# 第九讲 输入输出设备

## 901-输入输出接口的基本功能

> 计算机的核心能力就是运算，
>
> 而快速的数学运算，甚至是早期设计计算机时的唯一目标。
>
> 但是今天，计算机已经成为了我们日常生活中，密不可分的一部分。
>
> 我们平时感受更多的，可能是计算机与外界交互的能力， 因此，想要了解计算机的组成，我们就必须要知道
>
> 现代计算机当中，输入输出设备到底是怎么工作的。
>
> 我们先来一起回顾模型机的结构，
>
> 这个模型机由一个CPU，一个存储器，和一个输入输出设备组成。
>
> 在系统启动后，CPU就会从地址总线发出地址 给存储器，然后从数据总线读回对应的指令编码，
>
> 并通过指令译码部件，产生相应的控制信号，完成对应的操作， 那这些操作有可能是进行运算，
>
> 也可能是读写存储器当中的某个单元，那也有可能是要进行输入输出的动作，
>
> 那我们来看一看，CPU如果要完成一个简单的输入动作，会是怎么样的。
>
> 我们这里的输入设备是八个手动的开关，
>
> 我们假设前三个开关都拨到高电平，后五个开关 拨到低电平，那这八个开关可能连接到了一个寄存器，
>
> 那这个单元当中，就保存了11100000这个数， 而且事先在系统当中，也为这个单元分配了一个地址，
>
> 那么事先就编写了一个用于输入的程序，在这个程序当中，有一条指令就是读取1111这个- 地址单元。
>
> 那么当CPU执行了这条指令的时候，就会在地址总线上发出这个地址，
>
> 与此同时，在控制总线上发出表示当前是读操作的信号，
>
> 那这个输入输出设备收到这样地址和控制信号之后，就会
>
> 从1111这个单元，取出对应的内容，然后把它送到数据总线上去， 而CPU这时会采样数据总线上的信号，得到这个数值，
>
> 然后这条指令应该是把这个数，保存到某个通用寄存器当中去，
>
> 这样后续的程序也就可以对这个数进行操作了，这就完成了一个输入的动作。
>
> 那我们再来看一个输出的 简单场景，我们现在的输出设备是8个小灯泡，
>
> 它也和一个8位的寄存器相连，并有对应的地址， 那现在CPU执行到某一条指令，是想点亮这8个灯泡当中的某几个，
>
> 那这条指令就会在地址总线上送出1110，然后在控制总线上送出
>
> 写的控制信号，与此同时，还需要在数据总线上，送出要写的数据，
>
> 这个输入输出设备，就会根据控制总线发现是一次写操作，就找到

地址总线上的信号对应的单元，并将数据总线上对应的信号写进去，

于是1110这个单元，就被写入了11001100这个数，

然后这个单元的输出，就直接通过物理的连线，连接到8个小灯泡上，数值为1

代表高电平，就会点亮某个灯泡，数值为0，代表低电平，那对应的灯泡就是熄灭的状态。

这样我们就通过一条指令，完成了这几个小灯泡亮灭的控制， 从而让计算机外部的人，可以观察到输出的信息。

因此，由CPU、存储器和输入输出设备，就构成了一个简单的计算机系统。

用于输入输出的，可以是拨码开关、LED管，这样的的简单的设备，

也可能是比较复杂的设备，像打印机、硬盘，

那现在的计算机系统当中，输入输出设备变得越来越多，功能也非常的丰富。

这些设备的差异非常大，有些设备要求很高的数据传输率，比如说显示器，

有些设备的速度却很慢，比如键盘和鼠标，

而且有一些接口是串行的，有一些是并行的，有数字电路的接口，也有模拟电路的接口，

如此千差万别的设备，就没有办法直接和CPU这一个芯片进行连接，

因此我们就需要在CPU和这些设备之间，设置一个中转站，

这就是输入输出接口，也被称为I/O接口。

I/O接口主要提供了这些功能，1是数据缓冲， 用于解决高速的CPU和低速的外设之间的差距。

第2是提供联络信息，比如打印机什么时候能够接收数据。

第3是提供格式上的转换，比如模拟信号和数字信号之间的转换，

串行信号和并行信号之间的转换，不同电平之间的转换。

第4，一个接口可能连接了多个设备，比如说有多个硬盘， 那这个I/O接口还需要提供设备选择的功能。

其它还有一些功能，在用到的时候再做介绍。

因此，在现代的计算机系统当中，

往往把跟输入输出相关的设备分解成两个类型，一部分是I/O接口，

这可能是插在计算机主板上的一块插卡，也有可能是主板上的一个芯片，它内部

会有一些寄存器，CPU可以通过系统总线，去访问I/O接口当中的这些寄存器，

而这个I/O接口芯片，还会有一些管角，与外部的设备相连。

那这种划分的典型代表，就是显卡和显示器， 显卡是一个I/O接口，显示器则被称为外设，

那我们要注意的是，这样的划分和冯诺依曼结构 当中所定义的输入输出设备，是不能完全一一对应的，

在这里，我们描述的是现代计算机发展过程中的，一种具体的实践。

那I/O接口究竟是如何用外设相连的， 我们来看一个例子，这是一种最简单的I/O接口，

叫做并行接口，它一边通过系统总线和CPU相连， 另一边可以连接数码管、拨码开关，这样简单的设备，

也可以连接打印机这样比较复杂的设备。

那这样的并行接口，一般有两种常见的实现形式，一种是采用独立的芯片，比如这个

8255A，就是一个并行接口芯片。那么早期的I/O接口，大多会采用这样独立芯片- 的形式，

而现在可以在一个芯片当中，实现多个I/O接口的功能。

所以这个并行接口，也可能被实现在一个多功能的芯片当中， 但不管是哪种实现形式，它总是要与这些外设进行连接。

我们先来看看它和那些简单的外设是怎么连接的，

比如说和一个简单的输出设备，就是这个数码管，那这个并行接口的芯片，对外会有8根引角，

通过主板上的连线，就可以连接到这个数码管， 每根信号线，正好连接到其中一个可以发亮的部分，

加上旁边的这个小数点，一共有8个，因此，这和我们刚才用模型机演示的例子一样，

CPU只要将对应的数据写到这个数据输出寄存器上，

并行接口芯片就会直接将这个寄存器的内容，输出到这八根信号线上，

那这个数码管上对应的部分就会亮起，从而展现出我们编程想要输出的数字，

那类似的，这个并行接口芯片，还可以有8个引角，连接到拨码开关，

那拨码开关的状态，就会被保存在数据输入寄存器当中，等待CPU来读取，

这些都是非常简单的设备。那我们再看一些稍微复杂一点的设备，比如说打印机，

那么就不能像刚才那样简单的传入数据了，除了连接刚才那8根输出的信号之外，

还需要有一根信号，告诉打印机，此时输出的数据，是希望打印机接收的数据，

而打印机在接收了这个数据之后，还要给出一个回答的信号， 以表示它已经完成了这个数据的接收工作，

这样才能保证，这个并行接口和打印机进行数据传输时， 既不丢失数据，也不会传输到重复的数据。

那么对于扫描仪这样的输入设备，也是类似的情况，

扫描仪在将扫描了的数据进行输入时，也需要给出一个输入准备好的信号，

那并行接口电路在采样了数据输入之后，也需要给出输入的回答信号，

因此，并行接口芯片，在连接这些比较复杂的设备时，

除了那8根数据信号之外，还需要有两根用于通讯联络的信号，

而这两根信号，又被称为握手信号，在数据传输中， 起着协调和联络的作用，那对于稍微复杂一些的外设，

都得采用带握手信号的数据传输的方式。

在现代的计算机当中，这种并行接口电路，算是最简单的I/O接口了，

它和许多其它更为复杂的I/O接口，都会在集成在南桥芯片当中，

而还有少数对性能要求比较高的接口，则会采用独立的芯片，或者板卡的形式，

而在一些紧凑型的设备中，比如说平板电脑和智能手机，

这些I/O接口甚至会和CPU一起，集成在一个芯片当中。

那不管是哪种形式，这些I/O接口的功能都是独立存在的，

而且它们也需要各自的管角、连线，与对应的外设相连，从而让CPU可以与外部进行交互。

外面的世界很精彩，但也很复杂。

我们的CPU虽然又超群的计算能力，然而它的交往技巧却是很贫乏， 所以我们为它找到了I/O接口这个好帮手

## 902-输入输出接口的编址方式

CPU的运算能力很强， 但它与外界沟通交流的手段却非常地单一，

它总是希望有这样“我给你一个地址，你就给我一个数据”非常直白的沟通方法。

所以它平时也只能和存储器这样胸怀宽广，但是同样头脑简单的家伙在一起玩了。

然而现在它需要面对外界那么多的朋友，各个都非常复杂，有时还经常捣些乱，

那它恐怕就应付不了了，所以它就找了I/O接口这样的帮手帮它打理外部的世界，

而它呢还是希望能够用一个简单的方式和I/O接口进行交互和沟通。

所以我们就来看一看CPU是怎么和I/O进行沟通的。

这是I/O接口在计算机系统当中的位置，

和访问存储器中的单元一样，我们想要CPU访问I/O接口 当中的这些寄存器，也是需要通过编写指令来实现，

那问题就是，I/O接口里面的这些寄存器的地址究竟是什么？

我们先来看几个基本概念，在系统当中通常会有多个I/O接口，

每个I/O接口内部都有若干个寄存器，这些寄存器一般被称为I/O端口，

我们不要被这个词的字面意义所迷惑， 这个端口指的并不是我们计算机上的USB接口、网线接口这样实实在在的接口，

而是一个抽象的概念， 它实际上指的就是这些在I/O接口芯片内部的寄存器，它们就像在

存储器当中的一个个存储单元一样，CPU要访问它们，就得有特定的地址，

因此每个寄存器，也就是每个I/O端口都需要有自己的地址，

这称为端口地址，也叫做端口号。那在计算机系统中， 如何去设定这些端口号就称为I/O端口的编址方式。

常见的I/O端口编址方式有两种： 第一种是I/O端口和存储器分开编址，

又被称为I/O映像的方式，x86体系结构就采用了这种方式。

另一种常见的方式是I/O端口和存储器统一编址， 又称为存储器映像的I/O方式。

ARM、MIPS、PowerPC等体系结构都采用了这样的方式。

那我们先来看分开编址的方式。我们假设这个体系结构地址的宽度为3，

那它一共可以访问的地址单元就是2的三次方，总共8个，

如果每个单元是一个字节，那它的存储器最大就是8个字节， 然后我们需要在这个计算机系统当中增加一些I/O端口，

那I/O端口的地址是重新编排的，和存储器地址无关。

一般情况下，我们需要的I/O端口的数量都比存储器单元要少得多，

比如在这个事例的系统当中，我们需要四个I/O端口，

那我们就给它分配四个端口号，0、1、2、3，这样的编址方式就称为分开编址。

在这种编址方式下，要访问I/O端口需要用特殊的指令， x86提供了IN和OUT这两条指令，

IN指令用于把I/O端口的内容输入到CPU当中的寄存器， 而OUT指令则是把CPU寄存器当中的内容输出到I/O端口中。

那我们来看一看IN和OUT指令应该如何书写。

如果要访问的端口地址在0到255之间，那可以采用两种方式。

一是叫直接寻址，也就是写一个立即数指定端口地址，

例如这条指令，这个80H就是一个端口号，那这条指令说的是

从80H这个端口读出一个字节的内容，并存放到AL寄存器当中去，

而这条指令则是说，将AX当中的两个字节的内容，传送到80H

所指定的I/O端口中，当然这个I/O端口对应的应该是一个两字节的寄存器。

而如果端口地址大于255，则需要将这个地址先保存到DX寄存器当中，

然后再执行IN和OUT的操作。我们也来看一个例子， 假如我们要访问第288号端口，

这时候就需要先把这个端口号存到DX寄存器当中， 然后在IN和OUT指令当中，使用DX寄存器来指定端口号

这样就是对288号端口地址进行操作。

当然，对于端口地址在0到255之间的，也可以使用间接寻址 的方式，那么x86为什么要设定这两种方式呢?

主要还是为了指令的长度， 我们看到在直接寻址的情况下，我们需要有一个字节的操作码，

还需要有一个字节保存这个端口号，那么一个字节所能表达的范围就是0到255，

如果端口地址大于255，那原本就需要再增加一个字节来记录端口号，但这样

指令就太长了，为了缩短指令长度，宁可多增加一个指令，这也是CISC的特点。

所以对于间接寻址，有另外一个操作码，

这条指令只有一个字节，既没有立即数，也没有寄存器的编号， 所以它是默认地采用DX寄存器来保存端口地址，

这样在访问更大的端口地址时，指令的长度反而可以更短一些。

那这里就有一个问题，既然用这个方式能访问的地址范围更大，指令长度还更短，

那我们为什么不只用这一种方式呢？还需要去直接寻址这样的方式？ 这个问题就留给你自己思考。

那我们再通过一个例子来看一看这样的指令的操作过程。

这条指令是AL寄存器当中的一个字节，传送到21H这个端口号，

那么当CPU从存储器当中取回了这条指令，通过译码发现是一条OUT指令，

那它就会将AL寄存器当中的内容取出来，放到数据总线上，并将21H放到地址总线上，

那么这时候系统总线应该怎么办呢?

我们不妨把系统总线看成城市中的一条街道，而把系统总线所连接的存储器I/O接口看成

街道两旁的一些建筑，每个建筑里面还有很多个单位，每个单位都有一个门牌号，

那现在就好像有一个快递员接收了一个任务，要把一个包裹送到21H这个地址，

于是他就在这个街道上走，查看着每个大楼的门牌号，

然后他发现，这两个存储器地址范围，一个是从0到7FFF，

一个是从80000到5个F。那么在存储 器1当中，实际上是包含了21H这个地址的，

但他接着再看，这个I/O接口1当中的地址是00到1F，

而I/O接口2当中的地址是20到3F，I/O接口3的地址是40

到5F，所以在I/O接口2当中，也包含了21H这个地址。

那这个包裹应该送到哪儿呢?

所以单凭这个地址，系统总线是无法判定要访问哪个设备的。因此，CPU发出的信号中，

除了地址，还应该有一个别的信号，这个信号指明了当天要访问的是存储器还是I/O接口。

在x86的CPU当中，这个信号叫做M/IO，

当这个信号为0的时候，表明当前在访问I/O接口，而这个信号为1的时候，表明在

访问存储器，这样系统总线就知道该怎么办了，它会在所有的I/O接口当中，

寻找这个地址所对应的端口，那么这就发现是在I/O接口2中，

所以系统总线会把这个传输传到I/O接口2，I/O接口2可能是一个独立的芯片，

当它在系统总线上采样到这个地址和一个数据之后，就会在内部找到对应的端口号。

我们要注意的一点是，这些I/O接口内部一般只有 少数几个端口，所以它只会采样地址的低几位，

然后用这个低位在内部进行索引。在这个例子当中，21H的这个2

是系统总线用来找到这个I/O接口的，而这个I/O接口只用

接收到地址的低位这个1，然后在内部找到对应的端口就可以了。

最后这个I/O接口将从数据总线上采样到数据，也就是AL寄存器当中的内容，

保存到它内部的数据输出寄存器中，这就完成了这条OUT指令所需的操作。

至于这个数据输出寄存器它外面是连接到了几个小灯泡，还是一个数码管，

那就是这个I/O接口和外射的连接情况了。

然后我们再来看一看I/O端口和存储器统一编址的情况。

还是假设地址宽度为3，那我们在这个统一编址的体系结构当中，总共就只有8个单元，

然后根据需要，其中有一部分用来作为I/O端口的地址， 其他部分用来作为存储单元的地址。

那我们之前介绍的模型机就采用了统一编址的方式，

它的地址总线宽度为4位，这样一共就有16个单元，

而存储器当中用了的地址是0到7，一共8个单元。

而输入、输出设备则用了15和16这两个地址， 另外还有一些地址在这个系统当中没有使用，

那在之后扩展中，可以增加一些存储器，或者增加一些输入、输出端口， 但是总共只有这16个，是不可以重复的。

当然，因为它是统一编址的，所以给出任意一个地址，只有唯一的一个单元与之对应，

所以也就不需要刚才x86当中使用的M/IO这样的信号来指定当前的地址 到底是I/O地址还是存储器地址。

那我们来看一看统一编址方式的优缺点。

首先来看优点。因为在统一编址的情况下，是不区分存储器地址和I/O端口地址的，

所以我们就可以直接用访问存储器的指令来访问I/O端口，而访问

存储器的指令功能通常比较丰富，比如数据可以有各种的宽度， 地址也有多种的产生方式，可以

是立即数，可以是寄存器，也可以是寄存器加立即数，甚至还可以放在存储器当中，

所以这样就比较方便对I/O端口进行处理。另外，如果要涉及

单独的I/O操作的指令，无论它做得怎么简单，也是需要额外的一套硬件逻辑，

而采用统一编址的方式，CPU中只需要有一套对外部总线的控制逻辑就可以了，

内部结构简单，对外的引脚数目也会少一些。

但是统一编址也有它的缺点，由于I/O端口占用了 一部分地址空间，从而使用于存储器的地址空间变小了，

这个问题对于早期的处理器影响还是非常大的，

我们想一想，x86的早期只有16位宽的地址，按说只能访问64KB的存储器，

它为了有更多的存储器空间，还设计了一个非常复杂的 段加偏移的方式，才能访问一兆的地址空间，

如果在这个时候还要为了I/O端口占用了一部分地址空间，那就很难接受了，

这也是x86选用了分开编址方式的一个重要原因。

而当CPU的字长到了32位之后， 在很长一段时间，地址空间都不是一个问题，

所以那个时候直接从32位起步的MIPS就采用了统一编址的方式，

当然现在到了64位之后，物理的存储器远远用不了这么大的地址空间，

所以地址空间被挤占这个因素现在已经不成问题了。

另外，如果要用访问存储器的指令来进行I/O操作，那这些指令往往比单独

设计的I/O指令要长，而且因为这些指令比较复杂，执行的时间可能也会长一些，

这也是risk为什么普遍采用了统一编址方式的 原因，因为risk的指令都是固定长度的，

所以即使设计单独的I/O指令，也不会比普通的访存指令更短一些，

而x86这样的cisc采用了变长的指令， 所以就可以设计出更短的专门用于I/O的指令，从而提高指令的密度。

那么了解了统一编址的特点、分开编址的特点，我们也就清楚了 他们的优缺点刚好是相对的，在分开编址的

情况下，I/O端口不会挤占存储器的地址空间，

而且因为涉及了单独的I/O指令，它的指令编码可以做得很短，执行速度也比较快，

而I/O地址空间一般是远远小于存储器地址空间的，

所以独立的I/O指令可以使用较短的地址编码，从而让地址译码变得更为方便。

另外从软件编程的角度来看， 有了单独的IN和OUT指令，可以很清晰地看出哪些是I/O操作，

> 哪些是存储器操作，让程序的结构变得清晰易懂。
>
> 而分开编址的缺点，我们刚才也都已经说完了，就不再重复了。

> 现在CPU可以用它习惯的方式和I/O接口进行交互，
>
> 这确实让事情变得简单了很多，但是这并不意味着CPU就可以做iii掌柜了。
>
> 其实I/O接口只能帮CPU解决一些沟通交互上的琐碎细节， 真正的沟通的核心内容还得CPU自己来做。

## 903-输入输出的控制方式

> 那现在看来，跟外界交互也没有那么复杂嘛。
>
> 跟以前与存储器的交互差不了多少。我给一个地址，的到一个数，
>
> 或者给一个地址，写进去一个数，这样就跟外设进行交互了。
>
> 那这么说也不是不对，最简单的情况确实是这样的。
>
> 那我们也不妨从这最简单情况开始说起吧。

> CPU控制外设进行输入输出的方式主要有三种。
>
> 第一种叫做程序控制方式，第二种叫做 中断控制方式，第三种叫做直接存储器访问方式。
>
> 那我们先来看第一种程序控制方式。
>
> 我们还是用这个并行接口作为例子，来讲解程序控制方式是如何工作的。
>
> 程序控制方式，顾名思义， 其数据传送都是在程序的控制下进行的。
>
> 那这个方式又有两种具体的形式。
>
> 第一种成为无条件传送方式，这种方式适合于简单外设的操作。
>
> 我们假设外设始终处于准备好的状态，直接编写输入输出指令进行传送数据就可以了。
>
> 这样编写出来的控制程序可以非常的简单。
>
> 那像拨码开关、数码管，都是这样的简单外设，适合采用无条件传送的方式。
>
> 比如要控制数码管这样的设备，我们只用在程序中简单写一个out指令，
>
> 把想要点亮的数码管的对应的比特置一。
>
> 而并行接口则会将这个数直接放到并行数据输出线上， 那这次传送就完成了。同样，如果要进行输入，
>
> 因为这些拨码开关连接到了并行接口的数据输入寄存器， 所以CPU只需要简单写一个in指令，
>
> 从数据输入寄存器这个端口读入数据，就可以得到当前拨码开关的值。
>
> 但是对于稍微复杂一点的设备，这样的方式就不适用了。
>
> 我们就得采用程序控制方式当中的另一种，叫做程序查询传送方式。
>
> 在这种方式下，我们先得编写一段程序，用来查询外设的工作状态。
>
> 在确定外设已经准备就绪的时候，才进行数据的传送。
>
> 我们以打印机和扫描仪这样的设备为例， 这些设备也是可以连接在并行接口上的。
>
> 但是除了刚才提到的那八位并行数据线，还应该 有两根信号线来标明发送方和接收方是否已经准备好。

对于输入设备和输出设备都是如此。

那这对用于交互联络的信号，就叫做握手信号。

在数据传输的每一步，都需要通过握手信号进行确认，然后才可以进行下一步的传输。

那我们先通过数据输出的过程，来看一看程序查询方式是怎么工作的。

现在CPU通过系统总线连到了并行接口，

而并行接口的外面连接了一个输出设备，可以是一个打印机。

那我们现在要编写一段程序，希望在打印机上打印出一段文字。

那首先CPU要执行out指令，去写接口当中的控制寄存器。

控制寄存器当中由若干个位域组成，这些位域有着不同 的含义，通常是用来设置这个并行接口的工作模式。

比如说接口的时钟频率应该是多少，对外的接口信号是高电频有效，还是低电频有效。

那我们要写进控制寄存器的这个数值，常被称作控制字。

那在通常情况下，在初始化的时候写入一次控制字

就可以了。然后这个接口就会按照这样设置的工作模式一直工作下去。

除非要改变工作模式，才需要重新写入控制字。

那好，现在设置好了工作模式，我们就可以正式开始传输了。

那么在这段程序里面，接下来要写一个out指令， 将我们要打印的数据传送到这个接口的输出缓冲寄存器中。

那当这条out指令执行完之后，并行接口的硬件电路 就会将输出缓冲寄存器当中的内容送到并行数据输出信号线上，

并且会将输出准备好信号置为有效。

而对于这个打印机外设来说，它一直在监测输出准备好这根信号。

一旦发现这根信号有效，就会从并行数据输出信号上采样数据，

然后将输出回答信号置为有效，表示它已经接收到了这个数据。

这个数据可能表示一个字符， 那现在打印机就可以把这个字符打印在纸上了。

我们要注意，在这种情况下，这个握手信号是必须的。

因为打印机这样的外设，单纯看并行数据输出这组信号，并不能判断当前是否来了数据。

因为这组数据线即使是全0，也可能是有含义的。所以必须要等到

并行接口发出了这个输出准备好信号，它才会去采样，而且只采样一次。

在打印机将输出回答信号置为有效后，它又处于等待状态。

那这个并行接口芯片发现输出回答信号有效之后， 就会让状态寄存器当中的某一位置为有效。

这个状态位叫做输出缓冲空，表示当前输出缓冲寄存器中已经没有需要发送的数据了。

而在刚才的这个过程当中，CPU实际上还在反复的

执行指令，不断的从状态寄存器这个端口中，把数据读出来。

那么从状态寄存器当中读出的这个数，就称为状态字。

因为现在这段程序是在做数据输出，所以CPU读出状态字之后，

就会检查当前的这个状态字中输出缓冲空，这个状态位是否是有效的。

如果无效，那就再执行一次硬指令，继续读状态寄存器。

所以我们可以想像，这段程序应该是一条硬指令。

然后是几条逻辑运算指令，去检查对应的状态位是否有效。然后是一条条件转移指令，

决定是否要跳回去继续执行那条硬指令。

那么直到有一次读出来的状态字中输出缓冲空这个状态位是有效的，

那CPU就会将下一个要打印的字符，用out指令送到

并行接口的输出缓冲寄存器当中去，然后开始下一轮的输出过程。

这样不断的执行，打印机就可以打出很多的文字来了。

那我们再来看一个数据输入的过程。

我们可以假象这个外设是一个扫描仪。那我们现在启动一个控制扫描仪的程序，

那同样在初始化时，CPU会执行out指令，将控制字

写入接口的控制寄存器当中，设置好这个并行接口的工作模式。

然后我们操作这个扫描仪开始工作。

那扫描仪可能扫完了一张图片，但是先要存在扫描仪内部的缓冲区当中。

然后将其中第一个字节放到并行数据输入信号线上， 然后将输入准备好信号置为有效。

而并行接口的硬件一直在检查输入准备好信号是否有效。

一旦发现这根信号有效，就会采样并行数据 输入信号线上的内容，然后放入输入缓冲寄存器当中。

同时将输入回答信号置为有效，这个信号是告诉外设

先不要传送新的数据。因为我们现在设计的这个并行接口，它的

输入缓冲寄存器只有一个字节，所以一次只能接收一个字符。

如果是更为先进的并行接口，这个输入缓冲 可能会更大一些，这样一次就可以接收更多的数据。

那我们现在还是只接收了这一个字节。

接收完之后，接口还是需要将状态寄存器

当中输入缓冲满这个状态位置为有效。而在这个过程中，

CPU应该一直在执行一段循环代码，从状态寄存器当中不断的读出状态字，

然后检查输入缓冲满这个状态位是否有效。

一旦发现这个状态位有效，就会继续执行后面的指令。

那后面就应该是一条硬指令，从输入缓冲寄存器这个端口读入数据。

这就是从扫描仪传过来的第一个字符。

一旦CPU将这个数据读走了，那并行接口电路 就会将输入回答信号置为无效，等待外设输入新的数据。

当然同时也会将状态寄存器当中输入缓冲满这个状态位置为无效。

> 那在这个过程中，扫描仪一直在监测输入回答这个信号。一旦发现无效了，
>
> 它就将它自己内部缓冲区当中的第二个字符，送到并行数据输入上。
>
> 同时再次将输入准备好信号置为有效，
>
> 这样就开始了第二个字符的传输工作。如此往复，就可以完成很多数据的传输。
>
> 那我们再来看一看程序控制方式的优缺点。
>
> 其中的无条件传送方式，优点是要编写的控制程序非常的简单。
>
> 但缺点是只适用于非常简单的外设的操作。
>
> 像打印机这种还不算太复杂的设备，就已经不适用了。而程序查询传送方式，
>
> 由于有了握手的过程，比无条件传送方式准确和可靠。
>
> 但它的缺点是查询外设状态占用了大量的时间。
>
> 那这两种方式还有一些共同的特点。
>
> 它的优点在于对外设的要求比较低，不需要外设在接口上有很复杂的功能。
>
> 而且控制程序的操作流程非常的清晰。
>
> 但是缺点在于，要靠CPU进行数据的传送， 如果要传送的数据量很大，除了查询外设状态花的时间
>
> 之外，一个一个传送数据也需要花费大量的时间，
>
> 占用了CPU非常宝贵的运算资源，从而可能影响这个计算机系统的性能。

> 程序控制方式确实简单易行，
>
> 那对于出门开关一个小灯泡这样的事情，应该是没有问题的。
>
> 但是如果稍微复杂一些的情况呢，比如说我定了一个外卖，要送一份午餐过来。
>
> 那我不知道它什么时候才能送到，那我们如果用程序控制的方式， 比如说用无条件传送的方式，
>
> 那我在看的这个书，做一条条运算的之后呢， 突然一条写的，去校门口把这个外卖拿回来。
>
> 我就放下本，我去了，到了校门口， 不管是什么东西，看见就拿，啪，给拿回来了。
>
> 然后拿回来以后呢，不管是什么，张嘴就吃，就把这当午餐给吃了。
>
> 这恐怕不太合适，对吧，肯定会出问题。所以那我们用程序查询模式呢，
>
> 那么就是这样。好，我们做的运算。这个 要去拿外卖了，然后就说，诶，去校门口看一眼，
>
> 有没有外卖，好，我就去了。校门口一看，哎呀，没有，回来了。
>
> 然后写着， 那个再去校门口看一眼，好，放下，再去校门口看一眼，没有，再回来。
>
> 一直在做这个循环，一直放、看、没有、看、没有。然后直到又一次，看到有了，
>
> 就把这个盒饭拿回来，就可以吃上这顿午饭了。饭虽然是吃上了，
>
> 但是这事儿恐怕就干不成了，这一整天，啥也没干，就光顾等这饭了。
>
> 所以这个方式呢，看来还是不太实用， 我们还得想新的办法。

## 904-中断控制方式

> 那原本为了好好看书好好进行运算，我就不出去吃中饭了，

我定了一份外卖。结果呢？为了能收到这份外卖，我不得不总是去校门口看一看外卖来了

没有。回去看一遍，回来又看一遍，再看一遍，再看一遍。

那结果一上午，啥书都没有看成，对吧。这样就不合算了。

我都还不如玩会儿游戏呢，折腾这个干嘛？ 那怎么解决呢？我们得追加投资。啊，增加一个设备，

我们装一部电话。

装一部电话呢，然后跟送外卖的说好，你到了校门口就给我打电话。

那这样呢我就可以安心地坐在这先玩游戏了。

啊，那玩儿，啊，玩的很开心。咦，这个突然电话响了，电话响了。

然后"唉，啊送外卖的， 好好好，这就来，

这就来"，好，这样我们就 可以去拿了外卖回来吃完，对吧，什么事都不耽误。

这就是我们今天要说的"中断控制方式"。

在现在的计算机系统当中，

如果完全依靠CPU进行输入输出，那是有问题的。

比如说我们要通过键盘进行输入，那CPU怎么能保证及时发现键盘的输入呢？ 如果CPU不间断地去查询键盘的输入，

那显然是不合适的，那这样CPU就无法完成其他的任务了。

那如果是间歇地查询呢？先做一段运算的任务，然后查看一下键盘。

那这样也有问题。这个间隔设置为多久合适呢？ 间隔设置的太短，那还是影响CPU执行其他的任务。

间隔设置的太长，又会造成键盘的输入无法得到及时的响应。

所以想要让这个系统正常的工作，我们还得用其他的方法。

这就是我们要介绍的第二种I/O控制方式，中断。

之前我们已经知道了CPU是如何处理中断的， 并且分析了内部中断的不同类型。那我们现在要谈的是外部中断。

那我们现在就来了解CPU是如何运用中断的方式来进行I/O控制的。

我们先来看一个数据输入的过程，还是用这个并行接口电路为例。

那之前在这个结构上我们已经讲解了如何用程序查询方式进行数据的传送。

那想要实现中断控制方式，首先在这个并行接口电路当中，就需要有能够产生中断信号的逻- 辑电路，

电路，而且要把这个中断请求信号通过主板上的物理连线和CPU的中断输入接口连接起来。

然后我们再来看软件操作的过程。那在初始化时， CPU还是要执行OUT指令，将控制字写入到这个接口的控制计存器当中，设置

接口的工作模式。

然后CPU就可以去执行其他的任务了。

那当外设有数据输入的时候， 就会将数据发到这个并行数据输入信号上，并且把"输入准备好"这个信号置为有效。

而这个接口电路始终在检查”输入准备好“信号是否有效，

一旦发现有效，就从”并行数据输入"信号上接收数据，放入到“输入缓冲寄存器”当中去，

然后将“输入回答”信号置为有效，这是为了告诉外设暂时先不要发送新的数据。

那并行接口的这几步操作和程序查询方式下是一样的。

区别在于这个期间CPU不用来反复地读取状态计存器 以查询是否有输入的数据。

那在数据已经保存到输入缓冲寄存器当中后，并行接口电路会通过中断控制逻辑

将这个中断请求信号置为有效，同时 会将状态寄存器当中“输入缓冲满”这个状态位置为有效。

那CPU在发现这个中断请求有效后，就进入了中断处理的过程。

随后会进入我们专门为并行接口编写的中断服务程序。

那在这段中断服务程序当中，会首先使用硬指令读入“状态寄存器”当中的状态字，

因为这时CPU虽然是收到了中断请求，但它并不清楚是什么原因发生的中断。

所以它需要读出状态字，然后检查其中的各个状态位。

那发现是这个“输入缓冲满”的状态位有效了， 这时才知道是因为有数据的输入发生的中断。

那在这个中断服务程序当中，就会继续执行硬指令， 从这个“输入缓冲寄存器”当中读出数据。

CPU可能会对这个读出的数据进入处理，但更多的时候， CPU会将输入的数据再通过存储器写指令，

保存到设置在内存某处的一个缓存区当中。

而在CPU将这个输入缓冲寄存器当中的数据读走之后， 并行接口电路就会将输入回答信号置为无效。

这就告诉外设可以继续接收下一个新数据了。

那如果这是一个键盘的接口，可能会过很久才会有一个新数据。

那如果这是一个网络的接口，那可能会连续的有很多新的数据进来。

那刚才介绍的这段过程，就会不断地反复执行。

输入的数据会不断地被CPU读走，再写到内存的某个地方。

这样就构成了一大块数据输入的过程。

然后我们再来看看数据输出的过程。

那刚才说要用中断控制方式，是因为不知道输入什么时候发生。

那输出是由CPU控制的，这难道也会不知道什么时候发生吗？ 这当然不是。那为什么要用中断控制方式呢？

那我们不妨先来看一看这个过程。第一步还是设置接口的工作模式。

然后当CPU需要输出时，就执行OUT指令，将数据写到接口的输出缓冲寄存器当中。

然后接口将这个数据发到“并行数据输出”信号上，并将“输出准备好”置为有效。

那我们要注意的是，大多数对外接口的工作频率都是远远低于CPU的工作频率，

而且外设很可能会处在没有准备好的状态。

所以如果不采用中断的方式，CPU在这时就不得不反复地读取状态寄存器，

以确定当前的这个数据已经写到外设了，可以发送新的数据。

那这样也就浪费了CPU的性能。所以我们也可以采用中断控制的方式。

等到外设已经将这个数据取走之后， 再由并行接口电路将中断请求信号置为有效。

同时将状态寄存器当中的“输出缓冲空”状态位置为有效。

那这时CPU可能还在执行别的程序。

在收到中断请求后，又进入了并行接口的中断服务程序。

这和刚才数据输入过程发生的中断使用的是同一个中断向量，进入的也是同一个中断服务程序。

这个中断服务程序的一开头，就会去读取状态寄存器，

然后逐个检查其中的个个状态位。那这次发现的是“输出缓冲空”，

所以CPU就会将下一个要输出的数据再写到接口的“输出缓冲寄存器”当中去， 从而开始了下一个输出过程。

这样反复地执行刚才介绍的这几个步骤，那我们就可以完成很多数据的输出了。

现在我们来分析一下中断控制方式的特点。

先来看一看优点，有了中断控制方式，CPU 就可以和外设在一定程度上并行的工作，提高了工作的效率，

而且这也让外设有了申请服务的主动权。

而不是让程序查询方式那样，CPU什么时候查到你了，才有可能为你提供服务。

现在外设就可以在有需要服务的时候主动向CPU提出请求，

这样在一定程度上，也满足了输入输出处理的实时性要求。

然后我们再来看一看中断控制方式的缺点。

那它最大的缺点就输入输出数据的传送工作任然用CPU来承担。

这样就占用了宝贵的CPU运算资源，而且在存储器和外设之间进行数据的传送，

这种方式仍然需要将数据先放到CPU当中的通用寄存器，

这样通过CPU中转一次的传输过程就会显得很冗长，也影响了输入输出的性能。

当然程序查询方式同样也有这些缺点。

中断控制方式是节省了其中不断查询状态位的那些工作。

但是中断的机制也引入了一些新的开销。比如要进入和退出中断服务程序，

就需要执行额外的一些指令，这些指令和数据传送本身是没有关系的。

而且在现在复杂的操作系统当中，这样额外指令的数目可能还是非常多的。

那这样是不是说，中断方式就比程序查询方式要好呢?

倒也未必。中断方式确实提高了整个系统运行的效率。

但是对于一些特别重要的事情，查询方式反而能提高响应的速度。

比如说你现在要等的不是快递，而是你的女朋友。

那你就应该一早就到校门口去候着呀，唉一直在看， 来了没，来了没，对吧。那么这样第一时间到的时候，你就能接下来了。

> 可你现在非要想着怎么提高系统的整体效率， 就跟人家说，唉你到了之后就给我打电话，啊，我接了电话就去找你。
>
> 然后呢？你就呆在房间里啊， 玩游戏。玩得很高兴，玩儿，玩儿，玩儿，唉，一会儿电话响了，你还不接。
>
> 为什么呢，还得先存盘呢，对吧。保存现场，啊。
>
> 把菜单调出来，存，存，存，啊，存好了。存好了再准备 接电话。结果电话不响了，人家生气了，走了。
>
> 是啊，你倒是提高了整体系统运行的效率，
>
> 以后能给你打电话的也只有快递员了。

## 905-外部中断的处理过程

> 我们之前已经学习了内部中断的处理方法，
>
> 那么外部中断和内部中断总体上是很类似的。
>
> 只是在一些具体的处理细节上和硬件的连接方式上有所不同。
>
> 那我们现在就来看一看外部中断是如何处理的。
>
> 外部中断，也叫做硬件中断。
>
> 这是由CPU外部的中断请求信号启动的中断。以x86 CPU为例，
>
> 连到外部的中断请求信号一共有两个。一个信号叫做NMI， 这就是“非屏蔽中断”的缩写。
>
> 另一个信号叫INTR，这就是“中断”这个词的缩写。
>
> 相对于非屏蔽中断，我们一般也称它为可屏蔽中断。
>
> 不光是x86，其他很多别的体系结构的CPU也往往提供这两种类型的中断引脚。
>
> 这两个信号，是对应了CPU上真实存在的两个管脚的。
>
> 来自外设的中断请求信号就可以通过主板上的连线，连到这两个管脚上。
>
> 这个NMI，也就是非屏蔽中断，一般会连接一个 在这个系统当中非常重要，不希望被屏蔽的中断请求信号。
>
> 那至于什么是“非常重要”，就取决于系统设计者的观点。
>
> 例如在某些计算机当中，会将表示电池即将没电的这个信号连接到NMI上。
>
> 电池马上就要没电了，确实是一个非常紧急的情况。因为非屏蔽中断是不受中断允许标志- 的影响，
>
> 即使CPU现在将IF标志位设为0，关闭对外部中断的响应，
>
> CPU仍然会响应这个NMI的中断请求。
>
> 那这样就可以调用中断服务程序， 在断电之前把一些重要的信息保存到硬盘上去。
>
> 当然不同的系统可能会连接不同的中断请求信号到NMI上，
>
> 但都会是一些非常重大，不处理就会导致严重错误的事件。
>
> 一般的外设，它的中断请求都会连接到可屏蔽中断上， 但是CPU的可屏蔽中断信号的请求输入只有一根，
>
> 那就需要通过一些转换电路。现在计算机当中比较常见的是使用中断控制器这个芯片。

中断控制器会将外设输出的中断请求信号作为它的输入连接进来，

然后输出一根信号连接到CPU的可屏蔽中断请求信号上。

这个中断控制器也可以看作是一个I/O接口， 它内部也有一些被称为I/O端口的寄存器，

CPU可以访问这些端口，对中断控制器进行配置。

例如可以配置这些外设的中断请求，哪个优先级高，哪个优先级低， 或者可以在这些中断请求当中，屏蔽其中的一部分。

这些都是中断控制器的基本功能。

因为这样的中断控制器 是可以由编写程序进行配置的，所以又称为

可编程中断控制器，简称PIC。

有一个广泛使用的可编程中断控制器，就是英特尔的8259。

我们可以看到8259上，从IR0一直到IR7，

一共有8个中断请求的输入可以用来连接外设的中断请求信号，

然后它还有一些地址和数据信号，用以连接到系统总线上。

CPU对I/O端口的访问，就通过这些信号线来传递。

rNT信号则是由中断控制器发出的中断请求信号， 连接到CPU的可屏蔽中断请求信号上。

实际上CPU在收到中断请求后，还会发出一个中断响应信号，

这个信号会被连接到中断控制器的INTA引脚上。

后来在8259的基础上，又有了一些升级和功能的扩充，

这就是后来的高级可编程中断控制器，简称为APIC。

那么再来看一看中断控制器在系统当中的连接情况。

IBM PC/XT当中，就使用了一片8259。

8259和CPU之间，有中断请求信号和中断响应信号。

而另一端，8259则连接了来自各个外设的中断请求信号， 比如有来自定时器，有键盘，有串行接口，

有硬盘，有软盘，还有打印机。

那2号中断请求信号是被保留的，并没有连接外设。

等一会儿我们还会提到，这根保留的中断请求信号用来做什么了。

那这是三十多年前最早的个人计算机了， 在现在的个人计算机当中，有些设备都已经没有了，比如软盘。

但是现代的计算机系统仍然要遵守最早的个人计算机当中确定的一些规则。

那么就以键盘所对应的这个中断为例，来看一看现在最新的个人计算机当中的情况。

如果我们在一台装了Windows操作系统的个人计算机上， 调出设备管理器，然后在设备管理器中找到键盘，

用右键看这个键盘的属性，我们就会发现， 它的中断请求信号仍然是连接到中断控制器的1号接口，

仍然遵守三十多年前，IBM PC所制定的规范。

那在刚才的那个例子中我们可以看到， 一个中断控制器最多可以连接七个外设，

那如果有更多的外设应该怎么办呢？我们可以在系统中再增加一个中断控制器，

那这个中断控制器的请求信号不是连接到CPU的， 而是连接到原有的中断控制器的2号中断请求信号上。

那在这个中断控制器上，又可以再连接更多的外设。

这样两级中断控制器的结构，在早期的个人计算机当中使用了很长时间。

不过现在情况有一点不太一样，

因为很多I/O接口都集成到了南桥当中，所以这些I/O接口的中断请求信号实际上都是在- 南桥内部了。

因此在南桥内，一般也会实现一个中断控制器， 当然现在是APIC这样的中断控制器了。

这个中断控制器负责接收所有I/O接口的中断请求信号， 包括南桥内部集成的，和在外部独立的I/O接口。

它会将中断请求信号再送到CPU中去。

而现在的计算机当中，往往有多个CPU， 其实每个CPU当中，都还会带一个中断控制器。

因为现在的CPU不但要接受中断请求信号，它也会发出中断，

现在CPU发出的中断，是用来跟别的CPU进行交互的。

比如两个CPU要进行一些协同的工作，那其中一个CPU在处理完了内存当中的一部分数据，

它就可以通过发出中断请求来通知另一个CPU进行后续的工作。

因此在现代的个人计算机当中，可能已经找不到一个独立的中断控制器的芯片了，

但其实中断控制器的功能已经变得更为丰富，数量也变得更多了。

我们还是回到简单的情况，来看看这些来自外设的，可屏蔽中断的处理过程。

当外设有中断的需求， 那它就会通过中断控制器向CPU发出中断请求信号，

而CPU则会中断当前正在执行的程序， 向中断控制器发出中断响应信号，

然后中断控制器再会通过其他的信号线， 将对应外设的中断类型码发给CPU。而这个类型码，

其实也是在系统初始化时，通过写入中断控制器的I/O端口而设置的。

CPU在得到了中断类型码之后，后续的处理过程就和内部中断是一样的了。

我们快速地浏览一遍。CPU将相关的寄存器压栈，

然后清除IF和TF标志位，再取得对应的中断向量， 然后程序就会跳转到中断服务程序，在中断服务程序当中，

可以在适当的时机通过设置IF标志位，开放中断。

一旦开放了中断，就意味着在执行这个中断 服务程序的过程中，CPU还可能会响应其他外设发来的中断。

那在中断服务程序执行完之后，就会执行中断返回指令，

将返回地址等信息从堆栈中弹出，然后就可以回到刚才被中断的位置继续执行了。

如果在这个中断服务程序当中，开放中断后，外设又发来中断，

那其实CPU是会中断这个中断服务程序的执行，转而去响应这个新发生的中断。

> 这种情况就被称为“中断嵌套”。
>
> 当然，要想发生中断嵌套的情况， 必须要有比当前正在处理的中断优先级更高的中断请求，
>
> 那这时CPU就会去响应这个优先级更高的中断请求，
>
> 在执行完这个新的中断服务程序之后，再返回到刚才的中断服务程序当中继续执行。
>
> 我们再根据图示来看一看这个过程。
>
> 假设在一段主程序当中，开始是关闭了中断响应的。然后在这里打开了中断响应，
>
> 之后就发生了中断，于是CPU就会转向中断服务程序。
>
> 那么在这里CPU的硬件已经自动设置了IF标志位为0，屏蔽了外部的中断请求，
>
> 如果在这个中断服务程序的某个地方又打开了中断响应， 而且在这个过程中又有外设发起了更高优先级的中断请求，
>
> 那CPU又会去处理这个新的中断。
>
> 我们要注意，对于CPU来说，现在在执行的这个中断服务程序也就像是一个普通的程序，
>
> 所以在这个过程中如果发生了中断请求，
>
> 它依然会进行同样的那些处理步骤，比如压栈、关中断、保存现场等等，
>
> 然后根据取回的中断向量进入到第二层的中断服务程序。
>
> 那如果在这个中断服务程序当中没有开中断， 或者开了中断但是没有更高优先级的外设发起中断请求，
>
> 那一直执行到中断返回指令，CPU就会回到 刚才发生中断的地方继续执行，也就是第一层的中断服务程序。
>
> 如果之后没有再遇到中断请求， 就会一直执行到第一层中断服务程序的中断返回指令，
>
> 再返回到刚才主程序中断的地方，这就是一个简单的两层中断嵌套的过程。

> 现在我们已经知道了内部中断和外部中断是如何协同工作的，
>
> 而在现代的计算机系统当中大量地使用了中断来控制输入输出设备，
>
> 至于如何进行中断优先级的调配，什么时候能屏蔽中断，什么时候不能屏蔽中断，
>
> 这一些问题都是值得深入研究的。

## 906-直接存储器访问方式

> 我们还是来说快递的事。
>
> 有一天，你在这坐着运算，突然来电话了， 然后你接了一听，啊，
>
> 又有快递啦，啊什么？一千本书， 现在就得去拿，哎呀，好好好。
>
> 这个怎么办呢？这个，这搬一千本书又得浪费很多时间，
>
> 呃，好办，啊那我们就叫一个搬家公司， 跟他们说好，我们要从哪搬到哪谈好价钱，他们就去搬，
>
> 并且告诉他们搬完之后呐给我来一个电话，然后我去检查，这就可以了。
>
> 那么，虽然要多花一些钱，但是如果能够运算更重要的任务， 那还是值得的。这就是我们要说的DMA方式。

> 现在的计算机中有很多复杂的外设比如像显示器，网络，硬盘，

这些外设需要传输的数据量很大，而且对传输的速率也有很高的要求，

如果这些数据都要靠CPU一个一个来搬运的话，那恐怕就

难以应对了，所以这就需要用到DMA这种IO控制方式。

DMA就是直接存储器访问的简称。

那如果采用DMA方式进行I/O数据的传送在传送的过程中是

不需要CPU干预的，这个数据传送的工作是由一个专门的硬件电路控制，

可以直接将外设的数据传到存储器或者将存储器中的数据传到外设，

而这个专门的硬件控制电路就称为DMA控制器，简称为DMAC， 其实DMA控制器本身也是一个I/O口，

和其他I/O接口类似，它早期也是采用独立芯片的形式， 而现在通常是寄存在其他多功能的芯片当中。

那我们来看DMA控制器的基本工作步骤。

这是一个简化的系统，里面有一个CPU，一个存储器，

一个I/O接口，还有一个DMA控制器，它们通过系统总线连接在一起，

这里还增加了m和s这两种标记，m是master的缩写，

表示这个部件可以在系统总线上主动发起传输，

比如CPU就是这样的部件，它可以在系统总线上主动发起读写的传输。

而s是slave的缩写，它表示这个部件只能被动地接受来自系统总线的传输。

那存储器就是一个典型的只有slave接口的设备。

一般的I/O接口也是这样，只会接受来自CPU的访问， 而DMA控制器则是既有master接口又有slave接口。

那我们就要用这个DMA控制器进行一次外设到内存的传送，

我们可以把这个I/O接口看成是网卡，外面接着网线，

现在我们需要用DMA的方式接收一个网络包，并保存到存储器的某个区域，

那要完成这个操作，CPU首先需要设置DMA内部的配置寄存器，

那对于x86CPU，我们就要用out指令续写DMA 控制器当中的一些I/O端口，以配置好它的工作模式，

然后这个DMA控制器就处于空闲等待状态，而CPU也可以去执行其他的程序了。

那当外设送来数据到I/O接口的时候， I/O接口就会像DMA控制器发出DMA传送的申请，

这个申请需要通过一根额外的连线发出，

那DMA控制器收到I/O接口的申请之后，还会通过另一个连线响应这个申请，

然后DMA控制器就会通过它的master接口发起总线读传输，

而这个读传输的地址就是这个I/O接口当中的数据输入缓冲寄存器。

那这样数据就会从I/O接口被读到了DMA控置器当中，然后DMA控制器

会向存储器发起总线的写传输，将刚才读回的这个数据写到存储器的某个区域。

我们注意在有DMA控制器之前， 这个系统当中只有CPU可以发起总线传输，

而现在DMA控制器可以主动发起总线传输了。

那接下来DMA控制器会重复五和六这两个动作， 不断地从I/O接口中读出数据再写到存储器当中去，

那如果这时网络传输一直到收完一个网络包， 就次DMA传送才算完成，然后DMA控制器会返回到

第二步，等待I/O发起下一次DMA传送的申请。

那一次DMA传送的数据可能很多， 所花的时间也很长，但是在这段时间CPU一直可以在执行其他的程序，

这样就和数据传送的工作并行起来可以获得很好的系统性能。

但是CPU怎么知道DMA传送已经完成了呢， 那通常情况下DMA传送完成后，DMA控制器会

发出一个中断请求信号，通过中断控制器通知CPU， 那这个DMA控制器发出的中断，也是一个外部中断，

后面的处理过程就和其他I/O接口发出的中断是一样的，

只不过它对应的中断服务程序是让CPU来对这一次的DMA传输进行处理。

所以从这个步骤我们可以看出

DMA方式也不是完全不用CPU来干预，在DMA启动的时候CPU来进行配置

而传送完成后CPU还需要来进行处理。

那后续的处理根据任务的需要各有不同，而初始化的配置却大体是一样的。

在DMA传送开始前CPU要设置DMA控制器内部的寄存器，

一般至少要设置这么几项，既然等一会儿DMA控制器要进行数据的传送，

那就需要先设置好从哪里开始传，这也就是原地址的初始值。

还需要设置在传送的过程中这个原地址是递增还是递减，然后还需要

设置这些数据传输到哪里，也就是目的地址的初始值以及传送时的地址增减方式。

最后还要设置需要传送多少数据。

那么我们还是以刚才提到的外设到内存的传送为例。

那CPU在初始化配置时，会将某个源地址设为I/O端口， 比如就是一个网络控制器的数据输入寄存器端口，

而且传送时这个源地址是不变的。

因为每一次DMA控制器都是从同一个I/O端口读出数据， 而目的地址要设置为存储器的某个地址，

而且传送时是递增的，这样DMA控制器每次从I/O接口当中读出一个数 就把这个数写到存储器当中去，

下一次再读出一个数再写到存储器当中时地址就应该递增， 这样才不会覆盖刚才传过来的这个数。

第三要设置待传送数据的长度，那如果是从外设接收数据，

CPU在配置时可能不知道究竟这个数据有多长，

那就可以不设置这个参数。最后根据I/O接口的控制信号来判断是否传输完成。

那如果是从存储器发送一组数据到I/O接口， 这个时候CPU在初始化时就是知道究竟要发送多长的数据。

那就需要设置这个待传送数据的长度的这个参数。

那需要强调的是， 这些参数都是DMA控制器内部的寄存器，一般各自都有一个I/O端口的地址，

那么编程时通过out指令让CPU去写这些I/O端口，从而配置好了源地址，目的地址

和待传送数据的长度等信息。DMA控制器在运行的过程中就不需要再靠CPU

执行程序来控制，而是直接查看内部的源地址寄存器，就把对应的地址发到系统总线上，

就从I/O接口读回了相应的数据，然后再查看自己的目的地址寄存器，

把这个地址和刚才读回来的数据一起发到系统总线上，这样就写入到了存储器当中去。

而每读写一次， 就在内部累计已经传送的数据的长度，并和这个待传送数据的长度的寄存器的内容进行比较，

如果相等则意味着传输已经完成，如果还不等则继续传输。

所以CPU在一次配置完之后，后续的工作都由DMA控制器的内部硬件自动完成，

不再需要CPU的干预了。这就是所谓直接存储器访问的含义。

那 那在最早的个人计算机当中是没有DMA控制器的，

那后来为了提高输入输出的效率，就增加了独立的 DMA控制器的芯片。例如刚才提到的8237，

那通过CPU设置DMA控制器当中的不同的地址就可以为不同的I/O接口提供DMA- 的服务。

但是随着计算机的发展有一些I/O接口的速度越来越快， 对DMA传输的要求也越来越高。

那多个I/O接口共享一个独立的DMA控制器的方式可能就没有办法满足部分I/O接口的- 需求了。

这时就出现了自带DMA控制器的I/O接口，

那这样的I/O接口内部带有一个专属的DMA控制器，只为这个I/O接口

提供服务，那这个I/O接口现在也有了master的总线接口。

那在显卡、网卡、硬盘控制器这些 对传输力要求很高的I/O接口中一般都会自带DMA控制器，

那在系统初始化时，CPU要配置好各个DMA控制器，

然后外设有传输需求时这些DMA控制器就可以自动地开始工作了。

那如果我们把这个独立的DMA控制器比作一个搬家公司，

CPU就请这个搬家公司来完成I/O接口和存储器之间的数据搬运工作。

而有些部门的搬运工作量非常大， 实际上需要一个搬家公司全时地为他们服务才能够满足需求。

于是他就在部门内部自己组建了一个搬运队。

那这样一旦有数据传送的需求就可以马上开始工作，而不需要去申请外部的这个DMA控- 制器了。

传输的效率自然会大大提升，而且不同的I/O接口有不同的传输的特点。

比如显示、网络、硬盘传输的行为肯定有不同的特征，

而内线的DMA控制器就可以根据I/O接口的特点进行定制，从而更加

高效地完成传输。那在现代的计算系统当中，

> 大部分对数据传输率有比较高要求的设备都会自带DMA控制器，
>
> 而其他对数据传输率要求比较低的设备则可以共享系统中独立的DMA控制器。
>
> 另外这个独立的DMA控制器一般还会提供从内存到内存传送的服务。
>
> 那当我们编程时需要将内存中的一大块数据复制到内存的另一个区域的时候，
>
> 虽然不涉及输入输出，但是也可以享受到DMA带来了好处。
>
> 那当然也不是所有的输入输出设备都需要使用DMA的方式。
>
> 毕竟增加一个DMA控制器需要增加制造的成本， 而且CPU来配置DMA控制器以及进行后续的处理
>
> 还是要靠执行程序来完成的，也都需要花时间。
>
> 如果要传输的数据量很小，性能反而会变差了。

> 那现在用DMA方式的这些设备 不见得是这些工作CPU不愿意干，有可能是根本干不了。
>
> 比如说我们快递要送来的不是一千本书而是一万本书，而且非得一个小时内办完，
>
> 那你能干得了吗？非得叫搬家公司不可了，对不对？ 但是不是说所有的事情都用DMA方式都能好好的解决的。
>
> 比如送来的不是一千本一万本书，而就是一本书， 那你也去叫一个搬家公司把这本书给你运来？那样就得不偿失了。
>
> 而且有时候你可能送来的根本不是书，
>
> 而是你的午饭，对吧，订了一份盒饭，难道你也叫一个搬家公司来 给你收这个盒饭？那他们可能过了两个小时才来，
>
> 然后用他们规范的方法把你这顿饭给打包装箱送到他们的大货车， 然后运到你的楼下，再拆包，再给你送上来，
>
> 能赶上吃晚饭就不错了。