 ack=u+1。这时 B 就进入 **LAST-ACK**（最后确认）状态，等待 A 的确认。

A 在收到 B 的连接释放报文段后，必须对此发出确认。在确认报文段中把 ACK 置 1，确认

号 ack=w+1，而自己的序号是 seq=u+1。然后进入到 TIME-WAIT（时间等待）状态。请注意，现在 TCP 连接还没有释放掉。必须经过**时间等待计时器**设置的时间 2MSL 后，A 才进入到 **CLOSED** 状态。时间 MSL 叫做**最长报文段寿命**（Maximum Segment Lifetime）。

TCP 的有限状态机

TCP 有限状态机的图中每一个方框都是 TCP 可能具有的状态。每个方框中的大写英文字符串是 TCP 标准所使用的 TCP 连接状态名。状态之间的箭头表示可能发生的状态变迁。箭头旁边的字，表明引起这种变迁的原因，或表明发生状态变迁后又出现什么动作。



图中有三种不同的箭头：

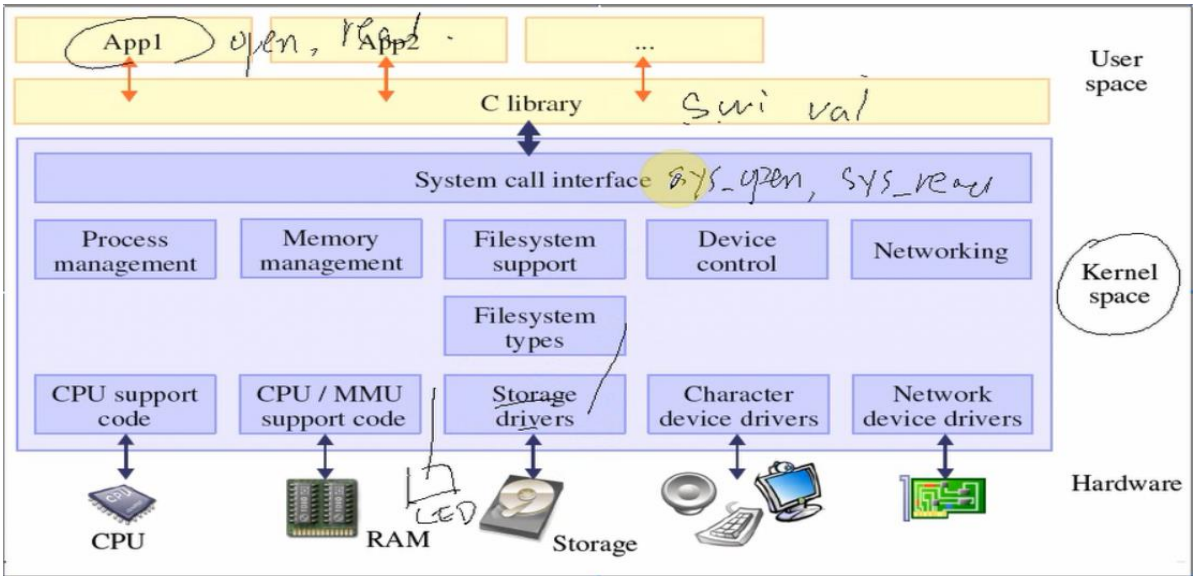
粗实线箭头表示对客户进程的正常变迁。

粗虚线箭头表示对服务器进程的正常变迁。

另一种细线箭头表示异常变迁。

驱动：

涉及的时钟一般由内核打印。Dmesg
时钟分频 1:4:8，FCLOCK=400MHZ，HCLK=100MHZ，PCLK=50MHZ
驱动的概念:　C 库中通过 open/read/write/seek 等来操作文件，所谓字符设备驱动，简单来说就是实现这几个函数的具体内容，linux 能把设备抽象为文件，用户调用 open/read/write/seek 对抽象的文件进行操作就可以操作实际硬件设备（或抽象的设备）。所以字符设备驱动的重点，在于编写内核空间的 open/read/write/seek 等函数。
C 库属于应用层，驱动属于内核
当应用调用 read ，write，read 实际执行 swi 指令，会引发异常，当发生异常进入内核异常处理函数。
有个系统调用接口，根据发生中断的原因（swi 传入的值不同），调用不同的处理函数。
调用函数接口下层是 VFS ，sys_open,sys_write,sys_read，实现不一样的行为
再下层为驱动函数



1．Input 输入子系统

驱动框架：

1. 写出需要的驱动函数。
2. 注册 file_operations 结构体
3. 注册 major=register_chrdev(0,"first_drv",&first_drv_fops); //名字可以随便写 .注册
   根据主设备在 VFS 生成一个字符设备数组，以主设备号为索引 （类型和主设备号）
4. 入口
5. 出口
6. 修饰

cat proc/devices 查看驱动设备

创建设备节点，才能被打开

mknod /dev/xxx c 111 0 (手动)

mdev （自动）：根据 system 信息创建设备节点，创建一个类，类下面创建一个设备

嵌入式对寄存器的操作*((volatile unsigned char *)0x530000000)

drivers/input/input.c：

```
input_init > err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
```

输入子系统：通用的系统，应用程序可以直接调用的。

我们要做的是写出驱动融入输入子系统

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
};
```

Input.c 只是起中转作用

问：怎么读按键?

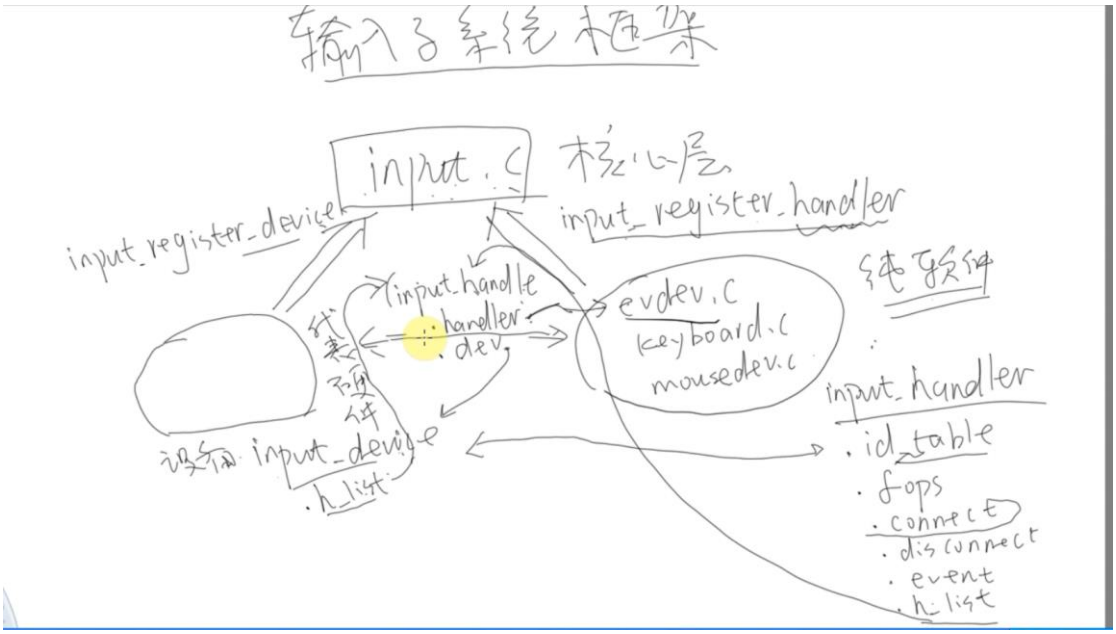
input_open_file

```
struct input_andler *handler = input_table[iminor(inode) >> 5];
new_fops = fops_get(handler->fops)  //  =>&evdev_fops
file->f_op = new_fops; //当前文件获得新的结构体
err = new_fops->open(inode, file);  打开新的 open
```

app: read > ... > file->f_op->read

input_table 数组由谁构造?

input_register_handler



注册 input_handler:
input_register_handler

```
// 放入数组
input_table[handler->minor >> 5] = handler;

// 放入链表
list_add_tail(&handler->node, &input_handler_list);

// 对于每个 input_dev，调用 input_attach_handler
list_for_each_entry(dev, &input_dev_list, node)
    input_attach_handler(dev, handler); // 根据 input_handler 的 id_table 判断能否支持这个 input_dev
```

注册输入设备:
input_register_device
```
    // 放入链表
    list_add_tail(&dev->node, &input_dev_list);

    // 对于每一个 input_handler，都调用 input_attach_handler
    list_for_each_entry(handler, &input_handler_list, node)
        input_attach_handler(dev, handler); // 根据 input_handler 的 id_table 判断能否支持这个 input_dev
```

input_attach_handler
```
    id = input_match_device(handler->id_table, dev);

    error = handler->connect(handler, dev, id);
```

注册 input_dev 或 input_handler 时，会两两比较左边的 input_dev 和右边的 input_handler，
根据 input_handler 的 id_table 判断这个 input_handler 能否支持这个 input_dev，
如果能支持，则调用 input_handler 的 connect 函数建立"连接"

怎么建立连接?
1. 分配一个 input_handle 结构体
2.
```
    input_handle.dev = input_dev;   // 指向左边的 input_dev
    input_handle.handler = input_handler;   // 指向右边的 input_handler
```
3. 注册:
```
    input_handler->h_list = &input_handle;
    inpu_dev->h_list      = &input_handle;
```

evdev_connect
```
    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL); // 分配一个 input_handle

    // 设置
    evdev->handle.dev = dev;   // 指向左边的 input_dev
    evdev->handle.name = evdev->name;
    evdev->handle.handler = handler;   // 指向右边的 input_handler
    evdev->handle.private = evdev;

    // 注册
    error = input_register_handle(&evdev->handle);
```

怎么读按键?

app: read

.....

evdev_read

// 无数据并且是非阻塞方式打开，则立刻返回

if (client->head == client->tail && evdev->exist && (file->f_flags & O_NONBLOCK))

return -EAGAIN;

// 否则休眠

retval = wait_event_interruptible(evdev->wait,

client->head != client->tail || !evdev->exist);

谁来唤醒?

evdev_event

wake_up_interruptible(&evdev->wait);

evdev_event 被谁调用?

猜：应该是硬件相关的代码，input_dev 那层调用的

在设备的中断服务程序里，确定事件是什么，然后调用相应的 input_handler 的 event 处理函数

gpio_keys_isr

// 上报事件

input_event(input, type, button->code, !!state);

input_sync(input);

input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)

struct input_handle *handle;

list_for_each_entry(handle, &dev->h_list, d_node)

if (handle->open)

handle->handler->event(handle, type, code, value);

怎么写符合输入子系统框架的驱动程序?

1. 分配一个 input_dev 结构体
2. 设置
3. 注册 input_dev 结构体 ，并不是 register_chrdev(0,"first_drv",&first_drv_fops)，输入子系统已经写好
4. 硬件相关的代码，比如在中断服务程序里上报事件

以前写驱动：5 步

现在：输入子系统：

```
struct input_dev {

        void *private;

        const char *name;
        const char *phys;
        const char *uniq;
        struct input_id id;

        unsigned long evbit[NBITS(EV_MAX)];      // 表示能产生哪类事件
        unsigned long keybit[NBITS(KEY_MAX)]; // 表示能产生哪些按键
        unsigned long relbit[NBITS(REL_MAX)]; // 表示能产生哪些相对位移事件, x,y,滚轮
        unsigned long absbit[NBITS(ABS_MAX)]; // 表示能产生哪些绝对位移事件, x,y
        unsigned long mscbit[NBITS(MSC_MAX)];
        unsigned long ledbit[NBITS(LED_MAX)];
        unsigned long sndbit[NBITS(SND_MAX)];
        unsigned long ffbit[NBITS(FF_MAX)];
        unsigned long swbit[NBITS(SW_MAX)];
```

测试:

1.

hexdump /dev/event1 (open(/dev/event1), read(),) hexdump:十六进制显示，真实的数据

```
              秒            微秒       类  code       value
0000000 0bb2 0000 0e48 000c 0001 0026 0001 0000
0000010 0bb2 0000 0e54 000c 0000 0000 0000 0000
0000020 0bb2 0000 5815 000e 0001 0026 0000 0000
0000030 0bb2 0000 581f 000e 0000 0000 0000 0000
```

2. 如果没有启动 QT:

cat /dev/tty1

按:s2,s3,s4

就可以得到 ls

或者:

exec 0</dev/tty1

然后可以使用按键来输入

3. 如果已经启动了 QT:

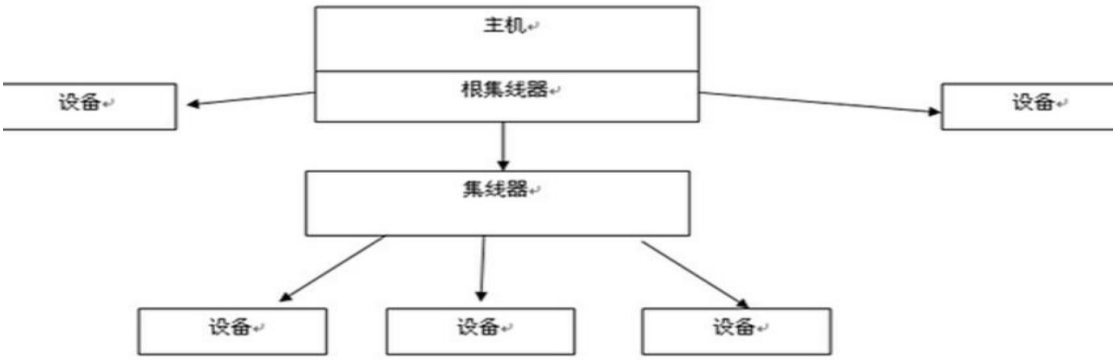
可以点开记事本

然后按:s2,s3,s4

2、usb 驱动

2.1 Usb 概述:

USB 协议有两种：USB1.1 和 USB2.0。USB2.0 和 USB1.1 完全兼容。USB1.1 支持的数据传输率为 12Mbps 和 1.5Mbps（用于慢速外设），USB2.0 支持的数据传速率可达 480Mbps。在普通用户看来，USB 系统就是外设通过一根 USB 电缆和 PC 机连接起来。通常把外设称为 USB 设备，把其所连接的 PC 机称为 USB 主机。将指向 USB 主机的数据传输方向称为上行通信，把指向 USB 设备的数据传输方向称为下行通信。
USB 网络采用阶梯式星形拓扑结构，如图 16-1。一个 USB 网络中只能有一个主机。主机内设置了一个根集线器，提供了主机上的初始附属点。



对于每个 usb 系统来说，都有一个称为 HOST 主机控制器的设备，该控制器和一个根 Hub 作为一个整体。这个根 Hub 下可接多级的 Hub，每个子 Hub 可以接多个子 Hub，每个 usb 最多接 127 个设备，Hub 是与主控制器类似

主机定时对集线器的状态进行查询。当一个新设备接入集线器时，主机会检测到集线器状态改变，主机发出一个命令使该端口有效并对其进行设置。位于这个端口上的设备进行响应，主机收到关于设备的信息，主机的操作系统确定对这个设备使用那种驱动程序，接着设备被

分配一个唯一标识的地址，范围从 0~127，其中 0 为所有的设备在没有分配惟一地址时使用的默认地址。主机向它发出内部设置请求。当一个设备从总线上移走时，主机就从其可用资源列表中将这个设备删除。
USB 的所有数据通信（不论是上行通信还是下行通信）都由 USB 主机启动，所以 USB 主机在整个数据传输过程中占据着主导地位。在 USB 系统中只允许有一个主机。从开发人员的角度看，USB 主机可分为三个不同的功能模块：客户软件、USB 系统软件和 USB 总线接口。
(1) 客户软件
(2) USB 系统软件
(3) USB 总线接口
Usb 系统基本概念

2.2 USB 逻辑组织：

包含 4 部分，设备、配置、接口、端点
一个 usb 硬件包括一个设备，一个设备包括一个或者多个配置，
一个配置通常包括一个或者多个接口，一个接口包括 0 个或多个端点。
例如：一个 usb 包括音频、视频、旋钮、按钮；
配置 1：音频（接口）+旋钮（接口）
配置 2：音频（接口）+视频（接口）
所有接口需要一个驱动程序，因此一个 usb 可能包括多个驱动程序
Usb 总线⇔高速公路
收发的数据⇔汽车
Usb 端点⇔收费口的入口或者出口
一次传输由 1 个或者多个事务：IN,OUT,SETUP
一个事务由 1 个或者多个包构成(packet):包:令牌包(setup),数据包(data),握手包(ACK),特殊包
一个包邮多个域构成：同步（sync），标识（PID），地址（ADDR），端点域（ENDP）,帧号域（FRAM），数据域（DATA）,校验域（CRC）
配置 VS 设置：
手机，通话，U 盘（配置）
作为通话，设置模式

2.3 usb 设备枚举：

Usb 设备在正常工作前，要做的第一件事：枚举
枚举：让 Host 识别这个 usb 设备，分配地址，建立连接
1. 获取设备描述符：(固定长度、未知)
2. 复位
3. 设置地址：发送设置地址请求->应答->设置
4. 再次设置设备描述符：（完整的描述符）
5. 获取配置描述符
6. 获取接口，端点描述符
7. 获取字符串描述符
8. 选择设备配置

2.4 usb urb(usb request block ,urb)

是 usb 设备驱动中用来描述与 usb 设备通信所用的基本载体和核心数据结构，是 usb 与设备通信的“电波”

2.4 Usb 处理流程：

①usb 设备驱动程序创建并出示化一个访问特定 usb 设备特定端点的 urb，并提交给 usb core
②usb core 提交该 urb 到 usb 主控制器
③usb 主控制器根据 urb 描述的信息，来访问 usb 设备
④设备访问结束，usb 主控通知 usb 设备驱动程序

2.5 usb 物理特性：

USB 使用一根屏蔽的 4 线电缆与网络上的设备进行互联。数据传输通过一个差分双绞线进行，这两根线分别标为 D+和 D-，另外两根线是 Vcc 和 Ground，其中 Vcc 向 USB 设备供电。使用 USB 电源的设备称为总线供电设备，而使用自己外部电源的设备叫做自供电设备。为了避免混淆，USB 电缆中的线都用不同的颜色标记，如表 16-1 所示。

引脚编号	信号名称	缆线颜色
1	Vcc	红
2	Data-(D-)	白
3	Data+(D+)	绿
4	Ground	黑

Usb 信号：
差分信号技术特点

传统的传输方式大多使用“正信号”或者“负信号”二进制表达机制，这些信号利用单线传输。用不同的信号电平范围来分别表示 1 和 0，它们之间有一个临界值，如果在数据传输过程中受到中低强度的干扰，高低电平不会突破临界值，那么信号传输可以正常进行。但如果遇到强干扰，高低电平突破临界值，由此造成数据传输出错。差分信号技术最大的特点是：必须使用两条线路才能表达一个比特位，用两条线路传输信号的压差作为判断 1 还是 0 的依据。这种做法的优点是具有极强的抗干扰性。倘若遭受外界强烈干扰，两条线路对应的电平同样会出现大幅度提升或降低的情况，但二者的电平改变方向和幅度几乎相同，电压差值就可始终保持相对稳定，因此数据的准确性并不会因干扰噪声而有所降低。

USB 的数据包使用反向不归零编码（NRZI）。图 16-3 描述了在 USB 电缆段上传输信息的步骤。反向不归零编码由传送信息的 USB 代理程序完成；然后，被编码的数据通过差分驱动器送到 USB 电缆上；接着，接收器将输入的差分信号进行放大，将其送给解码器。使用该编码和差动信号传输方式可以更好地保证数据的完整性并减少噪声干扰。

2.6 USB 通信格式:

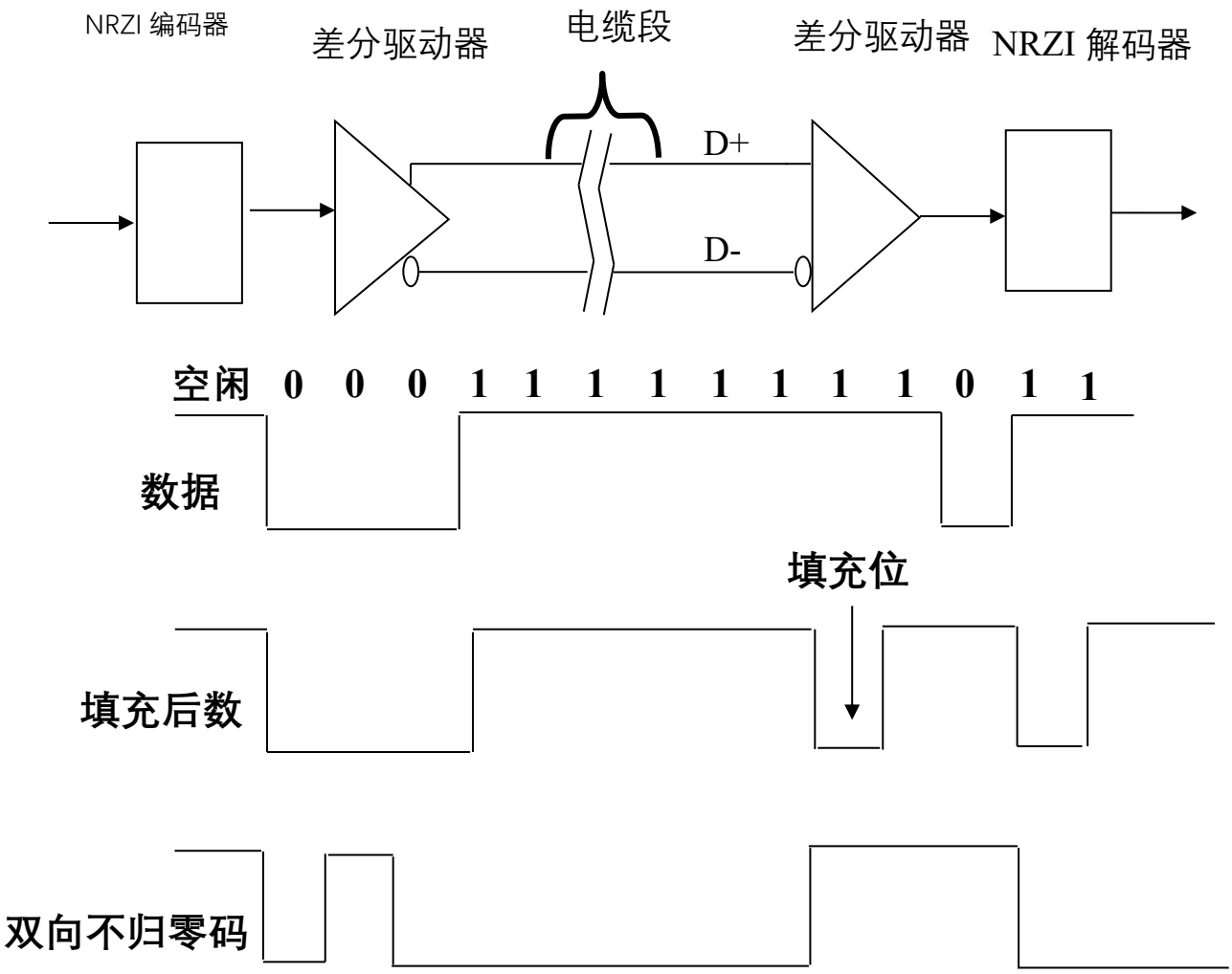


图 16-5 在 USB 电缆上使用双向不归零编码和差动信号的传输

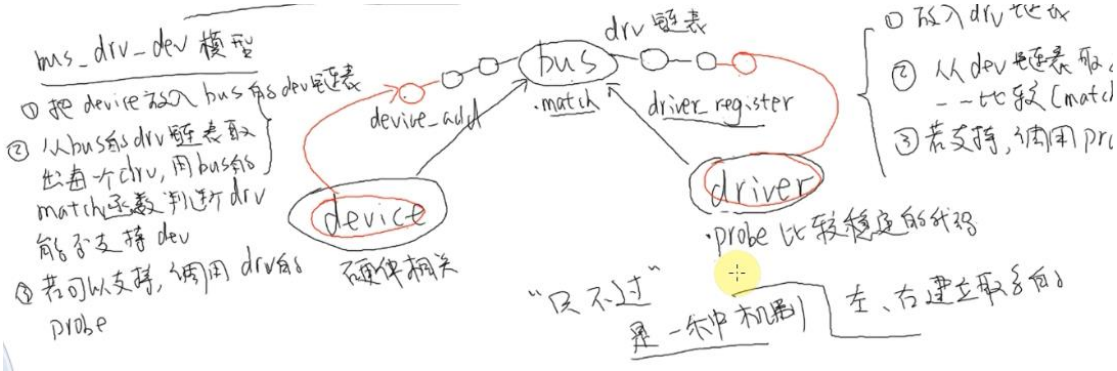
USB 的数据包使用反向不归零编码（NRZI)。图 16-3 描述了在 USB 电缆段上传输信息的步骤。反向不归零编码由传送信息的 USB 代理程序完成；然后，被编码的数据通过差分驱动器送到 USB 电缆上；接着，接收器将输入的差分信号进行放大，将其送给解码器。使用该编码和差动信号传输方式可以更好地保证数据的完整性并减少噪声干扰。使用反向不归零编码方式可以保证数据传输的完整性，而且不要求传输过程中有独立的时钟信号。反向不归零编码不是一个新的编码方式。它在许多方面都有应用。图 16-4 给出了一个数据流和编码之后的结果。在反向不归零编码时，遇到“0”转换，遇到“1”保持。反向不归零码必须保持与输入数据的同步性，以确保数据采样正确。反向不归零码数据流必须在一个数据窗口被采样，无论前一个位时间是否发生过转换。解码器在每个位时间采样数据以检查是否有转换。

若重复相同的“1”信号一直进入时，就会造成数据长时间无法转换，逐渐的积累，而导致接收器最终丢失同步信号的状况，使得读取的时序会发生严重的错误。因此，在 NRZI 编码之间，还需执行所谓的位填充的工作。位填充要求数据流中如果有连续的六个“1”就要强行转换。这样接收器在反向不归零码数据流中最多每七个位就检测到一次跳转。这样就

保证了接收器与输入数据流保持同步。反向不归零码的发送器要把“0”（填充位）插到数据流中。接收器必须被设计成能够在连续的六个“1”之后识别一个自动跳转，并且立即扔掉这六个“1”之后的“0”位。

图 16-5 的第一行是送到接收器的原始数据。注意数据流包括连续的八个“1”。第二行表示对原始数据进行了位填充，在原始的第六个和第七个“1”之间填入了一个“0”。第七个“1”延时一个位时间让填充位插入。接收器知道连续六个“1”之后将是一个填充位，所以该位就要被忽略。注意，如果原始数据的第七个位是“0”，填充位也同样插入，在填充过的数据流中就会有两个连续的“0”。

分层分离思想：
分离：硬件相关，相对稳定
‘分层：向上注册,每层有自己的任务



Probe 函数里面做的事情完全由自己决定

现象：把 USB 设备接到 PC
1. 右下角弹出"发现 android phone"
2. 跳出一个对话框，提示你安装驱动程序

问 1. 既然还没有"驱动程序"，为何能知道是"android phone"
答 1. windows 里已经有了 USB 的总线驱动程序，接入 USB 设备后，是"总线驱动程序"知道你是"android phone"
提示你安装的是"设备驱动程序"

USB 总线驱动程序负责：识别 USB 设备，给 USB 设备找到对应的驱动程序

问 2. USB 设备种类非常多，为什么一接入电脑，就能识别出来?
答 2. PC 和 USB 设备都得遵守一些规范。
比如：USB 设备接入电脑后，PC 机会发出"你是什么"?
USB 设备就必须回答"我是 xxx"，并且回答的语言必须是中文
USB 总线驱动程序会发出某些命令想获取设备信息(描述符)，
USB 设备必须返回"描述符"给 PC

问 3. PC 机上接有非常多的 USB 设备，怎么分辨它们?
USB 接口只有 4 条线: 5V,GND,D-,D+
答 3. 每一个 USB 设备接入 PC 时，USB 总线驱动程序都会给它分配一个编号

接在 USB 总线上的每一个 USB 设备都有自己的编号(地址)
PC 机想访问某个 USB 设备时，发出的命令都含有对应的编号(地址)

问 4. USB 设备刚接入 PC 时，还没有编号；那么 PC 怎么把"分配的编号"告诉它？
答 4. 新接入的 USB 设备的默认编号是 0，在未分配新编号前，PC 使用 0 编号和它通信。

问 5. 为什么一接入 USB 设备，PC 机就能发现它？
答 5. PC 的 USB 口内部，D-和 D+接有 15K 的下拉电阻，未接 USB 设备时为低电平
USB 设备的 USB 口内部，D-或 D+接有 1.5K 的上拉电阻；它一接入 PC，就会把 PC USB 口的 D-或 D+拉高，从硬件的角度通知 PC 有新设备接入

其他概念:
1. USB 是主从结构的
所有的 USB 传输，都是从 USB 主机这方发起；USB 设备没有"主动"通知 USB 主机的能力。
例子：USB 鼠标滑动一下立刻产生数据，但是它没有能力通知 PC 机来读数据，只能被动地等得 PC 机来读。

2. USB 的传输类型:
a. 控制传输：可靠，时间有保证，比如：USB 设备的识别过程
b. 批量传输: 可靠, 时间没有保证, 比如：U 盘
c. 中断传输：可靠，实时，比如：USB 鼠标
d. 实时传输：不可靠，实时，比如：USB 摄像头

3. USB 传输的对象：端点(endpoint)
我们说"读 U 盘"、"写 U 盘"，可以细化为：把数据写到 U 盘的端点 1，从 U 盘的端点 2 里读出数据
除了端点 0 外，每一个端点只支持一个方向的数据传输
端点 0 用于控制传输，既能输出也能输入

4. 每一个端点都有传输类型，传输方向

5. 术语里、程序里说的输入(IN)、输出(OUT) "都是" 基于 USB 主机的立场说的。
比如鼠标的数据是从鼠标传到 PC 机, 对应的端点称为"输入端点"

6. USB 总线驱动程序的作用
a. 识别 USB 设备
b. 查找并安装对应的设备驱动程序
c. 提供 USB 读写函数

USB 驱动程序框架:

app:

USB 设备驱动程序 // 知道数据含义
内核 ---
USB 总线驱动程序 // 1. 识别, 2. 找到匹配的设备驱动, 3. 提供 USB 读写函数 (它不知道数据含义)

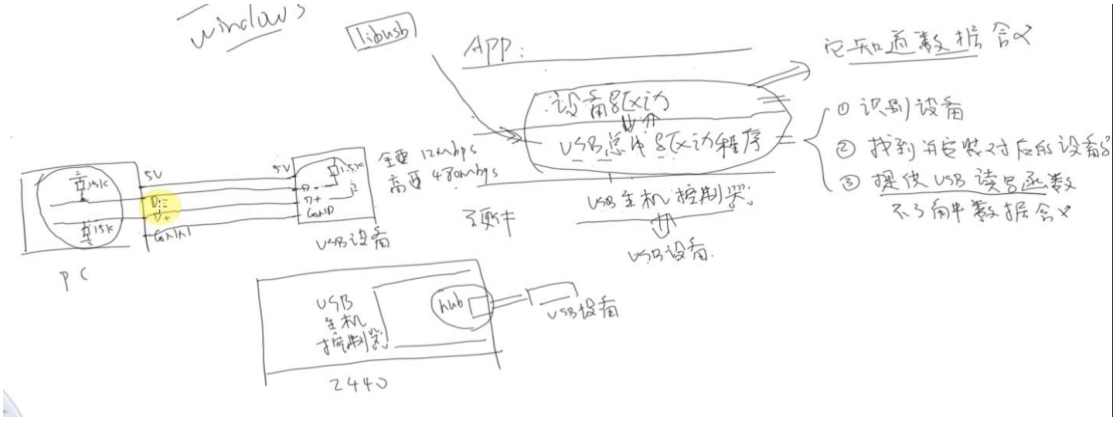
USB 主机控制器
UHCI OHCI EHCI
硬件 ---
USB 设备

UHCI: intel, 低速(1.5Mbps)/全速(12Mbps) usb1.1(低速) 大多为 intel 和 Via 主板上的 usb 芯片。他们都是由 usb1.1
OHCI: microsoft 低速/全速 非 PC 系统上的芯片
EHCI: 高速(480Mbps) 兼容 OHCI 和 UHCI ，遵循 usb2.0 规范
usb1.1(全速 full speed)
usb2.0(高速 high speed)
usb3.0（super speed）
usb 设备优点:

USB 总线驱动程序的作用
1. 识别 USB 设备
1.1 分配地址
1.2 并告诉 USB 设备(set address)
1.3 发出命令获取描述符
描述符的信息可以在 include\linux\usb\Ch9.h 看到

2. 查找并安装对应的设备驱动程序

3. 提供 USB 读写函数



把 USB 设备接到开发板上，看输出信息:
usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: configuration #1 chosen from 1 choice

scsi0 : SCSI emulation for USB Mass Storage devices
scsi 0:0:0:0: Direct-Access HTC Android Phone 0100 PQ: 0 ANSI: 2
sd 0:0:0:0: [sda] Attached SCSI removable disk
拔掉
usb 1-1: USB disconnect, address 2

再接上:
usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
scsi1 : SCSI emulation for USB Mass Storage devices
scsi 1:0:0:0: Direct-Access HTC Android Phone 0100 PQ: 0 ANSI: 2
sd 1:0:0:0: [sda] Attached SCSI removable disk

在内核目录下搜:
grep "USB device using" * -nR
drivers/usb/core/hub.c:2186: "%s %s speed %sUSB device using %s and address %d\n",

hub_irq
kick_khubd
hub_thread
hub_events
hub_port_connect_change

udev = usb_alloc_dev(hdev, hdev->bus, port1);
dev->dev.bus = &usb_bus_type;

choose_address(udev); // 给新设备分配编号(地址)

hub_port_init // usb 1-1: new full speed USB device using s3c2410-ohci and address 3

hub_set_address // 把编号(地址)告诉 USB 设备

usb_get_device_descriptor(udev, 8); // 获取设备描述符(8 个字节是第一次)
retval = usb_get_device_descriptor(udev, USB_DT_DEVICE_SIZE);

usb_new_device(udev)
err = usb_get_configuration(udev); // 把所有的描述符都读出来，并解析
usb_parse_configuration

device_add // 把 device 放入 usb_bus_type 的 dev 链表,
// 从 usb_bus_type 的 driver 链表里取出 usb_driver,
// 把 usb_interface 和 usb_driver 的 id_table 比较
// 如果能匹配，调用 usb_driver 的 probe

<LINUX 内核源代码情景分析>

怎么写 USB 设备驱动程序?
1. 分配/设置 usb_driver 结构体
.id_table
.probe
.disconnect
2. 注册

测试 1th/2th:
1. make menuconfig 去掉原来的 USB 鼠标驱动
-> Device Drivers
-> HID Devices
<> USB Human Interface Device (full HID) support

2. make uImage 并使用新的内核启动

3. insmod usbmouse_as_key.ko
4. 在开发板上接入、拔出 USB 鼠标

测试 3th:
1. insmod usbmouse_as_key.ko
2. ls /dev/event*
3. 接上 USB 鼠标
4. ls /dev/event*
5. 操作鼠标观察数据

测试 4th:
1. insmod usbmouse_as_key.ko
2. ls /dev/event*
3. 接上 USB 鼠标
4. ls /dev/event*
5. cat /dev/tty1 然后按鼠标键
6. hexdump /dev/event0
3、总线平台，Bus 不过是个结构体，虚拟的，为的是把一些稳定的和不稳定的区分开,可以放在任何地方

3、块设备

框架：

```
app:          open,read,write "1.txt"
------------------------------------------    文件的读写
文件系统: vfat, ext2, ext3, yaffs2, jffs2          (把文件的读写转换为扇区的读写)
-----------------ll_rw_block-----------------    扇区的读写
                                1. 把"读写"放入队列
                                2. 调用队列的处理函数(优化/调顺序/合并)
                块设备驱动程序
------------------------------------------
硬件:              硬盘,flash
```

<LINUX 内核源代码情景分析>

分析 ll_rw_block

```
        for (i = 0; i < nr; i++) {
            struct buffer_head *bh = bhs[i];
            submit_bh(rw, bh);
                struct bio *bio; // 使用 bh 来构造 bio (block input/output)
                submit_bio(rw, bio);
                    // 通用的构造请求: 使用 bio 来构造请求(request)
                    generic_make_request(bio);
                        __generic_make_request(bio);
                            request_queue_t *q = bdev_get_queue(bio->bi_bdev); // 找到
队列

                            // 调用队列的"构造请求函数"
                            ret = q->make_request_fn(q, bio);
                                // 默认的函数是__make_request
                                __make_request
                                    // 先尝试合并
                                    elv_merge(q, &req, bio);

                                    // 如果合并不成，使用 bio 构造请求
                                    init_request_from_bio(req, bio);

                                    // 把请求放入队列
                                    add_request(q, req);

                                    // 执行队列
```

```
__generic_unplug_device(q);
        // 调用队列的"处理函数"
        q->request_fn(q);
```

怎么写块设备驱动程序呢?
1. 分配 gendisk: alloc_disk
2. 设置
2.1 分配/设置队列: request_queue_t // 它提供读写能力
    blk_init_queue
2.2 设置 gendisk 其他信息 // 它提供属性: 比如容量
3. 注册: add_disk

4、Nand 驱动:

NAND FLASH 介绍
1. 硬件特性:

【Flash 的硬件实现机制】

Flash 全名叫做 Flash Memory，属于非易失性存储设备(Non-volatile Memory Device)，与此相对应的是易失性存储设备(Volatile Memory Device)。这类设备，除了 Flash，还有其他比较常见的如硬盘，ROM 等，

与此相对的，易失性就是断电了，数据就丢失了，比如大家常用的内存，不论是以前的 SDRAM，DDR SDRAM，还是现在的 DDR2，DDR3 等，都是断电后，数据就没了。

Flash 的内部存储是 MOSFET，里面有个悬浮门(Floating Gate)，是真正存储数据的单元。

金属-氧化层-半导体-场效晶体管，简称金氧半场效晶体管（Metal-Oxide-Semiconductor Field-Effect Transistor, MOSFET）是一种可以广泛使用在模拟电路与数字电路的场效晶体管（field-effect transistor)。MOSFET 依照其“通道”的极性不同，可分为 n-type 与 p-type 的 MOSFET，通常又称为 NMOSFET 与 PMOSFET，其他简称尚包括 NMOS FET、PMOS FET、nMOSFET、pMOSFET 等。

在 Flash 之前，紫外线可擦除(uv-erasable)的 EPROM，就已经采用用 Floating Gate 存储数据这一技术了。

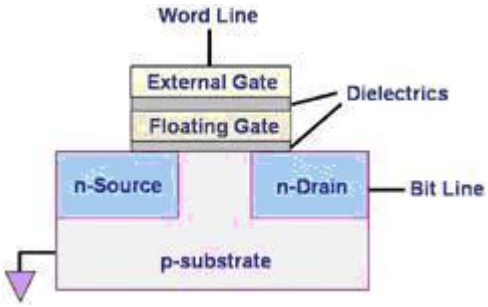


图 1.典型的 Flash 内存单元的物理结构

数据在 Flash 内存单元中是的。

存储电荷的多少，取决于图中的外部门（external gate）所被施加的电压，其控制了是向存储单元中冲入电荷还是使其释放电荷。

数据的表示，以所存储的电荷的电压是否超过一个特定的阈值 Vth 来表示。

【SLC 和 MLC 的实现机制】

Nand Flash 按照内部存储数据单元的电压的不同层次，也就是单个内存单元中，是存储 1 位数据，还是多位数据，可以分为 SLC 和 MLC：

1. SLC，Single Level Cell:

单个存储单元，只存储一位数据，表示成 1 或 0.

就是上面介绍的，对于数据的表示，单个存储单元中内部所存储电荷的电压，和某个特定的阈值电压 Vth，相比，如果大于此 Vth 值，就是表示 1，反之，小于 Vth，就表示 0.

对于 nand Flash 的数据的写入 1，就是控制 External Gate 去充电，使得存储的电荷够多，超过阈值 Vth，就表示 1 了。而对于写入 0，就是将其放电，电荷减少到小于 Vth，就表示 0 了。

关于为何 Nand Flash 不能从 0 变成 1，我的理解是，物理上来说，是可以实现每一位的，从 0 变成 1 的，但是实际上，对于实际的物理实现，出于效率的考虑，如果对于，每一个存储单元都能单独控制，即，0 变成 1 就是，对每一个存储单元单独去充电，所需要的硬件实现就很复杂和昂贵，同时，所进行对块擦除的操作，也就无法实现之前的，一闪而过的速度了，也就失去了 Flash 的众多特性了。

// 也就是放电的思路还是容易些。1->0

2. MLC，Multi Level Cell：

与 SLC 相对应，就是单个存储单元，可以存储多个位，比如 2 位，4 位等。其实现机制，说起来比较简单，就是，通过控制内部电荷的多少，分成多个阈值，通过控制里面的电荷多少，而达到我们所需要的存储成不同的数据。比如，假设输入电压是 Vin＝4V（实际没有这样的电压，此处只是为了举例方便），那么，可以设计出 2 的 2 次方＝4 个阈值，1/4

的 Vin＝1V，2/4 的 Vin＝2V，3/4 的 Vin＝3V，Vin＝4V，分别表示 2 位数据 00，01，10，11，对于写入数据，就是充电，通过控制内部的电荷的多少，对应表示不同的数据。

对于读取，则是通过对应的内部的电流（与 Vth 成反比），然后通过一系列解码电路完成读取，解析出所存储的数据。这些具体的物理实现，都是有足够精确的设备和技术，才能实现精确的数据写入和读出的。

单个存储单元可以存储 2 位数据的，称作 2 的 2 次方＝4 Level Cell，而不是 2 Level Cell；

同理，对于新出的单个存储单元可以存储 4 位数据的，称作 2 的 4 次方＝16 Level Cell。

【关于如何识别 SLC 还是 MLC】

Nand Flash 设计中，有个命令叫做 Read ID，读取 ID，意思是读取芯片的 ID，就像大家的身份证一样，这里读取的 ID 中，是：

读取好几个字节，一般最少是 4 个，新的芯片，支持 5 个甚至更多，从这些字节中，可以解析出很多相关的信息，比如：

此 Nand Flash 内部是几个芯片（chip）所组成的，

每个 chip 包含了几片（Plane），

每一片中的页大小，块大小，等等。

在这些信息中，其中有一个，就是识别此 flash 是 SLC 还是 MLC。下面这个就是最常见的 Nand Flash 的 datasheet 中所规定的，第 3 个字节，3rd byte，所表示的信息，其中就有 SLC/MLC 的识别信息：

	Description	I/O7	I/O6	I/O5 I/O4	I/O3 I/O2	I/O1 I/O0
Internal Chip Number	1 2 4 8					0 0 0 1 1 0 1 1
Cell Type	2 Level Cell 4 Level Cell 8 Level Cell 16 Level Cell				0 0 0 1 1 0 1 1	
Number of Simultaneously Programmed Pages	1 2 4			0 0 0 1 1 0		

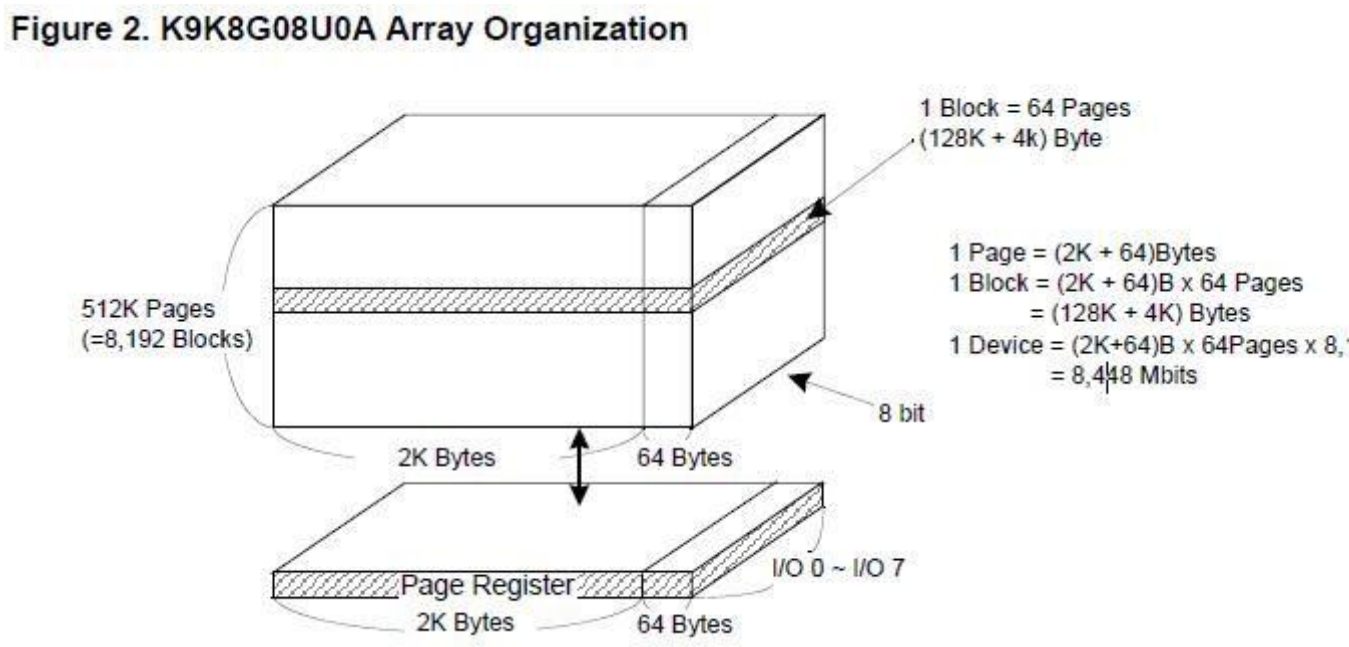
	8			1 1		
Interleave Program Between multiple chips	Not Support Support		0 1			
Cache Program	Not Support Support	0 1				

表 1.Nand Flash 第 3 个 ID 的含义

【Nand Flash 的物理存储单元的阵列组织结构】

Nand flash 的内部组织结构，此处还是用图来解释，比较容易理解：

图 2.Nand Flash 物理存储单元的阵列组织结构



上图是 K9K8G08U0A 的 datasheet 中的描述。

简单解释就是:

1.一个 nand flash 由很多个块（Block）组成，

块的大小一般是

-> 128KB，

-> 256KB，

-> 512KB

此处是 128KB。

2.每个块里面又包含了很多页（page）。每个页的大小，

老的 nand flash，页大小是 256B，512B，

这类的 nand flash 被称作 small block，。地址周期只有 4 个。

对于现在常见的 nand flash 多数是 2KB，

被称作 big block，对应的发读写命令地址，一共 5 个周期(cycle)，

更新的 nand flash 是 4KB，

块，也是 Nand Flash 的擦除操作的基本/最小单位。

3.每一个页，对应还有一块区域，叫做空闲区域（spare area）/冗余区域（redundant area），而 Linux 系统中，一般叫做 OOB（Out Of Band），这个区域，是最初基于 Nand Flash 的硬件特性：数据在读写时候相对容易错误，所以为了保证数据的正确性，必须要有对应的检测和纠错机制，此机制被叫做 EDC(Error Detection Code)/ECC（Error Code Correction,或者 Error Checking and Correcting），所以设计了多余的区域，用于放置数据的校验值。

页，是 Nand Flash 的写入操作的基本/最小的单位。

【Nand Flash 数据存储单元的整体架构】

简单说就是，常见的 nand flash，内部只有一个 chip，每个 chip 只有一个 plane。

而有些复杂的，容量更大的 nand flash，内部有多个 chip，每个 chip 有多个 plane。这类的 nand flash，往往也有更加高级的功能，比如下面要介绍的 Multi Plane Program 和 Interleave Page Program 等。

比如，型号为 K9K8G08U0A 这个芯片（chip），

内部有:

K9F4G08U0A (256MB) : Plane (1Gb), Plane (1Gb)

K9F4G08U0A (256MB) : Plane (1Gb), Plane (1Gb)

K9WAG08U1A ，内部包含了 2 个 K9K8G08U0A

K9NBG08U5A ，内部包含了 4 个 K9K8G08U0A

【Flash 名称的由来】

Flash 的擦除操作是以 block 块为单位的，与此相对应的是其他很多存储设备，是以 bit 位为最小读取/写入的单位，Flash 是一次性地擦除整个块：在发送一个擦除命令后，一次性地将一个 block，常见的块的大小是 128KB/256KB。，全部擦除为 1，也就是里面的内容全部都是 0xFF 了，由于是一下子就擦除了，相对来说，擦除用的时间很短，可以用一闪而过来形容，所以，叫做 Flash Memory。中文有的翻译为（快速）闪存。

【Flash 相对于普通设备的特殊性】

1. 上面提到过的，Flash 最小操作单位，有些特殊。

一般设备，比如硬盘/内存，读取和写入都是以 bit 位为单位，读取一个 bit 的值，将某个值写入对应的地址的位，都是可以按位操作的。

但是 Flash 由于物理特性，使得内部存储的数据，只能从 1 变成 0，这点，可以从前面的内部实现机制了解到，只是方便统一充电，不方便单独的存储单元去放电，所以才说，只能从 1 变成 0，也就是释放电荷。

所以，总结一下 Flash 的特殊性如下：

	普通设备(硬盘/内存等)	Flash
读取/写入的叫法	读取/写入	读取/编程(Program)①
读取/写入的最小单位	Bit/位	Page/页
擦除(Erase)操作的最小单位	Bit/位	Block/块②
擦除操作的含义	将数据删除/全部写入 0	将整个块都擦除成全是 1，也就是里面的数据都是 0xFF③
对于写操作	直接写即可	在写数据之前，要先擦除，然后再写

表 2.Flash 和普通设备相比所具有的特殊性

注：

①之所以将写操作叫做编程，是因为，flash 和之前的 EPROM，EEPROM 继承发展而来，而之前的 EEPROM(Electrically Erasable Programmable Read-Only Memory)，往里面写入数据，就叫做编程 Program，之所以这么称呼，是因为其对数据的写入，是需要用电去擦除/写入的，就叫做编程。

②对于目前常见的页大小是 2K/4K 的 Nand Flash，其块的大小有 128KB/256KB/512KB 等。而对于 Nor Flash，常见的块大小有 64K/32K 等。

③在写数据之前，要先擦除，内部就都变成 0xFF 了，然后才能写入数据，也就是将对应位由 1 变成 0。

【Nand Flash 引脚(Pin)的说明】

Pin Name	Pin Function
$I/O_0 \sim I/O_7$	**DATA INPUTS/OUTPUTS** The I/O pins are used to input command, address and data, and to output data during read operations. The I/O pins float to high-z when the chip is deselected or when the outputs are disabled.
CLE	**COMMAND LATCH ENABLE** The CLE input controls the activating path for commands sent to the command register. When active high, commands are latched into the command register through the I/O ports on the rising edge of the $\overline{WE}$ signal.
ALE	**ADDRESS LATCH ENABLE** The ALE input controls the activating path for address to the internal address registers. Addresses are latched on the rising edge of $\overline{WE}$ with ALE high.
$\overline{CE}$ / $\overline{CE1}$	**CHIP ENABLE** The $\overline{CE}$ / $\overline{CE1}$ input is the device selection control. When the device is in the Busy state, $\overline{CE}$ / $\overline{CE1}$ high is ignored, and the device does not return to standby mode in program or erase operation. Regarding $\overline{CE}$ / $\overline{CE1}$ control during read operation , refer to 'Page Read' section of Device operation.
$\overline{CE2}$	**CHIP ENABLE** The $\overline{CE2}$ input enables the second K9K8G08U0A
$\overline{RE}$	**READ ENABLE** The $\overline{RE}$ input is the serial data-out control, and when active drives the data onto the I/O bus. Data is valid tREA after the falling edge of $\overline{RE}$ which also increments the internal column address counter by one.
$\overline{WE}$	**WRITE ENABLE** The $\overline{WE}$ input controls writes to the I/O port. Commands, address and data are latched on the rising edge of the $\overline{WE}$ pulse.
$\overline{WP}$	**WRITE PROTECT** The $\overline{WP}$ pin provides inadvertent program/erase protection during power transitions. The internal high voltage generator is reset when the $\overline{WP}$ pin is active low.
R/$\overline{B}$ / R/$\overline{B1}$	**READY/BUSY OUTPUT** The R/$\overline{B}$ / R/$\overline{B1}$ output indicates the status of the device operation. When low, it indicates that a program, erase or random read operation is in process and returns to high state upon completion. It is an open drain output and does not float to high-z condition when the chip is deselected or when outputs are disabled.
Vcc	**POWER** Vcc is the power supply for device.
Vss	**GROUND**
N.C	**NO CONNECTION** Lead is not internally connected.

图 3.Nand Flash 引脚功能说明

上图是常见的 Nand Flash 所拥有的引脚（Pin）所对应的功能，简单翻译如下：

1. I/O0 ~ I/O7：用于输入地址/数据/命令，输出数据

2. CLE：Command Latch Enable，命令锁存使能，在输入命令之前，要先在模式寄存器中，设置 CLE 使能

3. ALE：Address Latch Enable，地址锁存使能，在输入地址之前，要先在模式寄存器中，设置 ALE 使能

4. CE#：Chip Enable，芯片使能，在操作 Nand Flash 之前，要先选中此芯片，才能操作

5. RE#：Read Enable，读使能，在读取数据之前，要先使 CE＃有效。

6. WE#：Write Enable，写使能,在写取数据之前，要先使 WE＃有效。

7. WP#：Write Protect，写保护

8. R/B#:Ready/Busy Output,就绪/忙,主要用于在发送完编程/擦除命令后,检测这些操作是否完成,忙,表示编程/擦除操作仍在进行中,就绪表示操作完成.

9. Vcc：Power，电源

10. Vss：Ground，接地

11. N.C：Non-Connection,未定义，未连接。

[小常识]

在数据手册中，你常会看到，对于一个引脚定义，有些字母上面带一横杠的，那是说明此引脚/信号是低电平有效，比如你上面看到的 RE 头上有个横线，就是说明，此 RE 是低电平有效，此外，为了书写方便，在字母后面加“#”，也是表示低电平有效，比如我上面写的 CE#；如果字母头上啥都没有，就是默认的高电平有效，比如上面的 CLE，就是高电平有效。

【为何需要 ALE 和 CLE】

突然想明白了，Nand Flash 中,为何设计这么多的命令,把整个系统搞这么复杂的原因了:

比如命令锁存使能(Command Latch Enable,CLE)和地址锁存使能(Address Latch Enable，ALE)，那是因为，Nand Flash 就 8 个 I/O，而且是复用的，也就是，可以传数据，也可以传地址，也可以传命令，为了区分你当前传入的到底是啥，所以，先要用发一个 CLE（或 ALE）命令，告诉 nand Flash 的控制器一声，我下面要传的是命令（或地址），这样，里面才能根据传入的内容，进行对应的动作。否则,nand flash 内部,怎么知道你传入的是数据,还是地址,还是命令啊,也就无法实现正确的操作了.

【Nand Flash 只有 8 个 I/O 引脚的好处】

1. 减少外围引脚：相对于并口(Parellel)的 Nor Flash 的 48 或 52 个引脚来说，的确是大大减小了引脚数目，这样封装后的芯片体积，就小很多。现在芯片在向体积更小，功能更强，功耗更低发展，减小芯片体积，就是很大的优势。同时，减少芯片接口，也意味着使用此芯片的相关的外围电路会更简化，避免了繁琐的硬件连线。

2. 提高系统的可扩展性，因为没有像其他设备一样用物理大小对应的完全数目的 addr 引脚，在芯片内部换了芯片的大小等的改动，对于用全部的地址 addr 的引脚，那么就会引起这些引脚数目的增加，比如容量扩大一倍，地址空间/寻址空间扩大一倍，所以，地址线数目/addr 引脚数目，就要多加一个，而对于统一用 8 个 I/O 的引脚的 Nand Flash，由于对外提供的都是统一的 8 个引脚，内部的芯片大小的变化或者其他的变化，对于外部使用者(比如编写 nand flash 驱动的人)来说，不需要关心，只是保证新的芯片，还是遵循同样的接口，同样的时序，同样的命令，就可以了。这样就提高了系统的扩展性。

【Nand flash 的一些典型(typical)特性】

1.页擦除时间是 200us，有些慢的有 800us。

2.块擦除时间是 1.5ms.

3.页数据读取到数据寄存器的时间一般是 20us。

4.串行访问（Serial access）读取一个数据的时间是 25ns，而一些旧的 nand flash 是 30ns，甚至是 50ns。

5.输入输出端口是地址和数据以及命令一起 multiplex 复用的。

以前老的 Nand Flash，编程/擦除时间比较短，比如 K9G8G08U0M，才 5K 次，而后来很多 6.nand flash 的编程/擦除的寿命，最多允许的次数，以前的 nand flash 多数是 10K 次，也就是 1 万次，而现在很多新的 nand flash，技术提高了，比如，Micron 的 MT29F1GxxABB，Numonyx 的 NAND04G-B2D/NAND08G-BxC，都可以达到 100K，也就是 10 万次的编程/擦除。和之前常见的 Nor Flash 达到同样的使用寿命了。

7.48 引脚的 TSOP1 封装或 52 引脚的 ULGA 封装

【Nand Flash 中的特殊硬件结构】

由于 nand flash 相对其他常见设备来说，比较特殊，所以，特殊的设备，也有特殊的设计，所以，有些特殊的硬件特性，就有比较解释一下：

1. 页寄存器（Page Register）：由于 Nand Flash 读取和编程操作来说，一般最小单位是页，所以，nand flash 在硬件设计时候，就考虑到这一特性，对于每一片，都有一个对应的区域，专门用于存放，将要写入到物理存储单元中去的或者刚从存储单元中读取出来的，一页的数据，这个数据缓存区，本质上就是一个 buffer，但是只是名字叫法不同，datasheet 里面叫做 Page Register，此处翻译为页寄存器，实际理解为页缓存，更为恰当些。

注意：只有写到了这个页缓存中，只有等你发了对应的编程第二阶段的确认命令 0x10 之后，实际的编程动作才开始，才开始把页缓存中的数据，一点点写到物理存储单元中去。

所以，简单总结一下就是，对于数据的流向，实际是经过了如下步骤：

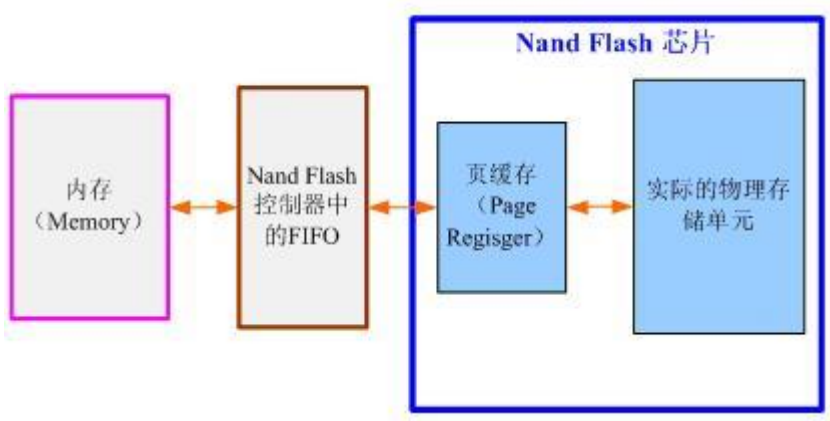


图 4 Nand Flash 读写时的数据流向

【Nand Flash 中的坏块(Bad Block)】

Nand Flash 中，一个块中含有 1 个或多个位是坏的，就成为其为坏块。

坏块的稳定性是无法保证的，也就是说，不能保证你写入的数据是对的，或者写入对了，读出来也不一定对的。而正常的块，肯定是写入读出都是正常的。

坏块有两种：

（1）一种是出厂的时候，也就是，你买到的新的，还没用过的 Nand Flash，就可以包含了坏块。此类出厂时就有的坏块，被称作 factory (masked)bad block 或 initial bad/invalid block，在出厂之前，就会做对应的标记，标为坏块。

具体标记的地方是，对于现在常见的页大小为 2K 的 Nand Flash，是块中第一个页的 oob 起始位置（关于什么是页和 oob，下面会有详细解释）的第 1 个字节（旧的小页面，pagesize 是 512B 甚至 256B 的 nand flash，坏块标记是第 6 个字节），如果不是 0xFF，就说明是坏块。相对应的是，所有正常的块，好的块，里面所有数据都是 0xFF 的。

（2）第二类叫做在使用过程中产生的，由于使用过程时间长了，在擦块除的时候，出错了，说明此块坏了，也要在程序运行过程中，发现，并且标记成坏块的。具体标记的位置，和上面一样。这类块叫做 worn-out bad block。

对于坏块的管理，在 Linux 系统中，叫做坏块管理（BBM，Bad Block Managment），对应的会有一个表去记录好块，坏块的信息，以及坏块是出厂就有的，还是后来使用产生的，这个表叫做 坏块表（BBT，Bad Block Table）。在 Linux 内核 MTD 架构下的 Nand Flash 驱动，和 Uboot 中 Nand Flash 驱动中，在加载完驱动之后，如果你没有加入参数主动要求跳过坏块扫描的话，那么都会去主动扫描坏块，建立必要的 BBT 的，以备后面坏块管理所使用。

而关于好块和坏块，Nand Flash 在出厂的时候，会做出保证：

1.关于好的，可以使用的块的数目达到一定的数目，比如三星的 K9G8G08U0M，整个 flash 一共有 4096 个块，出厂的时候，保证好的块至少大于 3996 个，也就是意思是，你新买到这个型号的 nand flash，最坏的可能，有 3096－3996＝100 个坏块。不过，事实上，现在出厂时的坏块，比较少，绝大多数，都是使用时间长了，在使用过程中出现的。

2.保证第一个块是好的，并且一般相对来说比较耐用。做此保证的主要原因是，很多 Nand Flash 坏块管理方法中，就是将第一个块，用来存储上面提到的 BBT，否则，都是出错几率一样的块，那么也就不太好管理了，连放 BBT 的地方，都不好找了，^_^。

一般来说，不同型号的 Nand Flash 的数据手册中，也会提到，自己的这个 nand flash，最多允许多少个坏块。就比如上面提到的，三星的 K9G8G08U0M，最多有 100 个坏块。

对于坏块的标记，本质上，也只是对应的 flash 上的某些字节的数据是非 0xFF 而已，所以，只要是数据，就是可以读取和写入的。也就意味着，可以写入其他值，也就把这个坏块标记信息破坏了。对于出厂时的坏块，一般是不建议将标记好的信息擦除掉的。

uboot 中有个命令是“nand scrub”就可以将块中所有的内容都擦除了，包括坏块标记，不论是出厂时的，还是后来使用过程中出现而新标记的。一般来说，不建议用这个。不过，我倒是经常用，其实也没啥大碍，呵呵。

最好用“nand erase”只擦除好的块，对于已经标记坏块的块，不擦除。

【nand Flash 中页的访问顺序】

在一个块内，对每一个页进行编程的话，必须是顺序的，而不能是随机的。比如，一个块中有 128 个页，那么你只能先对 page0 编程，再对 page1 编程，。。。。，而不能随机的，比如先对 page3，再 page1，page2.，page0，page4，.。。。

【片选无关(CE don’t-care)技术】

很多 Nand flash 支持一个叫做 CE don’t-care 的技术，字面意思就是，不关心是否片选，

那有人会问了，如果不片选，那还能对其操作吗？答案就是，这个技术，主要用在当时是不需要选中芯片却还可以继续操作的这些情况：在某些应用，比如录音，音频播放等应用，中，外部使用的微秒（us）级的时钟周期，此处假设是比较少的 2us，在进行读取一页或者对页编程时，是对 Nand Flash 操作，这样的串行（Serial Access）访问的周期都是 20/30/50ns，都是纳秒（ns）级的，此处假设是 50ns，当你已经发了对应的读或写的命令之后，接下来只是需要 Nand Flash 内部去自己操作，将数据读取除了或写入进去到内部的数据寄存器中而已，此处，如果可以把片选取消，CE#是低电平有效，取消片选就是拉高电平，这样会在下一个外部命令发送过来之前，即微秒量级的时间里面，即 2us－50ns≈2us，这段时间的取消片选，可以降低很少的系统功耗，但是多次的操作，就可以在很大程度上降低整体的功耗了。

总结起来简单解释就是：由于某些外部应用的频率比较低，而 Nand Flash 内部操作速度比较快，所以具体读写操作的大部分时间里面，都是在等待外部命令的输入，同时却选中芯片，产生了多余的功耗，此“不关心片选”技术，就是在 Nand Flash 的内部的相对快速的操作（读或写）完成之后，就取消片选，以节省系统功耗。待下次外部命令/数据/地址输入来的时候，再选中芯片，即可正常继续操作了。这样，整体上，就可以大大降低系统功耗了。

注:Nand Flash 的片选与否，功耗差别会有很大。如果数据没有记错的话，我之前遇到我们系统里面的 nand flash 的片选，大概有 5 个 mA 的电流输出呢，要知道，整个系统优化之后的待机功耗，也才 10 个 mA 左右的。

【带 EDC 的拷回操作以及 Sector 的定义（Copy-Back Operation with EDC & Sector Definition for EDC）】

Copy-Back 功能，简单的说就是，将一个页的数据，拷贝到另一个页。

如果没有 Copy-Back 功能，那么正常的做法就是，先要将那个页的数据拷贝出来放到内存的数据 buffer 中，读出来之后，再用写命令将这页的数据，写到新的页里面。

而 Copy-Back 功能的好处在于，不需要用到外部的存储空间，不需要读出来放到外部的 buffer 里面，而是可以直接读取数据到内部的页寄存器（page register）然后写到新的页里面去。而且，为了保证数据的正确，要硬件支持 EDC（Error Detection Code）的，否则，在数据的拷贝过程中，可能会出现错误，并且拷贝次数多了，可能会累积更多错误。

而对于错误检测来说，硬件一般支持的是 512 字节数据，对应有 16 字节用来存放校验产生的 ECC 数值，而这 512 字节一般叫做一个扇区。对于 2K＋64 字节大小的页来说，按照

512 字节分，分别叫做 A，B，C，D 区，而后面的 64 字节的 oob 区域，按照 16 字节一个区，分别叫做 E，F，G，H 区，对应存放 A，B，C，D 数据区的 ECC 的值。

总结：

512+16

2K +64 ： A B C D - E F G H 区

Copy-Back 编程的主要作用在于，去掉了数据串行读取出来，再串行写入进去的时间，所以，这部分操作，是比较耗时的，所以此技术可以提高编程效率，提高系统整体性能。

【多片同时编程(Simultaneously Program Multi Plane)】

对于有些新出的 Nand Flash，支持同时对多个片进行编程，比如上面提到的三星的 K9K8G08U0A，内部包含 4 片(Plane)，分别叫做 Plane0，Plane1，Plane2，Plane3。.由于硬件上，对于每一个 Plane，都有对应的大小是 2048+64=2112 字节的页寄存器（Page Register），使得同时支持多个 Plane 编程成为可能。K9K8G08U0A 支持同时对 2 个 Plane 进行编程。不过要注意的是，只能对 Plane0 和 Plane1 或者 Plane2 和 Plane3，同时编程，而不支持 Plane0 和 Plane2 同时编程。

【交错页编程（Interleave Page Program）】

多片同时编程，是针对一个 chip 里面的多个 Plane 来说的，

而此处的交错页编程，是指对多个 chip 而言的。

可以先对一个 chip，假设叫 chip1，里面的一页进行编程，然后此时，chip1 内部就开始将数据一点点写到页里面，就出于忙的状态了，而此时可以利用这个时间，对出于就绪状态的 chip2，也进行页编程，发送对应的命令后，chip2 内部也就开始慢慢的写数据到存储单元里面去了，也出于忙的状态了。此时，再去检查 chip1，如果编程完成了，就可以开始下一页的编程了，然后发完命令后，就让其内部慢慢的编程吧，再去检查 chip2，如果也是编程完了，也就可以进行接下来的其他页的编程了。如此，交互操作 chip1 和 chip2，就可以有效地利用时间，使得整体编程效率提高近 2 倍，大大提高 nand flash 的编程/擦写速度了。

【随机输出页内数据（Random Data Output In a Page）】

在介绍此特性之前，先要说说，与 Random Data Output In a Page 相对应的是，普通的，正常的，sequential data output in a page。

正常情况下，我们读取数据，都是先发读命令，然后等待数据从存储单元到内部的页数据寄存器中后，我们通过不断地将 RE#(Read Enale，低电平有效)置低，然后从我们开始传入的列的起始地址，一点点读出我们要的数据，直到页的末尾，当然有可能还没到页地址的末尾，就不再读了。所谓的顺序（sequential）读取也就是，根据你之前发送的列地址的起始地址开始，每读一个字节的数据出来，内部的数据指针就加 1，移到下个字节的地址，然后你再读下一个字节数据，就可以读出来你要的数据了，直到读取全部的数据出来为止。

而此处的随机（random）读取，就是在你正常的顺序读取的过程中，

先发一个随机读取的开始命令 0x05 命令，

再传入你要将内部那个数据指针定位到具体什么地址，也就是 2 个 cycle 的列地址，

然后再发随机读取结束命令 0xE0，

然后，内部那个数据地址指针，就会移动到你所制定的位置了，

你接下来再读取的数据，就是从那个制定地址开始的数据了。

而 nand flash 数据手册里面也说了，这样的随机读取，你可以多次操作，没限制的。

请注意，上面你所传入的地址，都是列地址，也就是页内地址，也就是说，对于页大小为 2K 的 nand flash 来说，所传入的地址，应该是小于 2048+64＝2112 的。

不过，实际在 nand flash 的使用中，好像这种用法很少的。绝大多数，都是顺序读取数据。

【页编程（写操作）】

Nand flash 的写操作叫做编程 Program，编程，一般情况下，是以页为单位的。

有的 Nand Flash，比如 K9K8G08U0A，支持部分页编程，但是有一些限制：在同一个页内的，连续的部分页的编程，不能超过 4 次。一般情况下，很少使用到部分页编程，都是以页为单位进行编程操作的。

一个操作，用两个命令去实现，看起来是多余，效率不高，但是实际上，有其特殊考虑，

至少对于块擦除来说，开始的命令 0x60 是擦除设置命令(erase setup comman)，然后传入要擦除的块地址，然后再传入擦除确认命令（erase confirm command）0xD0，以开始擦除的操作。

这种，分两步：开始设置，最后确认的命令方式，是为了避免由于外部由于无意的/未预料而产生的噪音，比如，，此时，即使被 nand flash 误认为是擦除操作，但是没有之后的确认操作 0xD0，nand flash 就不会去擦除数据，这样使得数据更安全，不会由于噪音而误操作。

分类: Flash 驱动

【读（read）操作过程详解】

以最简单的 read 操作为例，解释如何理解时序图，以及将时序图中的要求，转化为代码。

解释时序图之前，让我们先要搞清楚，我们要做的事情：那就是，要从 nand flash 的某个页里面，读取我们要的数据。

要实现此功能，会涉及到几部分的知识，至少很容易想到的就是：需要用到哪些命令，怎么发这些命令，怎么计算所需要的地址，怎么读取我们要的数据等等。

下面，就一步步的解释，需要做什么，以及如何去做：

1.需要使用何种命令

首先，是要了解，对于读取数据，要用什么命令。

下面是 datasheet 中的命令集合：

Table 1. Command Sets

Function	1st Cycle	2nd Cycle	Acceptable Command during B
Read	00h	30h	
Read for Copy Back	00h	35h	
Read ID	90h	-	
Reset	FFh	-	O
Page Program	80h	10h	
Two-Plane Page Program(4)	80h---11h	81h---10h	
Copy-Back Program	85h	10h	
Two-Plane Copy-Back Program(4)	85h---11h	81h---10h	
Block Erase	60h	D0h	
Two-Plane Block Erase	60h---60h	D0h	
Random Data Input(1)	85h	-	
Random Data Output(1)	05h	E0h	
Read Status	70h		O
Read EDC Status(2)	7Bh		O
Chip1 Status(3)	F1h		O
Chip2 Status(3)	F2h		O

图 5.Nand Flash K9K8G08U0A 的命令集合

很容易看出，我们要读取数据，要用到 Read 命令，该命令需要 2 个周期，第一个周期发 0x00，第二个周期发 0x30。

2.发送命令前的准备工作以及时序图各个信号的具体含义

知道了用何命令后，再去了解如何发送这些命令。

[小常识]

在开始解释前，多罗嗦一下”使能”这个词，以便有些读者和我以前一样，在听这类虽然对于某些专业人士说是属于最基本的词汇了，但是对于初次接触，或者接触不多的人来说，听多了，容易被搞得一头雾水：使能（Enable），是指使其（某个信号）有效，使其生效的意思，“使其”“能够”怎么怎么样。。。。比如，上面图中的 CLE 线号，是高电平有效，如果此时将其设为高电平，我们就叫做，将 CLE 使能，也就是使其生效的意思。

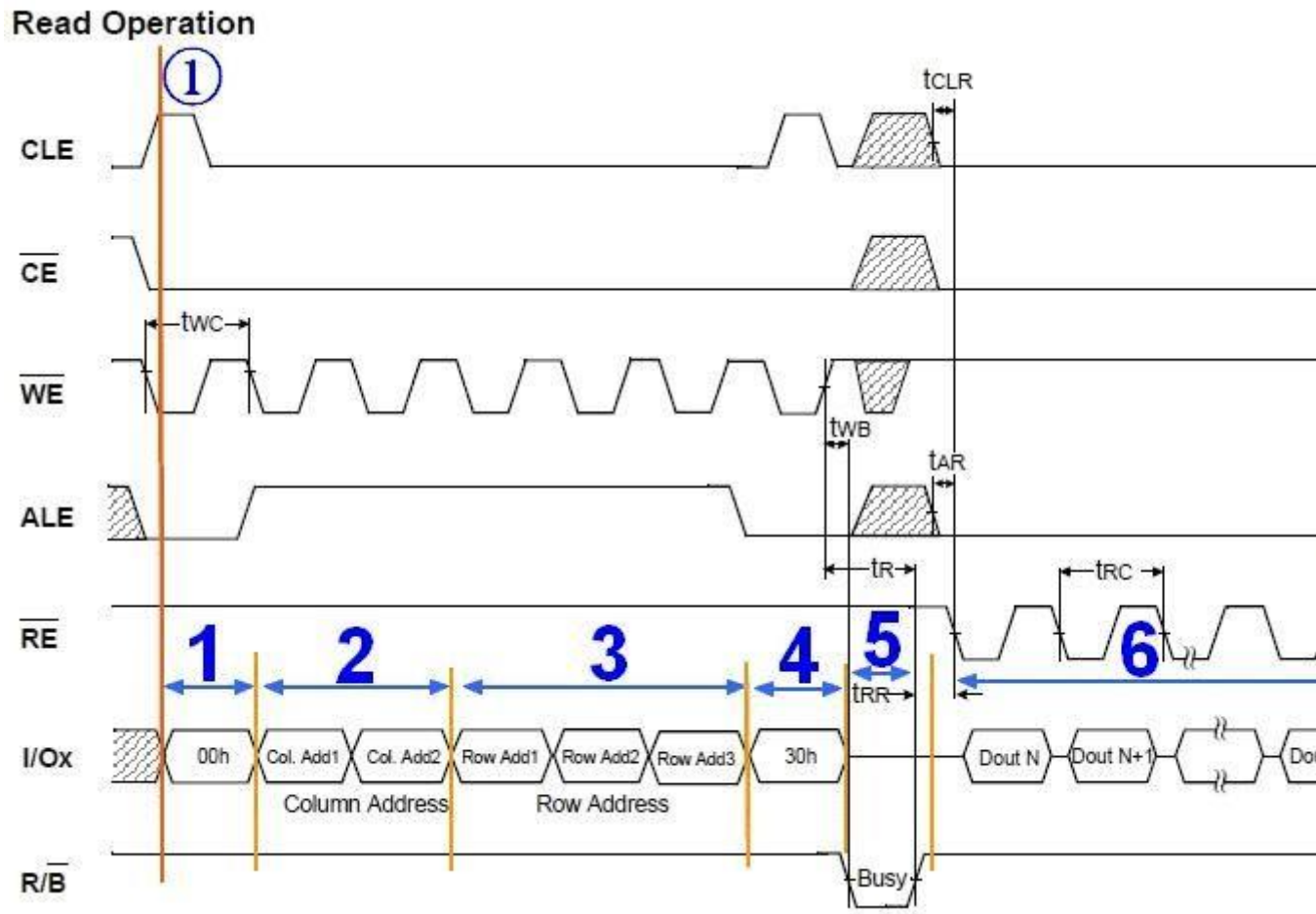


图 6.Nand Flash 数据读取操作的时序图

注：此图来自三星的型号 K9K8G08U0A 的 nand flash 的数据手册(datasheet)。

我们来一起看看，我在图 6 中的特意标注的①边上的黄色竖线。

黄色竖线所处的时刻，是在发送读操作的第一个周期的命令 0x00 之前的那一刻。

让我们看看，在那一刻，其所穿过好几行都对应什么值，以及进一步理解，为何要那个值。

（1）黄色竖线穿过的第一行，是 CLE。还记得前面介绍命令所存使能（CLE）那个引脚吧？CLE，将 CLE 置 1，就说明你将要通过 I/O 复用端口发送进入 Nand Flash 的，是命令，而不是地址或者其他类型的数据。只有这样将 CLE 置 1，使其有效，才能去通知了内部硬件逻辑，你接下来将收到的是命令，内部硬件逻辑，才会将受到的命令，放到命令寄存器中，才能实现后面正确的操作，否则，不去将 CLE 置 1 使其有效，硬件会无所适从，不知道你传入的到底是数据还是命令了。

（2）而第二行，是 CE#，那一刻的值是 0。这个道理很简单，你既然要向 Nand Flash 发命令，那么先要选中它，所以，要保证 CE#为低电平，使其有效，也就是片选有效。

（3）第三行是 WE#，意思是写使能。因为接下来是往 nand Flash 里面写命令，所以，要使得 WE#有效，所以设为低电平。

（4）第四行，是 ALE 是低电平，而 ALE 是高电平有效，此时意思就是使其无效。而对应地，前面介绍的，使 CLE 有效，因为将要数据的是命令，而不是地址。如果在其他某些场合，比如接下来的要输入地址的时候，就要使其有效，而使 CLE 无效了。

（5）第五行，RE#，此时是高电平，无效。可以看到，知道后面低 6 阶段，才变成低电平，才有效，因为那时候，要发生读取命令，去读取数据。

（6）第六行，就是我们重点要介绍的，复用的输入输出 I/O 端口了，此刻，还没有输入数据，接下来，在不同的阶段，会输入或输出不同的数据/地址。

（7）第七行，R/B#,高电平，表示 R（Ready）/就绪，因为到了后面的第 5 阶段，硬件内部，在第四阶段，接受了外界的读取命令后，把该页的数据一点点送到页寄存器中，这段时间，属于系统在忙着干活，属于忙的阶段，所以，R/B#才变成低，表示 Busy 忙的状态的。

介绍了时刻①的各个信号的值，以及为何是这个值之后，相信，后面的各个时刻，对应的不同信号的各个值，大家就会自己慢慢分析了，也就容易理解具体的操作顺序和原理了。

3.如何计算出，我们要传入的地址

在介绍具体读取数据的详细流程之前，还要做一件事，那就是，先要搞懂我们要访问的地址，以及这些地址，如何分解后，一点点传入进去，使得硬件能识别才行。

此处还是以 K9K8G08U0A 为例，此 nand flash，一共有 8192 个块，每个块内有 64 页，每个页是 2K+64 Bytes，假设，我们要访问其中的第 7000 个块中的第 25 页中的 1208 字节处的地址，此时，我们就要先把具体的地址算出来：

物理地址=块大小×块号+页大小×页号+页内地址=7000×128K+64×2K+1208=0x36B204B8,接下来，我们就看看，怎么才能把这个实际的物理地址，转化为 nand Flash 所要求的格式。

在解释地址组成之前，先要来看看其 datasheet 中关于地址周期的介绍：

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	Column Address
2nd Cycle	A8	A9	A10	A11	*L	*L	*L	*L	Column Address
3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19	Row Address
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27	Row Address
5th Cycle	A28	A29	A30	*L	*L	*L	*L	*L	Row Address

NOTE : Column Address : Starting Address of the Register.
* L must be set to "Low".
* The device ignores any additional input of address cycles than required.

图 7 Nand Flash 的地址周期组成

结合图 7 和图 5 中的 2，3 阶段，我们可以看出，此 nand flash 地址周期共有 5 个，2 个列(Column)周期，3 个行（Row）周期。

而对于对应地，我们可以看出，实际上，

列地址 A0~A10，就是页内地址，地址范围是从 0 到 2047，而对出的 A11，理论上可以表示 2048～4095，但是实际上，我们最多也只用到了 2048～2112，用于表示页内的 oob 区域，其大小是 64 字节。

A12～A30，称作页号，页的号码，可以定位到具体是哪一个页。

而其中，A18～A30，表示对应的块号，即属于哪个块。

// 可见：地址的传输顺序是是 页内地址，页号，块号。 从小到大。

简单解释完了地址组成，那么就很容易分析上面例子中的地址了：

0x36B204B8 = 0011 0110 1011 0010 0000 0100 1011 1000，分别分配到 5 个地址周期就是：

1st 周期，A7～A0 ： 1011 1000 = 0x B8

2nd 周期，A11～A8 ： 0000 0100 = 0x04

3rd 周期，A19～A12 ： 0010 0000 = 0x20

4th 周期，A27～A20 ： 0110 1011 = 0x6B

5th 周期，A30～A28 ： 0000 0011 = 0x03

注意，与图 7 中对应的，*L，意思是低电平，由于未用到那些位，datasheet 中强制要求设为 0，所以，才有上面的 2nd 周期中的高 4 位是 0000.其他的 A30 之后的位也是类似原理，都是 0。

因此，接下来要介绍的，我们要访问第 7000 个块中的第 25 页中的 1208 字节处的话，所要传入的地址就是分 5 个周期，分别传入两个列地址的：0xB8，0x04，然后再传 3 个行地址的：0x20，0x6B，0x03，这样硬件才能识别。

4.读操作过程的解释

准备工作终于完了，下面就可以开始解释说明，对于读操作的，上面图中标出来的，1-6 个阶段，具体是什么含义。

（1） 操作准备阶段：此处是读（Read）操作，所以，先发一个图 5 中读命令的第一个阶段的 0x00,表示，让硬件先准备一下，接下来的操作是读。

（2） 发送两个周期的列地址。也就是页内地址，表示，我要从一个页的什么位置开始读取数据。

（3） 接下来再传入三个行地址。对应的也就是页号。

（4） 然后再发一个读操作的第二个周期的命令 0x30。接下来，就是硬件内部自己的事情了。

（5） Nand Flash 内部硬件逻辑，负责去按照你的要求，根据传入的地址，找到哪个块中的哪个页，然后把整个这一页的数据，都一点点搬运到页缓存中去。而在此期间，你所能

做的事，也就只需要去读取状态寄存器，看看对应的位的值，也就是 R/B#那一位，是 1 还是 0，0 的话，就表示，系统是 busy，仍在”忙“(着读取数据)，如果是 1，就说系统活干完了，忙清了，已经把整个页的数据都搬运到页缓存里去了，你可以接下来读取你要的数据了。

对于这里。估计有人会问了，这一个页一共 2048+64 字节，如果我传入的页内地址，就像上面给的 1028 一类的值，只是想读取 1028 到 2011 这部分数据，而不是页开始的 0 地址整个页的数据，那么内部硬件却读取整个页的数据出来，岂不是很浪费吗？答案是，的确很浪费，效率看起来不高，但是实际就是这么做的，而且本身读取整个页的数据，相对时间并不长，而且读出来之后，内部数据指针会定位到你刚才所制定的 1208 的那个位置。

（6） 接下来，就是你“窃取“系统忙了半天之后的劳动成果的时候了，呵呵。通过先去 Nand Flash 的控制器中的数据寄存器中写入你要读取多少个字节(byte)/字(word)，然后就可以去 Nand Flash 的控制器的 FIFO 中，一点点读取你要的数据了。

至此，整个 Nand Flash 的读操作就完成了。

对于其他操作，可以根据我上面的分析，一点点自己去看 datasheet，根据里面的时序图去分析具体的操作过程，然后对照代码，会更加清楚具体是如何实现的。

【Flash 的类型】

Flash 的类型主要分两种，nand flash 和 nor flash。

除了网上最流行的这个解释之外：

NAND 和 NOR 的比较

再多说几句：

1.nor 的成本相对高，具体读写数据时候，不容易出错。总体上，比较适合应用于存储少量的代码。

2.Nand flash 相对成本低。使用中数据读写容易出错，所以一般都需要有对应的软件或者硬件的数据校验算法，统称为 ECC。由于相对来说，容量大，价格便宜，因此适合用来存储大量的数据。其在嵌入式系统中的作用，相当于 PC 上的硬盘，用于存储大量数据。

所以，一个常见的应用组合就是，用小容量的 Nor Flash 存储启动代码，比如 uboot，系统启动后,初始化对应的硬件，包括 SDRAM 等，然后将 Nand Flash 上的 Linux 内核读取到内存中，做好该做的事情后，就跳转到 SDRAM 中去执行内核了，然后内核解压（如果是压缩内核的话，否则就直接运行了）后，开始运行，在 Linux 内核启动最后，去 Nand Flash 上，挂载根文件，比如 jffs2，yaffs2 等，挂载完成，运行初始化脚本，启动 consle 交互，才运行你通过 console 和内核交互。至此完成整个系统启动过程。

而 Nor Flash 存放的是 Uboot，Nand Flash 存放的是 Linux 的内核镜像和根文件系统，以及余下的空间分成一个数据区。

Nor flash，有类似于 dram 之类的地址总线，因此可以直接和 CPU 相连，CPU 可以直接通过地址总线对 nor flash 进行访问，而 nand flash 没有这类的总线，只有 IO 接口，只能通过

IO 接口发送命令和地址，对 nand flash 内部数据进行访问。相比之下，nor flash 就像是并行访问，nand flash 就是串行访问，所以相对来说，前者的速度更快些。

但是由于物理制程/制造方面的原因，导致 nor 和 nand 在一些具体操作方面的特性不同：

	NOR	NAND	（备注）
接口	总线	I/O接口	这个两者最大的区别
单个cell大小	大	小	
单个Cell成本	高	低	
读耗时	快	慢	
单字节的编程时间	快	慢	
多字节的编程时间	慢	快	
擦除时间	慢	快	
功耗	高	低，但是需要额外的RAM	
是否可以执行代码	是	不行，但是一些新的芯片，可以在第一页之外执行一些小的loader（1）	即是否允许，芯片内执行(XIP, eXecute In Place) (2)
位反转(Bit twiddling/bit flip)	几乎无限制	1-4次，也称作“部分页编程限制”	也就是数据错误，0->1或1->0
在芯片出厂时候是否允许坏块	不允许	允许	

表 3 Nand Flash 和 Nor Flash 的区别

1. 理论上是可以的，而且也是有人验证过可以的，只不过由于 nand flash 的物理特性，不能完全保证所读取的数据/代码是正确的，实际上，很少这么用而已。因为，如果真是要用到 nand flash 做 XIP，那么除了读出速度慢之外，还要保证有数据的校验，以保证读出来的，将要执行的代码/数据，是正确的。否则，系统很容易就跑飞了。。。

2. 芯片内执行(XIP, eXecute In Place):

http://hi.baidu.com/serial_story/blog/item/adb20a2a3f8ffe3c5243c1df.html

【Nand Flash 的种类】

具体再分，又可以分为

1)Bare NAND chips：裸片，单独的 nand 芯片

2)SmartMediaCards： =裸片+一层薄塑料，常用于数码相机和 MP3 播放器中。之所以称 smart，是由于其软件 smart，而不是硬件本身有啥 smart 之处。^_^

3)DiskOnChip：裸片+glue logic，glue logic=硬件 ECC 产生器+用于静态的 nand 芯片控制的寄存器+直接访问一小片地址窗口，那块地址中包含了引导代码的 stub 桩，其可以从 nand flash 中拷贝真正的引导代码。

【spare area/oob】

Nand 由于最初硬件设计时候考虑到，额外的错误校验等需要空间，专门对应每个页，额外设计了叫做 spare area 空区域，在其他地方，比如 jffs2 文件系统中，也叫做 oob（out of band）数据。

其具体用途，总结起来有：

1. 标记是否是坏快

2. 存储 ECC 数据

3. 存储一些和文件系统相关的数据，如 jffs2 就会用到这些空间存储一些特定信息，yaffs2 文件系统，会在 oob 中，存放很多和自己文件系统相关的信息。

【内存技术设备，MTD（Memory Technology Device）】

MTD，是 Linux 的存储设备中的一个子系统。其设计此系统的目的是，对于内存类的设备，提供一个抽象层，一个接口，使得对于硬件驱动设计者来说，可以尽量少的去关心存储格式，比如 FTL，FFS2 等，而只需要去提供最简单的底层硬件设备的读/写/擦除函数就可以了。而对于数据对于上层使用者来说是如何表示的，硬件驱动设计者可以不关心，而 MTD 存储设备子系统都帮你做好了。

对于 MTD 字系统的好处，简单解释就是，他帮助你实现了，很多对于以前或者其他系统来说，本来也是你驱动设计者要去实现的很多功能。换句话说，有了 MTD，使得你设计 Nand Flash 的驱动，所要做的事情，要少很多很多，因为大部分工作，都由 MTD 帮你做好了。

当然，这个好处的一个“副作用”就是，使得我们不了解的人去理解整个 Linux 驱动架构，以及 MTD，变得更加复杂。但是，总的说，觉得是利远远大于弊，否则，就不仅需要你理解，而且还是做更多的工作，实现更多的功能了。

此外，还有一个重要的原因，那就是，前面提到的 nand flash 和普通硬盘等设备的特殊性：

有限的通过出复用来实现输入输出命令和地址/数据等的 IO 接口，最小单位是页而不是常见的 bit，写前需擦除等，导致了这类设备，不能像平常对待硬盘等操作一样去操作，只能采取一些特殊方法，这就诞生了 MTD 设备的统一抽象层。

MTD，将 nand flash，nor flash 和其他类型的 flash 等设备，统一抽象成 MTD 设备来管理，根据这些设备的特点，上层实现了常见的操作函数封装，底层具体的内部实现，就需要驱动设计者自己来实现了。具体的内部硬件设备的读/写/擦除函数，那就是你必须实现的了。

HARD drives	MTD device
连续的扇区	连续的可擦除块
扇区都很小(512B,1024B)	可擦除块比较大 (32KB,128KB)
主要通过两个操作对其维护操作：读扇区，写扇区	主要通过三个操作对其维护操作：从擦除块中读，写入擦除块，**擦写可擦除块**
坏快被重新映射，并且被硬件隐藏起来了（至少是在如今常见的LBA硬盘设备中是如此）	坏的可擦除块没有被隐藏，软件中要处理对应的坏块问题。
HDD扇区没有擦写寿命超出的问题。	可擦除块是有擦除次数限制的，大概是10^4-10^5次.

表 4.MTD 设备和硬盘设备之间的区别

==========

多说一句，关于 MTD 更多的内容，感兴趣的，去附录中的 MTD 的主页去看。

关于 mtd 设备驱动，感兴趣的可以去参考

MTD 原始设备与 FLASH 硬件驱动的对话

MTD 原始设备与 FLASH 硬件驱动的对话-续

那里，算是比较详细地介绍了整个流程，方便大家理解整个 mtd 框架和 nand flash 驱动。

【Nand flash 驱动工作原理】

在介绍具体如何写 Nand Flash 驱动之前，我们先要了解，大概的，整个系统，和 Nand Flash 相关的部分的驱动工作流程，这样，对于后面的驱动实现，才能更加清楚机制，才更容易实现，否则就是，即使写完了代码，也还是没搞懂系统是如何工作的了。

让我们以最常见的，Linux 内核中已经有的三星的 Nand Flash 驱动，来解释 Nand Flash 驱动具体流程和原理。

此处是参考 2.6.29 版本的 Linux 源码中的\drivers\mtd\nand\s3c2410.c，以 2410 为例。

1. 在 nand flash 驱动加载后，第一步，调用对应的 init 函数 ---- s3c2410_nand_init: 去将 nand flash 驱动注册到 Linux 驱动框架中。

2. 驱动本身真正的开始，是从 probe 函数： s3c2410_nand_probe->s3c24xx_nand_probe, 在 probe 过程中：
clk_enable //打开 nand flash 控制器的 clock 时钟，
request_mem_region //去申请驱动所需要的一些内存等相关资源。
s3c2410_nand_inithw //去初始化硬件相关的部分，主要是关于时钟频率的计算，以及启用 nand flash 控制器，使得硬件初始化好了，后面才能正常工作。

3. 需要多解释一下的，是这部分代码：

```
for (setno = 0; setno < nr_sets; setno++, nmtd++) {
pr_debug("initialising set %d (%p, info %p)\n", setno, nmtd, info);
```

/*调用 init chip 去挂载你的 nand 驱动的底层函数到"nand flash 的结构体"中，以及设置对应的"ecc mode"，挂载 ecc 相关的函数 */
s3c2410_nand_init_chip(info, nmtd, sets);

/* scan_ident，扫描 nand 设备，设置 nand flash 的默认函数，获得物理设备的具体型号以及对应各个特性参数，这部分算出来的一些值，对于 nand flash 来说，是最主要的参数，比如 nand flash 的芯片的大小，块大小，页大小等。 */
nmtd->scan_res = nand_scan_ident(&nmtd->mtd, (sets) ? sets->nr_chips : 1);

if (nmtd->scan_res == 0) {
s3c2410_nand_update_chip(info, nmtd);

/*扫描的后一阶段，经过前面的 scan_ident，我们已经获得对应 nand flash 的硬件的各个参数，
*然后就可以在 scan tail 中，根据这些参数，去设置其他一些重要参数，尤其是 ecc 的 layout，即 ecc 是如何在 oob 中摆放的，
*最后，再去进行一些初始化操作，主要是根据你的驱动，如果没有实现一些函数的话，那么就用系统默认的。 */
nand_scan_tail(&nmtd->mtd);

/*add partion，根据你的 nand flash 的分区设置，去分区 */
s3c2410_nand_add_partition(info, nmtd, sets);
}
if (sets != NULL)
sets++;
}

4. 等所有的参数都计算好了，函数都挂载完毕，系统就可以正常工作了。

上层访问你的 nand falsh 中的数据的时候，通过 MTD 层，一层层调用，最后调用到你所实现的那些底层访问硬件数据/缓存的函数中。

【Linux 下 nand flash 驱动编写步骤简介】

关于上面提到的，在 nand_scan_tail 的时候，系统会根据你的驱动，如果没有实现一些函数的话，那么就用系统默认的。如果实现了自己的函数，就用你的。
"那么到底我要实现哪些函数呢，而又有哪些是可以不实现，用系统默认的就可以了呢。"此问题的，就是我们下面要介绍的，也就是，你要实现的，你的驱动最少要做哪些工作，才能使整个 nand flash 工作起来。

1. 对于驱动框架部分

其实，要了解，关于驱动框架部分，你所要做的事情的话，只要看看三星的整个 nand flash 驱动中的这个结构体，就差不多了：

```
static struct platform_driver s3c2410_nand_driver = {
.probe = s3c2410_nand_probe,
.remove = s3c2410_nand_remove,
.suspend = s3c24xx_nand_suspend,
.resume = s3c24xx_nand_resume,
.driver = {
.name = "s3c2410-nand",
.owner = THIS_MODULE,
},
};
```

对于上面这个结构体，没多少要解释的。从名字，就能看出来：
（1）probe 就是系统“探测”，就是前面解释的整个过程，这个过程中的多数步骤，都是和你自己的 nand flash 相关的，尤其是那些硬件初始化部分，是你必须要自己实现的。
（2）remove，就是和 probe 对应的，“反初始化”相关的动作。主要是释放系统相关资源和关闭硬件的时钟等常见操作了。
（3）suspend 和 resume，对于很多没用到电源管理的情况下，至少对于我们刚开始写基本的驱动的时候，可以不用关心，放个空函数即可。

2. 对于 nand flash 底层操作实现部分

而对于底层硬件操作的有些函数，总体上说，都可以在上面提到的 s3c2410_nand_init_chip 中找到：

```
static void s3c2410_nand_init_chip(struct s3c2410_nand_info *info,
struct s3c2410_nand_mtd *nmtd, //主要是完善该结构体
struct s3c2410_nand_set *set)
{

struct nand_chip *chip = &nmtd->chip;
void __iomem *regs = info->regs;

chip->write_buf = s3c2410_nand_write_buf;
chip->read_buf = s3c2410_nand_read_buf;
chip->select_chip = s3c2410_nand_select_chip;
chip->chip_delay = 50;
chip->priv = nmtd;
chip->options = 0;
chip->controller = &info->controller;

switch (info->cpu_type) {
case TYPE_S3C2410:
```

```
/* nand flash 控制器中，一般都有对应的数据寄存器，用于给你往里面写数据，表示将要读取或写入多少个字节(byte,u8)/字(word,u32) ，所以，此处，你要给出地址，以便后面的操作所使用 */

chip->IO_ADDR_W = regs + S3C2410_NFDATA;
info->sel_reg = regs + S3C2410_NFCONF;
info->sel_bit = S3C2410_NFCONF_nFCE;
chip->cmd_ctrl = s3c2410_nand_hwcontrol;
chip->dev_ready = s3c2410_nand_devready;

break;

。。。。。。

}

chip->IO_ADDR_R = chip->IO_ADDR_W;

nmtd->info = info;
nmtd->mtd.priv = chip;
nmtd->mtd.owner = THIS_MODULE;
nmtd->set = set;

if (hardware_ecc) {
chip->ecc.calculate = s3c2410_nand_calculate_ecc;
chip->ecc.correct = s3c2410_nand_correct_data;

/* 此处，多数情况下，你所用的 Nand Flash 的控制器，都是支持硬件 ECC 的，所以，此处设置硬件 ECC(HW_ECC) ，也是充分利用硬件的特性，
* 而如果此处不用硬件去做 ECC 话，那么下面也会去设置成 NAND_ECC_SOFT，系统会用默认的软件去做 ECC 校验，相比之下，比硬件 ECC 的效率就低很多，而你的 nand flash 的读写，也会相应地要慢不少 */

chip->ecc.mode = NAND_ECC_HW; //设置成了硬件方式校验 ecc

switch (info->cpu_type) {

case TYPE_S3C2410:

chip->ecc.hwctl = s3c2410_nand_enable_hwecc;
chip->ecc.calculate = s3c2410_nand_calculate_ecc;

break;

。。。。。
}

} else {
chip->ecc.mode = NAND_ECC_SOFT; //也就是说，怎么搞也得校验了
}
```

```
if (set->ecc_layout != NULL)
chip->ecc.layout = set->ecc_layout;

if (set->disable_ecc)
chip->ecc.mode = NAND_ECC_NONE;
}
```

而我们要实现的底层函数，也就是上面蓝色标出来的一些函数而已：

（1）s3c2410_nand_write_buf 和 s3c2410_nand_read_buf：这是两个最基本的操作函数，其功能，就是往你的 nand flash 的控制器中的 FIFO 读写数据。一般情况下，是 MTD 上层的操作，比如要读取一页的数据，那么在发送完相关的读命令和等待时间之后，就会调用到你底层的 read_buf，去 nand Flash 的 FIFO 中，一点点把我们要的数据，读取出来，放到我们制定的内存的缓存中去。写操作也是类似，将我们内存中的数据，写到 Nand Flash 的 FIFO 中去。

（2）s3c2410_nand_select_chip： 实现 Nand Flash 的片选。

（3）s3c2410_nand_hwcontrol： 给底层发送命令或地址，或者设置具体操作的模式，都是通过此函数。

（4）s3c2410_nand_devready： Nand Flash 的一些操作，比如读一页数据，写入（编程）一页数据，擦除一个块，都是需要一定时间的，在命令发送完成后，就是硬件开始忙着工作的时候了，而硬件什么时候完成这些操作，什么时候不忙了，变就绪了，就是通过这个函数去检查状态的。一般具体实现都是去读硬件的一个状态寄存器，其中某一位是否是 1，对应着是出于“就绪/不忙”还是“忙”的状态。这个寄存器，也就是我们前面分析时序图中的 R/B#。

（5）s3c2410_nand_calculate_ecc：如果是上面提到的硬件 ECC 的话，就不用我们用软件去实现校验算法了，而是直接去读取硬件产生的 ECC 数值就可以了。

（6）s3c2410_nand_correct_data： 当实际操作过程中，读取出来的数据所对应的硬件或软件计算出来的 ECC，和从 oob 中读出来的 ECC 不一样的时候，就是说明数据有误了，就需要调用此函数去纠正错误。对于现在 SLC 常见的 ECC 算法来说，可以发现 2 位，纠正 1 位。如果错误大于 1 位，那么就无法纠正回来了。一般情况下，出错超过 1 位的，好像几率不大。至少我看到的不是很大。更复杂的情况和更加注重数据安全的情况下，一般是需要另外实现更高效和检错和纠错能力更强的 ECC 算法的。

（7）s3c2410_nand_enable_hwecc： 在硬件支持的前提下，前面设置了硬件 ECC 的话，要实现这个函数，用于每次在读写操作前，通过设置对应的硬件寄存器的某些位，使得启用硬件 ECC，这样在读写操作完成后，就可以去读取硬件校验产生出来的 ECC 数值了。

当然，除了这些你必须实现的函数之外，在你更加熟悉整个框架之后，你可以根据你自己的 nand flash 的特点，去实现其他一些原先用系统默认但是效率不高的函数，而用自己的更高效率的函数替代他们，以提升你的 nand flash 的整体性能和效率。

NAND FLASH 是一个存储芯片
那么：这样的操作很合理"读地址 A 的数据，把数据 B 写到地址 A"

问 1. 原理图上 NAND FLASH 和 S3C2440 之间只有数据线，
怎么传输地址?
答 1．在 DATA0～DATA7 上既传输数据，又传输地址
当 ALE 为高电平时传输的是地址，

问 2. 从 NAND FLASH 芯片手册可知，要操作 NAND FLASH 需要先发出命令
怎么传入命令?
答 2．在 DATA0～DATA7 上既传输数据，又传输地址，也传输命令
当 ALE 为高电平时传输的是地址，
当 CLE 为高电平时传输的是命令
当 ALE 和 CLE 都为低电平时传输的是数据

问 3. 数据线既接到 NAND FLASH，也接到 NOR FLASH，还接到 SDRAM、DM9000 等等
那么怎么避免干扰?
答 3. 这些设备，要访问之必须"选中"，
没有选中的芯片不会工作，相当于没接一样

问 4. 假设烧写 NAND FLASH，把命令、地址、数据发给它之后，
NAND FLASH 肯定不可能瞬间完成烧写的，
怎么判断烧写完成?
答 4. 通过状态引脚 RnB 来判断：它为高电平表示就绪，它为低电平表示正忙

问 5. 怎么操作 NAND FLASH 呢?
答 5. 根据 NAND FLASH 的芯片手册，一般的过程是：
发出命令
发出地址
发出数据/读数据

	NAND FLASH	S3C2440
发命令	选中芯片	
	CLE 设为高电平	NFCMMD=命令值
	在 DATA0~DATA7 上输出命令值	
	发出一个写脉冲	
发地址	选中芯片	NFADDR=地址值
	ALE 设为高电平	
	在 DATA0~DATA7 上输出地址值	
	发出一个写脉冲	
发数据	选中芯片	NFDATA=数据值
	ALE,CLE 设为低电平	
	在 DATA0~DATA7 上输出数据值	
	发出一个写脉冲	

读数据　　选中芯片　　val=NFDATA
　　发出读脉冲
　　读 DATA0~DATA7 的数据

用 UBOOT 来体验 NAND FLASH 的操作：

1. 读 ID

	S3C2440	u-boot
选中	NFCONT 的 bit1 设为 0	md.l 0x4E000004 1; mw.l 0x4E000004 1
发出命令 0x90	NFCMMD=0x90	mw.b 0x4E000008 0x90
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
读数据得到 0xEC	val=NFDATA	md.b 0x4E000010 1
读数据得到 device code 0xda	val=NFDATA	md.b 0x4E000010 1
退出读 ID 的状态	NFCMMD=0xff	mw.b 0x4E000008 0xff

2. 读内容: 读 0 地址的数据
使用 UBOOT 命令:
nand dump 0
Page 00000000 dump:
　　17 00 00 ea 14 f0 9f e5　14 f0 9f e5 14 f0 9f e5

	S3C2440	u-boot
选中	NFCONT 的 bit1 设为 0	md.l 0x4E000004 1; mw.l 0x4E000004 1
发出命令 0x00	NFCMMD=0x00	mw.b 0x4E000008 0x00
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出命令 0x30	NFCMMD=0x30	mw.b 0x4E000008 0x30
读数据得到 0x17	val=NFDATA	md.b 0x4E000010 1
读数据得到 0x00	val=NFDATA	md.b 0x4E000010 1
读数据得到 0x00	val=NFDATA	md.b 0x4E000010 1
读数据得到 0xea	val=NFDATA	md.b 0x4E000010 1
退出读状态	NFCMMD=0xff	mw.b 0x4E000008 0xff

NAND FLASH 驱动程序层次

看内核启动信息

S3C24XX NAND Driver, (c) 2004 Simtec Electronics
s3c2440-nand s3c2440-nand: Tacls=3, 30ns Twrph0=7 70ns, Twrph1=3 30ns
NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (Samsung NAND 256MiB 3,3V 8-bit)
Scanning device for bad blocks
Bad eraseblock 256 at 0x02000000
Bad eraseblock 257 at 0x02020000
Bad eraseblock 319 at 0x027e0000
Bad eraseblock 606 at 0x04bc0000
Bad eraseblock 608 at 0x04c00000
Creating 4 MTD partitions on "NAND 256MiB 3,3V 8-bit":
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"

搜"S3C24XX NAND Driver"
S3c2410.c (drivers\mtd\nand)

s3c2410_nand_inithw
s3c2410_nand_init_chip
nand_scan // drivers/mtd/nand/nand_base.c 根据 nand_chip 的底层操作函数识别 NAND FLASH，构造 mtd_info
    nand_scan_ident
        nand_set_defaults
            if (!chip->select_chip)
                chip->select_chip = nand_select_chip; // 默认值不适用

            if (chip->cmdfunc == NULL)
                chip->cmdfunc = nand_command;
                                chip->cmd_ctrl(mtd, command, ctrl);
            if (!chip->read_byte)
                chip->read_byte = nand_read_byte;
                                readb(chip->IO_ADDR_R);
            if (chip->waitfunc == NULL)
                chip->waitfunc = nand_wait;
                                chip->dev_ready

        nand_get_flash_type
            chip->select_chip(mtd, 0);
            chip->cmdfunc(mtd, NAND_CMD_READID, 0x00, -1);
            *maf_id = chip->read_byte(mtd);
            dev_id = chip->read_byte(mtd);
    nand_scan_tail

```
                mtd->erase = nand_erase;
                mtd->read = nand_read;
                mtd->write = nand_write;
s3c2410_nand_add_partition
    add_mtd_partitions
        add_mtd_device
            list_for_each(this, &mtd_notifiers) { //  问. mtd_notifiers 在哪设置
                                                    //          答          .
drivers/mtd/mtdchar.c,mtd_blkdev.c 调用 register_mtd_user
                struct mtd_notifier *not = list_entry(this, struct mtd_notifier, list);
                not->add(mtd);
                // mtd_notify_add   和 blktrans_notify_add
                先看字符设备的 mtd_notify_add
                        class_device_create
                        class_device_create
                再看块设备的 blktrans_notify_add
                    list_for_each(this, &blktrans_majors) { //  问. blktrans_majors 在哪设置
                                                            //          答          .
drivers\mtd\mdblock.c 或 mtdblock_ro.c     register_mtd_blktrans
                        struct mtd_blktrans_ops *tr = list_entry(this, struct mtd_blktrans_ops,
list);
                        tr->add_mtd(tr, mtd);
                                mtdblock_add_mtd (drivers\mtd\mdblock.c)
                                    add_mtd_blktrans_dev
                                        alloc_disk
                                        gd->queue     =     tr->blkcore_priv->rq;     //
tr->blkcore_priv->rq = blk_init_queue(mtd_blktrans_request, &tr->blkcore_priv->queue_lock);
                                        add_disk
```

测试 4th:

1. make menuconfig 去掉内核自带的 NAND FLASH 驱动

-> Device Drivers

  -> Memory Technology Device (MTD) support

    -> NAND Device Support

  < > NAND Flash support for S3C2410/S3C2440 SoC

2. make uImage

   使用新内核启动, 并且使用 NFS 作为根文件系统

3. insmod s3c_nand.ko

4. 格式化 (参考下面编译工具)

   flash_eraseall /dev/mtd3 // yaffs

5. 挂接

mount -t yaffs /dev/mtdblock3 /mnt
6. 在/mnt 目录下建文件

编译工具:
1. tar xjf mtd-utils-05.07.23.tar.bz2
2. cd mtd-utils-05.07.23/util
修改 Makefile:
#CROSS=arm-linux-
改为
CROSS=arm-linux-
3. make
4. cp flash_erase flash_eraseall /work/nfs_root/first_fs/bin/

5、NOR flash

使用 UBOOT 体验 NOR FLASH 的操作(开发板设为 NOR 启动，进入 UBOOT)
先使用 OpenJTAG 烧写 UBOOT 到 NOR FLASH

1. 读数据
md.b 0

2. 读 ID
NOR 手册上:
往地址 555H 写 AAH
往地址 2AAH 写 55H
往地址 555H 写 90H
读 0 地址得到厂家 ID: C2H
读 1 地址得到设备 ID: 22DAH 或 225BH
退出读 ID 状态: 给任意地址写 F0H

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址
UBOOT 怎么操作?

往地址 AAAH 写 AAH	mw.w aaa aa
往地址 554 写 55H	mw.w 554 55
往地址 AAAH 写 90H	mw.w aaa 90
读 0 地址得到厂家 ID: C2H	md.w 0 1
读 2 地址得到设备 ID: 22DAH 或 225BH	md.w 2 1
退出读 ID 状态:	mw.w 0 f0

3. NOR 有两种规范, jedec, cfi(common flash interface)
读取 CFI 信息

NOR 手册:
进入 CFI 模式　　往 55H 写入 98H
读数据:　　读 10H 得到 0051
读 11H 得到 0052
读 12H 得到 0059
读 27H 得到容量

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址
UBOOT 怎么操作?
进入 CFI 模式　　往 AAH 写入 98H　　mw.w aa 98
读数据:　　读 20H 得到 0051　　md.w 20 1
读 22H 得到 0052　　md.w 22 1
读 24H 得到 0059　　md.w 24 1
读 4EH 得到容量　　md.w 4e 1
退出 CFI 模式　　mw.w 0 f0

4. 写数据: 在地址 0x100000 写入 0x1234
md.w 100000 1　　// 得到 ffff
mw.w 100000 1234
md.w 100000 1　　// 还是 ffff

NOR 手册:
往地址 555H 写 AAH
往地址 2AAH 写 55H
往地址 555H 写 A0H
往地址 PA 写 PD

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址
UBOOT 怎么操作?
往地址 AAAH 写 AAH　　mw.w aaa aa
往地址 554H 写 55H　　mw.w 554 55
往地址 AAAH 写 A0H　　mw.w aaa a0
往地址 0x100000 写 1234h　　mw.w 100000 1234

NOR FLASH 驱动程序框架

测试 1:通过配置内核支持 NOR FLASH

1. make menuconfig
-> Device Drivers
  -> Memory Technology Device (MTD) support
    -> Mapping drivers for chip access
    <M> CFI Flash device in physical memory map
    (0x0) Physical start address of flash mapping // 物理基地址
    (0x1000000) Physical length of flash mapping // 长度
    (2) Bank width in octets (NEW) // 位宽

2. make modules
  cp drivers/mtd/maps/physmap.ko /work/nfs_root/first_fs
3. 启动开发板
  ls /dev/mtd*
  insmod physmap.ko
  ls /dev/mtd*
  cat /proc/mtd

测试 2: 使用自己写的驱动程序:

1. ls /dev/mtd*
2. insmod s3c_nor.ko
3. ls /dev/mtd*
4. 格式化: flash_eraseall -j /dev/mtd1
5. mount -t jffs2 /dev/mtdblock1 /mnt
  在/mnt 目录下操作文件

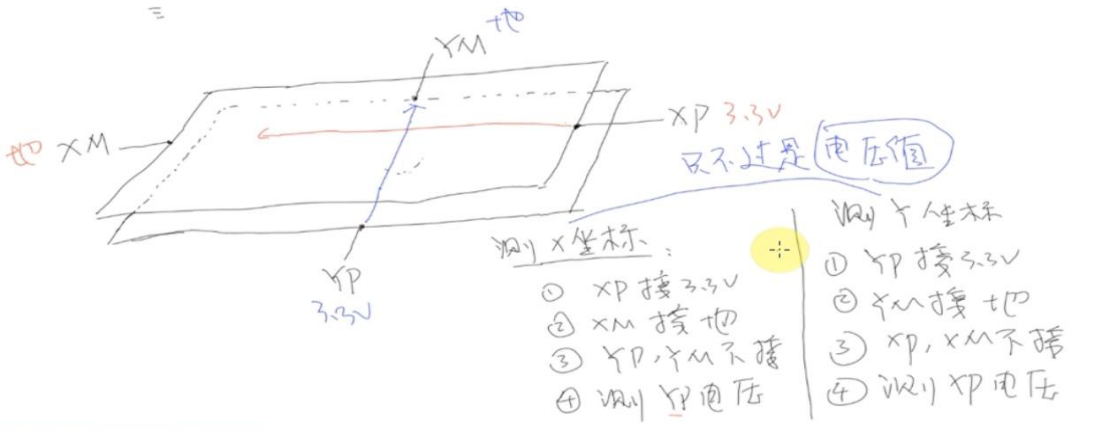
NOR FLASH 识别过程:
do_map_probe("cfi_probe", s3c_nor_map);
  drv = get_mtd_chip_driver(name)
  ret = drv->probe(map); // cfi_probe.c
    cfi_probe
      mtd_do_chip_probe(map, &cfi_chip_probe);
        cfi = genprobe_ident_chips(map, cp);
          genprobe_new_chip(map, cp, &cfi)
            cp->probe_chip(map, 0, NULL, cfi)
              cfi_probe_chip
                // 进入 CFI 模式
                cfi_send_gen_cmd(0x98, 0x55, base,
map, cfi, cfi->device_type, NULL);
                // 看是否能读出"QRY"
                qry_present(map,base,cfi)
                .....

```
do_map_probe("jedec_probe", s3c_nor_map);
    drv = get_mtd_chip_driver(name)
    ret = drv->probe(map);    // jedec_probe
            jedec_probe
                mtd_do_chip_probe(map, &jedec_chip_probe);
                    genprobe_ident_chips(map, cp);
                        genprobe_new_chip(map, cp, &cfi)
                            cp->probe_chip(map, 0, NULL, cfi)
                                jedec_probe_chip
                                    // 解锁
                                    cfi_send_gen_cmd(0xaa, cfi->addr_unlock1, base,
map, cfi, cfi->device_type, NULL);
                                    cfi_send_gen_cmd(0x55, cfi->addr_unlock2, base,
map, cfi, cfi->device_type, NULL);

                                    // 读 ID 命令
                                    cfi_send_gen_cmd(0x90, cfi->addr_unlock1, base,
map, cfi, cfi->device_type, NULL);

                                    // 得到厂家 ID，设备 ID
                                    cfi->mfr = jedec_read_mfr(map, base, cfi);
                                    cfi->id = jedec_read_id(map, base, cfi);

                                    // 和数组比较
                                    jedec_table
```

6.触摸屏（touch screen）

利用输入子系统，原理：相当于 AD 转换器，使用欧姆定律

两层不同的膜粘在一起



Ts use:

1. press,produce interrupt
2. in interrupt process,initialize ADC ,measure voltage

3. ADC complete，produce ADC interrupt
4. In ADC interrupt,input_event，initialize timer
5. timeout,go to 2.(deal long pressed slither)
6. unpress

ts 有等待中断模式，即按下中断和松开中断，只有设置这两个，才能不断进入中断

优化：1. ADC 测量延迟 （刚按下电压不稳定）

2.若 ADC 完成时，发现触笔已松开，放弃此次结果

3.多次测量取平均值

4.软件过滤

5.使用定时器处理长安，滑动的情况

7.PCI 总线：

总线是一种传输信号的信道；总线是连接一个或多个道题的电气连线。总线由电气接口和编程接口组成，重点关注编程接口。

PCI:peripheral component interconnect(外围设备互联)，是在 PC 机上使用更多

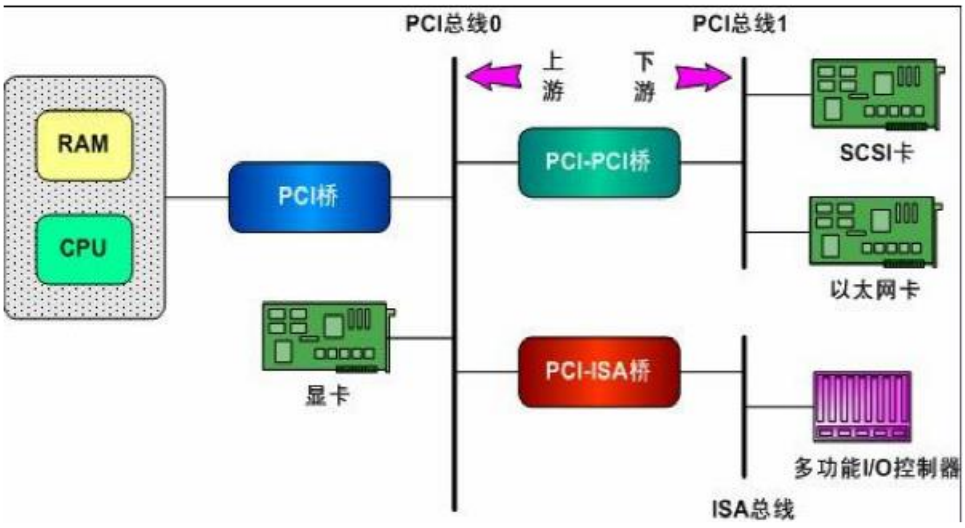
优点：

1.在计算机和外设间传输数据时具有更好的性能

2.能够尽量独立于具体的平台

3.可以方便地实现即插即用

只有 PCI 桥才能生成 PCI 总线



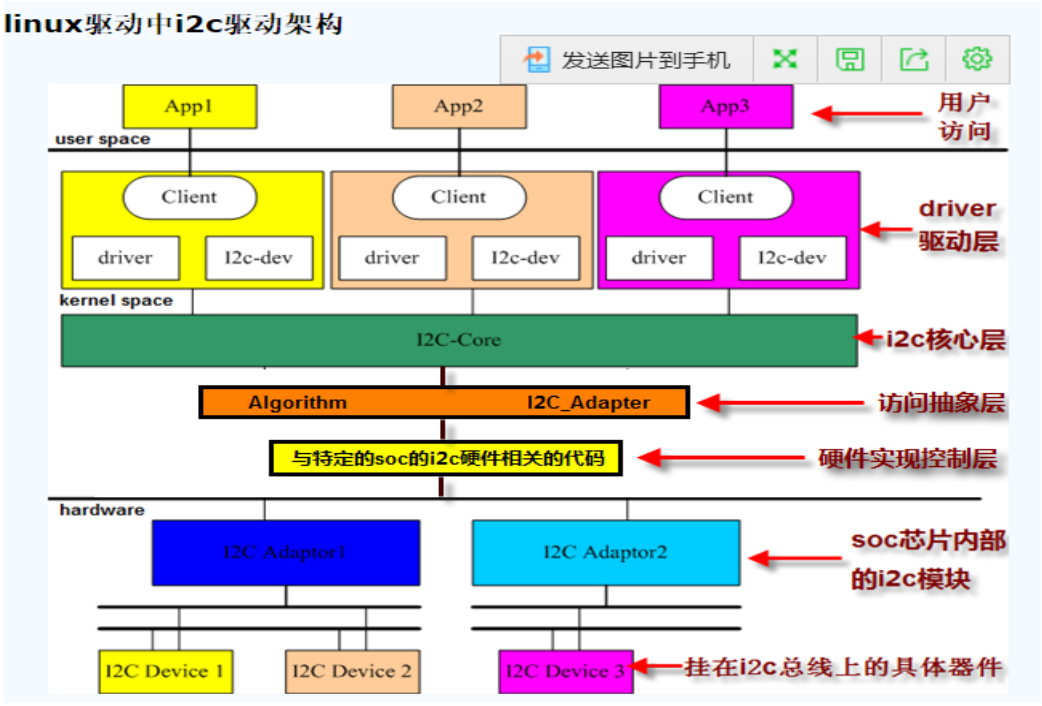
系统的各个部分通过 PCI 总线和 PCI-PCI 桥连接在一起。CPU 和 RAM 通过 PCI 桥连接到 PCI 总线 0（即主 PCI 总线），而具有 PCI 接口的显卡直接连接到主 PCI 总线上。PCI-PCI 桥是一个特殊的 PCI 设备，它负责将 PCI 总线 0 和 PCI 总线 1 连接在一起。图中连接到 PCI 1 号总线上的是 SCSI 卡和以太网卡。为了兼容旧的 ISA 总线标准，PCI 总线还可以通过 PCI-ISA 桥来连接 ISA 总线，从而支持以前的 ISA 设备，图中 ISA 总线上连接着一个多功能 I/O 控制器，用于控制键盘、鼠标和软驱等

PCI 设备寻址：

每个 PCI 设备由一个总线号、一个设备号、和一个功能号确定。PCI 规范允许一个系统最多拥有 256 条总线，每条总线最多带 32 个设备，但每个设备可以是最多 8 个功能的多功能板

（如一个音频设备带一个 CD-ROM 驱动
器）。
/proc/iomem 描述了系统中所有的设备 I/O 在内存地址空间上的映射。我们来看地址从 1G 开始的第一个设备在
/proc/iomem 中是如何描述的：40000000-400003ff : 0000:00:1f.1
这是一个 PCI 设备，40000000-400003ff 是它所映射的内存空间地址，占据了内存地址空间 1024 bytes 的位置，而 0000:00:1f.1 则是这个 PCI 外设的地址,它以冒号和逗号分隔为 4 个部分，第一个 16 位表示域，第二个 8 位表示一个总线号，第三个 5 位表示一个设备号，最后是 3 位，表示功能号
因为 PCI 规范允许单个系统拥有最多 256 条总线，所以总线编号是 8 位。每个总线上可支持 32 个设备，所以设备号是 5 位，而每个设备上最多可有 8 种功能，所以功能号是 3 位。由此 ，由此可以得出上述的 PCI 设备的地址是 0 号总线上的 31 号设备上的 1 号功能。

8.I2C：



架构层次分类

第一层：提供 i2c adapter 的硬件驱动，探测、初始化 i2c adapter（如申请 i2c 的 io 地址和中断号），驱动 soc 控制的 i2c adapter 在硬件上产生信号（start、stop、ack）以及处理 i2c 中断。覆盖图中的硬件实现层

第二层：提供 i2c adapter 的 algorithm，用具体适配器的 xxx_xferf()函数来填充 i2c_algorithm 的 master_xfer 函数指针，并把赋值后的 i2c_algorithm 再赋值给 i2c_adapter 的 algo 指针。覆盖图中的访问抽象层、i2c 核心层

第三层：实现 i2c 设备驱动中的 i2c_driver 接口，用具体的 i2c device 设备的 attach_adapter()、detach_adapter()方法赋值给 i2c_driver 的成员函数指针。实现设备 device 与总线（或者叫 adapter）的挂接。覆盖图中的 driver 驱动层

第四层：实现 i2c 设备所对应的具体 device 的驱动，i2c_driver 只是实现设备与总线的挂接，而挂接在总线上的设备则是千差万别的，所以要实现具体设备 device 的 write()、read()、

ioctl()等方法，赋值给 file_operations，然后注册字符设备（多数是字符设备）。覆盖图中的 driver 驱动层

第一层和第二层又叫 i2c 总线驱动(bus)，第三第四属于 i2c 设备驱动(device driver)。

在 linux 驱动架构中，几乎不需要驱动开发人员再添加 bus，因为 linux 内核几乎集成所有总线 bus，如 usb、pci、i2c 等等。并且总线 bus 中的(与特定硬件相关的代码)已由芯片提供商编写完成，例如三星的 s3c-2440 平台 i2c 总线 bus 为/drivers/i2c/buses/i2c-s3c2410.c，里面有各种函数，收发.
其中 adapter 函数，

第三第四层与特定 device 相干的就需要驱动工程师来实现了。

驱动架构：

i2c_add_driver

i2c_register_driver

driver->driver.bus = &i2c_bus_type;

driver_register(&driver->driver);

list_for_each_entry(adapter, &adapters, list) {

driver->attach_adapter(adapter);

i2c_probe(adapter, &addr_data, eeprom_detect);

i2c_probe_address // 发出 S 信号,发出设备地址(来自 addr_data)

i2c_smbus_xfer

i2c_smbus_xfer_emulated

i2c_transfer

adap->algo->master_xfer // s3c24xx_i2c_xfer

怎么写 I2C 设备驱动程序?

1. 分配一个 i2c_driver 结构体

2. 设置

attach_adapter // 它直接调用 i2c_probe(adap, 设备地址, 发现这个设备后要调用的函数);

detach_client // 卸载这个驱动后,如果之前发现能够支持的设备,则调用它来清理

3. 注册：i2c_add_driver

测试 1th:

1. insmod at24cxx.ko

观察输出信息

2. 修改 normal_addr 里的 0x50 为 0x60

编译加载，观察输出信息