第六章

AVR 微控制器

一个真正有用的引擎……

— W.V. Awdrey

在这一章，我们将要介绍ATMEL公司的AVR处理器。与PIC处理器一样，这一系列的处理器都是集成到单个芯片的独立的计算机。对于任何一种小型的控制或监视应用来说，AVR处理器都是理想的选择。它们包含有一组内置的外设，还可以在片外扩充附加功能。

与PIC一样，AVR处理器也采用RISC技术。根据我个人的经验，在这两种结构中，AVR的指令执行速度最快，可是为它们编写代码就不知道哪个更简单了。PIC与AVR一样使用的都是单周期执行的指令。不过我发现AVR有一个更加通用的内部结构，因而实际上使用它可以获得更高的吞吐量。如果让我给一个小规模嵌入式应用选择处理器的话，AVR将是我的首选。

本章中，我们将通过设计一个小型的基于AVR的计算机系统来学习开发计算机硬件的基础知识。这个小型的计算机所使用的处理器为AVR系列的ATtiny15处理芯片。我们还会学习如何将代码下载到基于AVR的计算机中，以及如何在电路内再编程。之后，我们将会继续介绍一些规模更大、功能更强的AVR处理器。

本章的后半部分主要讲述了如何使用地址总线、数据总线和控制总线来完成处理器与内存以及外设间的互连。这种接口方式也为大多数的处理器所采用，因而对AVR而言，可用的存储设备和外设的种类是很多的，是可以与总线接口共同使用的。所以知道如何连接那些基于总线的设备，将能极大地扩充嵌入式计算机的应用。系统可以连接的设备包括RAM、ROM（或闪存）、串行控制器、并行端口、磁盘控制器、声卡、网卡以及其他设备的主机。

大多数小型的微控制器都是完全独立的整体，并不向外“牵引”出总线。本章所要学习的ATMEL AT90S8515处理器是AVR系列产品中惟一一种可以允许访问CPU总线的处理器。不过，我们首先先来大体上看一下AVR的体系结构。

AVR处理器的体系结构

AVR处理器在挪威开发，并由ATMEL公司负责生产。这是一款采用哈佛结构的RISC处理器，其设计的主要目的是加快指令的执行速度并减少系统的功耗。它有32个8位的通用寄存器（从`r0`～`r31`），其中六个可以复合成3个16位的索引寄存器（`X`，`Y`，`Z`），如图6-1所示。它总共有118条指令，故而可以提供多功能编程环境。

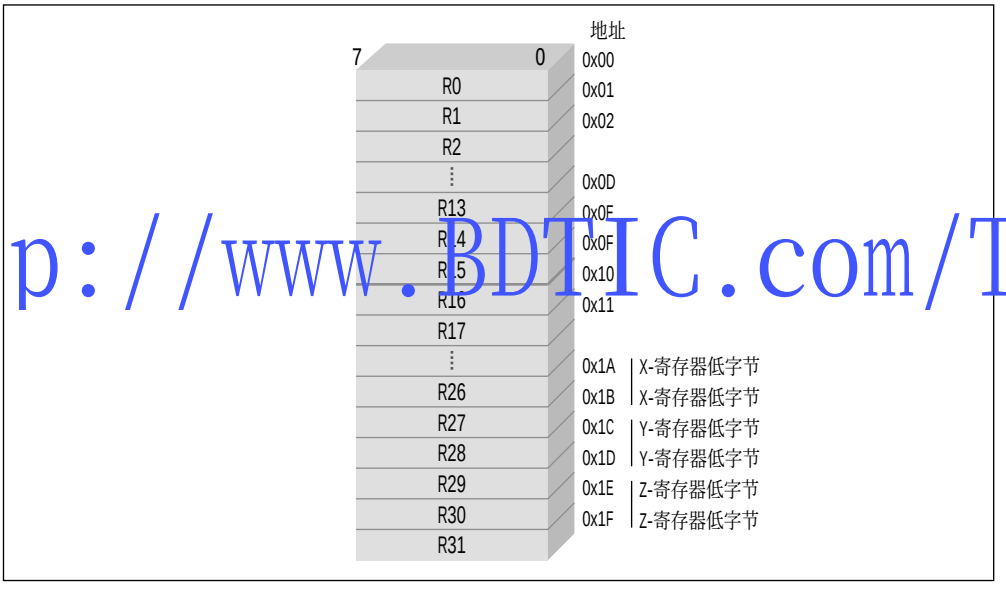

图6-1：AVR寄存器

在大多数AVR处理器中，堆栈存在于通用内存空间中，因此可以由指令直接操作，而且堆栈大小也和在PIC中一样没有限制。

AVR拥有独立的程序空间和数据空间，最大可支持8M的地址空间。AT90S8515 AVR处理器的内存映射如图6-2所示。

一直让ATMEL公司引以为荣的是AVR处理器的吞吐量。该公司给出了如下一段示例的C代码，这段代码可以在多种不同的处理器上编译运行：

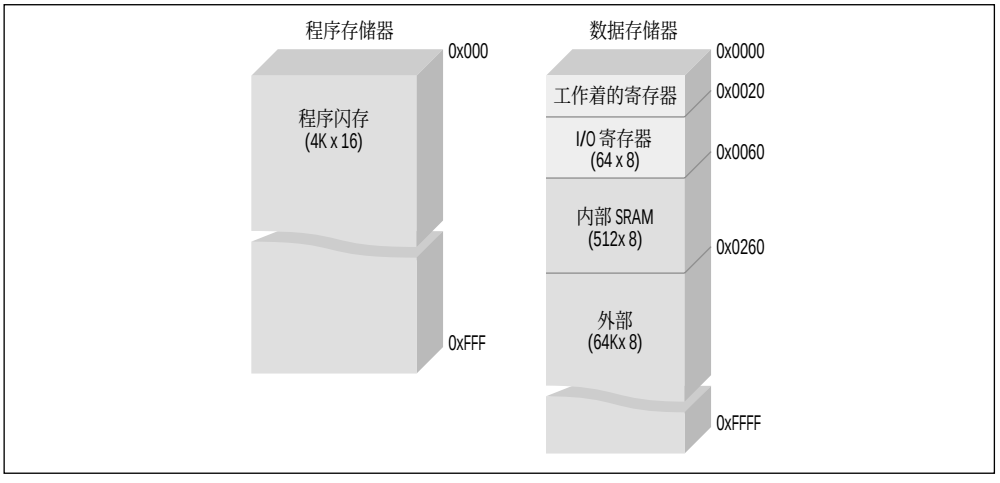

图 6-2：ATMEL AT90S8515 内存映射

```
int max(int *array)
{
  char a;
  int maximum = -32768;
  for (a = 0; a < 16; a++)
    if (array[a] > maximum)
       maximum = array[a];
  return (maximum);
}
```


程序的运行结果很有趣（见表 6-1）。

表 6-1：ATMEL 的几种处理器执行速度与效率的比较

处理器	编译代码的大小	执行时间（周期）
AVR	46	335
8051	112	9384
PIC16C74	87	2492
68HC11	57	5244

从上表我们可以看到，以相同时钟频率运行时，AVR 的执行速度是 PIC16 的 7 倍，是 68HC11 的 15 倍，是 8051 的 28 倍之多！换句话说，如果 8051 要达到频率为 8MHz 的 AVR 的速度，那么它的时钟频率至少得是 224MHz。现在 ATMEL 公司并没有说明他们在测试中使用的是哪一种编译器，于是对于选定的源代码的途径不同，结果必然有差异。但从我个人的使用经验来看，AVR 确实能使你的程序执行得更快，而且代码也更紧凑。对于大多数小规模的应用，AVR 是我的首选，同样它的结构也是本章我们要讨论的重

点。AVR的速度快过同级别的PIC处理器的局面可能会随着新的dsPIC处理器的出现而有所改变。这种处理器有着出色的体系结构，可以提供极其强大的处理器能力。

AVR 体系结构的处理器可以分为三个基本系列。最原始的一个系列是 AT90xxx。对于复杂的应用，可以使用 ATmega 系列，而 ATtiny 系列则用于小规模应用中。ATMEL 也生产大规模的现场可编程门阵列（Field-Programmable Gate Arrays，FPGA），它包括了一个 AVR 核和上千的可编程逻辑门。

为了软件开发，还为 AVR 提供了 gcc 端口，同时，ATMEL 公司还提供了一个汇编器、一个仿真器以及一套软件用以将程序下载到处理器中。ATMEL 公司的软件可以在他们的网站上免费下载。低价格的ATMEL开发系统是进行AVR开发的很好的工具，它为你开始 AVR 开发提供所需的软件和工具。

下面我们将要讨论的 AVR 处理器是小型 ATtiny15、AT90S8535/AT90S4434 以及 AT90S8515。

ATtiny15 处理器

对很多简单的数字应用来说，小型微处理器较离散逻辑电路是一个更好的选择，因为它能执行软件，所以它在执行某些任务时可以降低硬件复杂度。下面我将向你展示，用一片 ATMEL ATtiny15 AVR处理器构建一个集成到大型系统中的小型的嵌入式计算机是多么简单。ATtiny15 有 512 字的闪存用以存储程序，但没有 RAM。（想像一下当接下来你必须把一些 100MB 的应用程序装入 PC 机时。）与其他大的 AVR 同类产品不同，ATtiny15 处理器完全依靠它的 32 个寄存器来存储中间变量。

由于没有 RAM 来配置堆栈空间，因而 ATtiny15 采用了专用硬件堆栈，但堆栈深度只有 3 个输入项，而且为子程序调用和中断所共享。（可以想像如果嵌套超过三层，结果将会如何!）程序计数器为 9 位（用以寻址 512 个字的程序存储空间）；因而堆栈也是 9 位。和较大的 AVR 处理器不同之处还有：只有两个寄存器（r30 和 r31）可以联合组成一个 16 位的变址寄存器（称为 Z）。

处理器还包括 64 字节的 EEPROM（用于存放系统参数），高达 5 个的通用 I/O 引脚，8 个内部和外部中断源，两个 8 位定时器 / 计数器，一个 4 通道 10 位 A/D 转换器以及一个模拟比较器，而且处理器还能进行电路内的再编程。它的封装带 8 个小引脚，从中可获得高达 8 MIPS 的吞吐量。对于它的大部分特性我们本章先不介绍，在后面介绍 I/O 时会一并讨论。目前我们将专注于如何用这样一个元件来完成简单的数字控制。

使用一个如ATtiny15那样的小型微控制器非常简单，因为它只需很少的外部支持就可以实现自身的操作。图 6-3 显示了 ATtiny15 是如何简单。

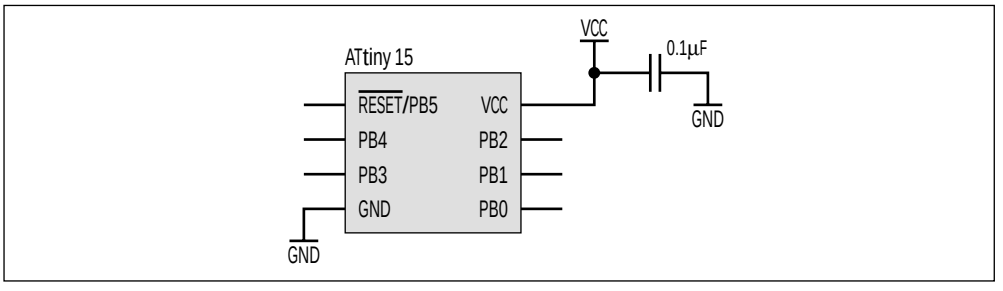

图 6-3：最简单的计算机

让我们快速地看一下ATtiny15的设计。**VCC**用以连接电源，电源的电压可以介于2. 7V与5.5V 之间。**VCC** 使用一个 0.1µF的电容接地，用以退耦。五个引脚 **PB0** 到 **PB4** 可以用作数字的输入或输出。它们可以用于读取开关的状态，打开或关闭外部设备，产生信号波形用以控制小型电机，甚至还可以作为一些简单外部芯片的接口。这些数字 I/O 线（**PB0** 到 **PB4**）还可以用来连接到任何处理器负责监控或控制的设备上。稍后我们将在本章给出几个相关的例子。

最后，还有一个输入端（**RESET**）没有任何连接，这对其他任何处理器来说是不可思议的。许多处理器都需要一个上电复位（Power-On Reset，POR）电路，以便在程序执行时使系统初始化。还有一些处理器具有一个内部上电复位电路，因而不需要外部电路支持，但这样的处理器仍会有一个复位输入，使用户或是外部系统可以手动复位。一般情况下复位输入端仍需要一个上拉电阻，以使其处于非激活状态。但ATtiny15处理器没有这个需要，因为它内置了一个上电复位和一个上拉电阻。因此和其他大部分（也许是全部）的处理器不同，在ATtiny15上**RESET**端是处于非连接状态的。事实上，如果ATtiny15不需要一个外部复位电路，那么它的**RESET**引脚也可以作为通用输入端（**PB5**）来使用。但必须注意的是：**RESET/PB5** 没有通常的输入电压保护机制（用来防止高于正常值的电压输入），因为在程序烧录器（program burner）下载软件时，**PB5** 端的电压将会升至 +12V。因此，当你在使用 **PB5** 引脚时，一定要注意不要使输入电压超出 VCC 1V以上。如果不能做到这一点，处理器将会处于软件下载模式，从而绝对会使你的嵌入式计算机完全崩溃。

AVR 处理器（PIC 也是如此）还包括一个内部电路，叫作电压不足检测器（brownout detector，BOD）。BOD能检测出处理器电源供应的微小波动，这个小小的波动有时会影响指令的执行。一旦检测到这种波动，BOD就产生复位指令，使计算机重启。ATtiny15

处理器还有另外一个复位发生装置，叫作看门狗（watchdog），用以在软件崩溃时重新启动计算机。看门狗是一个小的定时器，其作用是当超时发生时自动重启处理器。正常操作时，软件有规律地重置看门狗。它们的工作原则是“ 在你复位我之前，我就复位你 ”。如果软件崩溃，而看门狗又未清零，这时超时就发生了，计算机就被重启。带有看门狗的处理器能使软件有能力区别上电复位和看门狗复位。对于看门狗复位，就可以从内存中恢复系统的状态，使程序能够继续执行，而无需完全地重新初始化。

ATtiny15 设计的另一个独特之处，就是它没有时钟电路。ATtiny15 芯片可以有一个外部晶体电路。（ATtiny15的**PB3**、**PB4**引脚可以分别当作晶体的输入端**XTAL1**和**XTAL2**）但在我们的设计中没有，也没有必要使用晶体，因为处理器自身有一个完备的频率为1.6MHz的内部振荡器（本例中为RC 振荡器），所以就没有必要为时钟再添加额外一个外部元件。但问题在于振荡器是不太稳定的，它的频率可能会随着温度的改变而变化。（ATtiny15 振荡器的工作频率可在 800kHz～1.6MHz 之间变化!）通常情况下，RC 振荡器并不适合那些对时序要求严格的应用，对于那些应用就需要用外部晶体来代替。但如果你的ATtiny15只是做一些简单的控制功能，时序可能也不是个问题，你就可以使用内部 RC 振荡器，以降低硬件复杂度。ATMEL 公司还为 ATtiny15 提供了一个 8 位的校准寄存器，所以你能够随时调整内部振荡器，使之精确度更高。

到此我们已经看了ATtiny15机器的基本设计，总的来说它是一种低价格、多功能的小型计算机，也并不需要对核心设计再做任何工作。设计时需要作的惟一努力就是确保计算机能够同与之接口相连的I/O设备很好地工作。如果你不打算用电池组给你的系统供电，那么电容是可以选择的。从以上的讨论中，我们可以看出在嵌入式系统的设计中，只有处理器才是不可缺少的（或许在此之前你还以为设计计算机硬件是多么困难的一件事呢）。

这就是基本的 AVR 计算机硬件，只需最少量的元件。我们马上就会看到如何为你的硬件下载软件。

从上面所介绍的 ATtiny15 系统的基本情况可以看出，它与同类型的 PIC12C508 系统没有太大的不同。它们间的真正区别在于它们的内部结构（和指令集）、对操作电压的敏感度以及接口功能。正如你所看到的一样，把这类小元件加到你的嵌入式系统并不是一件很困难的工作。

到目前为止，我们介绍的计算机都没有连接任何外部设备。下面我们就开始一些简单的工作，为AVR添加一个LED。这种基本的连接技术也同样适用于所有带有可编程I/O线的微控制器。

添加一个状态 LED

当有电流通过时，发光二极管 LED 便会发光。由于二极管具有单向导电性，所以只有当电流从阳极（正极）流向阴极（负极）时，二极管才能工作。在 LED 的电路符号中，阴极用水平横线表示，阳极用三角形表示。

图 6-4 所示的是一个状态 LED 的电路图，它使用微控制器上的一根 I/O 线来控制 LED 的开关。我们将会看到：当 I/O 线输入低电平时，LED 启动；输入高电平时，LED 闭合。图中电阻 R 的作用是防止太强的电流流入 I/O 线路，稍后我们还将看到同样的情况。

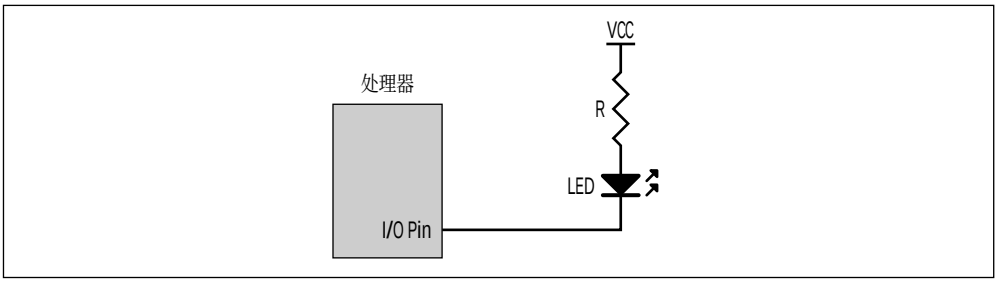

图 6-4：状态发光二极管 LED

当有电流通过时（此时 LED 发光），LED 有一个正向电压降，表明阴极的电压要低于阳极。不同的 LED 电压降的幅度有所不同，你可以通过查找相应的技术手册来获得你所使用 LED 的电压降的数值。

当处理器的工作电压为 3.3V 时，ATtiny15 的 I/O 引脚输出的低压为 0.6V；当处理器的工作电压为 3V 时，输出的低压则为 0.5V。假定我们使用的是电压为 5V（为了本例计算方便）的电源 **VCC**，LED 的正向电压降为 1.6V。现在，要将 LED 阴极输出电压置为 0.6V，意味着电源 **VCC**（5V）与 LED 阴极间的电压为 4.4V，如果 LED 的电压降为 1.6V，那么电阻上的电压降应该是 2.8V（由 `5V - 1.6V - 0.6V = 2.8V` 可得）。

从技术手册可知，如果处理器的工作电压为 5V，则 AVR 的数字 I/O 引脚上的电流可达 20mA。显然，我们必须限制电流的流量，引入电阻就是出于这个目的。如果电阻的电压降为 2.8V（上面已计算过），流入的电流为 20mA，那么由欧姆定律可知，我们所需的电阻的大小为：

```
R = V / I
  = 2.8 / 20mA
  = 140
```

我们就选择与这个值最接近的可用电阻，即 150 。（使用 150 电阻将会使实际电流为 18.6mA，这正合适。）

警告：当工作电压为5V时，AVR的每个引脚可以接受的电流为20mA。不过随着工作电压的减小，可接受的电流也随之减少。当工作电压为2. 7V时，可接受的电流就只有10mA。所以，在使用前仔细阅读技术手册是很有必要的。

下一个问题是：电阻会耗掉多少电能？换句话说，当电压降为2.8V时，电阻会消耗掉多少能量？这个问题很关键：如果通过电阻的电流过大，那么电阻就有可能被烧坏。所以，我们在选择电阻时要选择一个额定功率大于所需值的电阻。功率可由电压和电流的乘积计算：

```
P = V * I
  = 2.8V * 20mA
  = 0.056 Watts = 56mW
```

这个值是微不足道的，所以我们可以选择电阻值为150 ，功率为0.0625W的电阻（因为0.0625W是目前可用电阻中额定功率的最小值了）。

那么，当I/O线由过高电压驱动时，会发生什么事情呢？此时（工作电压为5V）AVR的I/O引脚的输出电压至少为4.3V，则此时**VCC**与LED阴极间的电压降只能是0. 7V（或更少）。而LED的正向电压降（即阈值电压）为1.6V，故没有足够的电压使其导通。

在这种情况下，只需使用简单的处理器的数字输出，我们就可以控制LED的闭合与导通了。我们还知道了如何计算电压和电流。设计的每一方面都非常重要。忽视它你将会设计出很糟糕的机器。而且更可怕的是，这样也许会烧毁芯片，使空气中弥漫着晶体硅焦臭的味道。

在下面我们要看到的是如何使用AVR的数字输出端去控制LED。这种方法对于那些工作电流低于20mA的设备同样有效。事实上，小功率的元件（如传感器）都可以使用AVR的输出端来对电源进行直接控制，就像我们直接控制LED的电源那样。在那些使用电池供电的应用中，这也是一个很有用的技术，因为它可以降低系统的总功耗。

开关模拟信号

我们还可以使用处理器的数字I/O线路来控制系统内的模拟信号流。比如，我们的嵌入式计算机有可能被集成到一个音频系统中，用以在不同的音频源之间的切换。为了完成这一任务，我们还需要为每一信道准备一个模拟开关，如MAX4626。这种微小的元件（以表面贴片技术所设计的MAX4626只有一粒米那么大）需要单独的工作电压（其值介于1.8V与5.5V之间）。它还有一个内置的过载保护电路，以免设备在短路时被损坏。图6-5所示的是MAX4626与ATtiny15AVR的连接电路图。AVR输出端（**PB2**）为高电平

时，MAX4626 将处于工作状态，并连通**NO**和**COM**端；将**PB2**置为低电平会使连接中断。通过这种方式，MAX4626 便可以在软件的控制之下，完成输入端与输出端的连接。

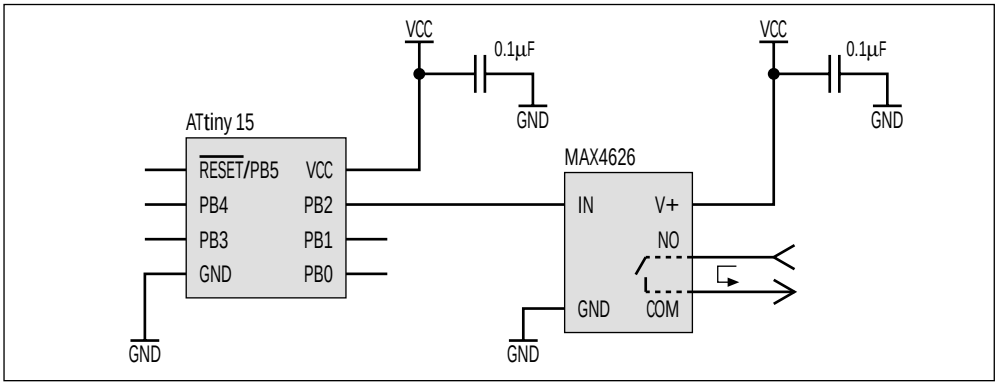

图 6-5：一个模拟信号的切换

问题是 MAX4626 与 AVR 兼容吗？当工作电压为 5V 时，MAX4626 的输入端（引脚 4，**IN**）需要的逻辑低压输入要小于 0.8V，逻辑高压输入至少为 2.4V；而 AVR 的逻辑低压输出为 0.6V 或更低，逻辑高压输出最小值为 4.3V。因而，AVR 的数字输出端电压完全符合 MAX4626 的要求。再来看电流，MAX4626 接收或是发送的电流只有 1 µA，可以对 AVR 来说，这不是什么问题。

即使 MAX4626 不合用的话，MAXIM 公司或其他的制造商也还生产了多种不同特性的类似产品，相信总会有适合你需要的产品。

图 6-6 中的原理图给出了一个连接**PB3**的按钮，此处**PB3**作为数字输入端。不过，对于这个简单的输入电路，有两个有趣的问题要注意。第一，并没有外部上拉电阻连接**PB3**。而通常情况下，这种电路是需要一个上拉电阻在当按钮放开时（即未被按下）将输入置为一个已知状态。除了按钮闭合或输入端直接接地时，上拉电阻都将输入端置为高电平。本例中没有上拉电阻的原因是：在 AVR 内部已包含有一个上拉电阻，并且可以由软件控制是否选择上拉电阻的功能。

第二个需要注意的有趣的事情是，在按钮与输入端之间没有连接去抖动电路。当按下任何一种机械开关（包括键盘的按键）时，都如同一个小的电感。其结果是会在信号线上产生一个迅速衰减的振荡。如图 6-7 所示。

因此，可以看到当按钮被按下时，状态并不仅仅改变一次，却像是用户在快速连续地敲击按钮，于是负责响应这些输入变化的软件会记录多次脉冲信号，而并不是像用户所希

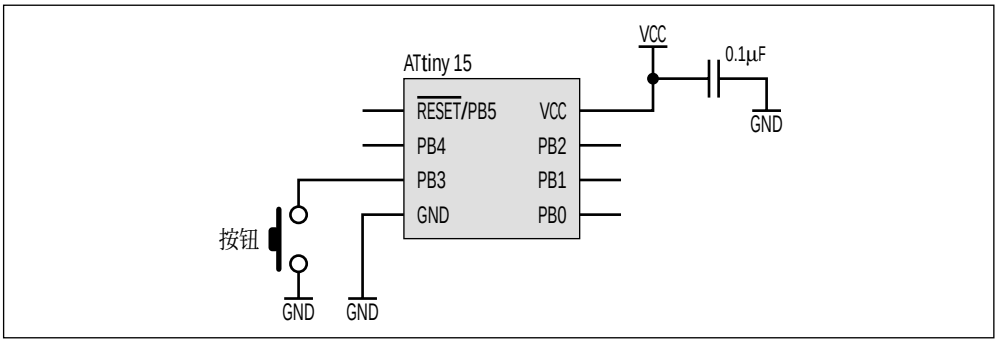

图 6-6：按钮输入端电路

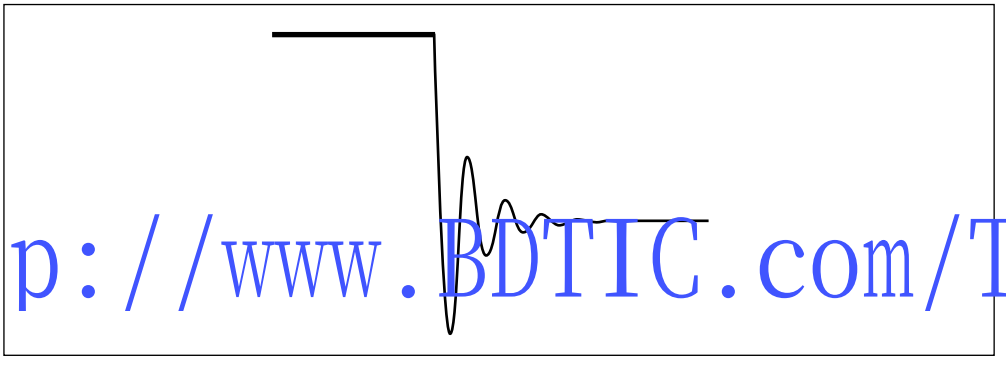

图 6-7：信号的抖动

望的单一信号。所以，从信号中消除那些瞬时的过渡状态是很重要的，这个过程被称作去抖动（debouncing）。

目前，有很多种电路可用来消除这些振荡信号。但问题是，你并不是非要用这些电路不可！当一个用户按下按钮时，他通常不立即放开，而是持续至少半秒钟，直至振荡信号完全消失。所以这一问题也可以用软件的方法来解决。当软件第一次记录的输入电平信号为低，而且如果隔几百微秒后采样所获得的信号（当然，可能不止采样一次）也为低时，那么软件就可以断定这是一次有效的按钮输入。之后，软件会“重置”输入端，以等待下一次按键。但如果按钮与中断电路或者复位端相连的话，那么去抖动硬件就很有必要了。

到目前为止，我们已经学习了如何用 AVR 控制数字输出，以及如何处理一些简单的数字输入。在观察前两个电路图时，喜欢思考的人也许会问：难道处理器是必不可少的吗？将按钮直接连到MAX4626的输入端不也是可行的吗？那么处理器在这里到底起什

么作用呢？现在我来解释一下。我们已经看到处理器的一个作用：即取代输入端的去抖动电路。处理器另外的作用是：由于处理器有内置的存储器，能够执行软件，所以可以用来记录系统的状态（和模式）；监控多种信号的输入；能为输出端提供复杂的控制序列。简而言之，使用微处理器可以减小系统的硬件复杂度，增强系统的功能。所以，它们是非常有用的工具。通过使用更先进的处理器、更多种类的 I/O，嵌入式计算机能够提供的应用和功能也就越来越强大！

代码的下载

AVR处理器使用内部闪存来存储程序，还可以实现电路内编程，如果使用的是即插即用的存储元件（Socketed Component），则也可以在电路外编程。AVR 处理器通过芯片上的串行外部接口（Serial Peripheral Interface，SPI）来实现再编程。（SPI 接口将在第九章详细讨论。）甚至像 ATtiny15 这样的处理器自身也不包含 SPI 接口，在进行重复编程时仍然需要配合 SPI 端口使用。当处于编程状态时，芯片的引脚 **PB0**、**PB1** 以及 **PB2** 分别连接到 SPI 的功能端（**MOSI**、**MISO** 和 **SCK**）。

VCC 电压可以由用于下载代码的外部编程器提供。编程时，**VCC** 必须是 5V。如果嵌入式系统内部电源能提供5V的电压，那么编程器上的 **VCC** 端可以不用。但如果嵌入式系统电压不是 5V，则编程器的 **VCC** 端一定要用，同时要禁用系统的内部电源。复位端 **RESET** 的作用也很重要。当 **RESET** 启动时（置为低电平），系统开始编程，此时，处理器内部的 CPU 被禁用，因而可以访问内部存储器。引脚 **PB0**、**PB1** 和 **PB2** 的功能也随之改变以连接 SPI 接口。然后，开发软件通过 SPI 接口将某一代码串发送出去，用以“解锁”程序存储器和启动下载软件。一旦程序下载开始，写指令序列就会被执行，软件（以及相关设置）就被逐字节地下载。ATMEL 公司的软件可以完成这一整个过程，所以正常情况下，你不必关心具体的实现细节。但如果你需要“手动地”完成这一过程，比如从其他类型的主机进行操作，则可以从 ATMEL 的数据手册上查询相关的实现协议。

ATMEL 开发系统还带有一个专用的适配器电缆，它可以插入到开发面板上，这样你就可以使用PC机的并行端口对微处理器进行再编程了。通过在电路内安装合适的连接器，你也可以在嵌入式系统中使用同样的编程电缆。根据开发面板的特点，你可以从两种可能的连接器中选择一种实现电路内编程。图 6-8 所示的是它们的引脚图。**VTG** 用于为目标系统提供输入电压。但如果目标系统有自己的电源且输出电压可以达到编程的要求（+5V），那么 **VTG** 引脚也可以不用。在某些 ATMEL 应用说明中，引脚 3 通常标为没有连接。不过，在一些开发系统中引脚3通常被用来驱动LED（用以表明程序正在下载中）。

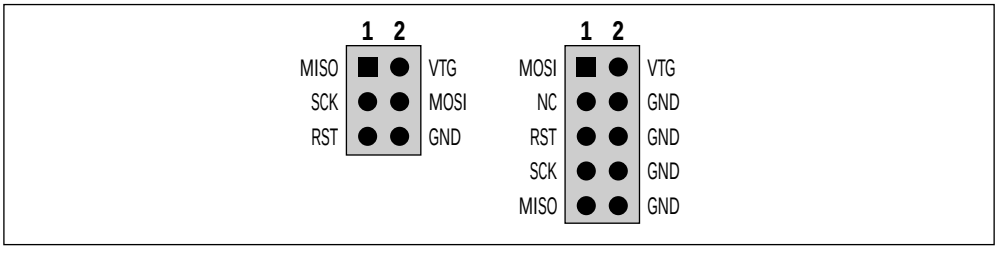

图 6-8：电路内编程连接器

支持电路内编程的电路图如图 6-9 所示。需要指出的是，连接器的 **MOSI** 端将连接到处理器的 **MISO** 端，同样，处理器的 **MOSI** 端连接到连接器的 **MISO** 端。这样是因为，在编程过程中处理器一直处于从机（slave）地位而非主机（master）地位。

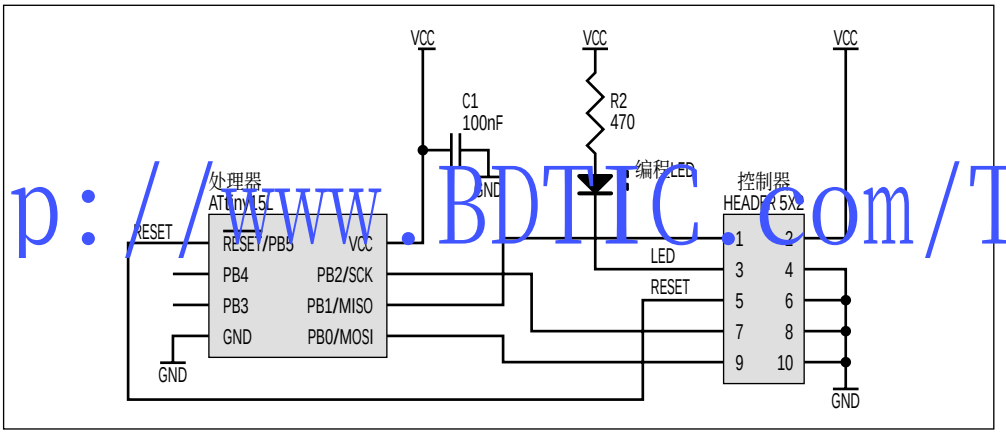

图 6-9：电路内编程

连接器的类型可以看作IDC的头部，电缆可以提供编程所需的所有信号，其中也包括驱动编程LED指示灯信号的指令。当不用于编程时，连接器还可以作为嵌入式计算机的一个简单的 I/O 连接器，用来访问数字信号。因而一种硬件可以起到两个作用。

然而，有一点非常重要：如果你使用 **PB0**、**PB1** 或 **PB2** 去连接了计算机内部的其他元件，那么必须要小心，在编程时不要对它们造成不利的影响。比如，一个带有MAX4626的电路使用了 PB2 引脚作为控制输入端。在编程过程中，**PB2** 会作为一个时钟信号（**SCK**）。因而，在代码下载到处理器的过程中，MAX4626 会一直在开与关之间来回快速地切换。如果此时MAX4626在控制某种设备，那么该设备同样会不停地打开、闭合，这就有可能带来灾难性的后果。从另一方面来说，如果你的系统中还有其他的元件，那么在编程的过程中，一定不要试图将其连接到 **PB0**、**PB1** 或 **PB2** 引脚上。如果这样做

了，轻则下载中断，重则会损坏嵌入式系统和编程器！所以在编程时，充分考虑电路内编程对系统内其他组件的影响是极其重要的。

那么我们该如何解决这一问题呢？当然，我们可以使用 **PB3** 去控制MAX4626，因为它不参与编程。不过如果我们一定要使用 **PB2** 的话，那么可以在处理器与MAX4626之间加入一个缓冲器（可由 **$\overline{\text{RESET}}$** 控制）。当 **$\overline{\text{RESET}}$** 为低电平时（处于编程状态），缓冲器失效，则MAX4626被隔离。还有一种方法就是使用采用DIP封装技术的同种类型的处理器。它通过插槽安装，需要重新编程时，可将其物理地取下。如果你使用的处理器采用的是表面贴片技术，它可以被安装到一个小的PCB面板上，而该PCB面板又可以插入到嵌入式计算机中［就像台式计算机上的SIMM（Single-in-line Memory Module，单列直插式内存组件）内存条那样］，那么在编程时也可以将其取下。方法远不止这些，哪种方法最好需要根据应用决定。

有些AVR处理器（不包括ATtiny15）具有使用SPM（Store Program Memory）指令来修改自身程序存储器的功能。使用这样的处理器，你的软件可以通过处理器的串行端口来下载新的代码并将其写入程序存储器。为了完成这一过程，你的处理器需要预编程引导载入程序。正常情况下，在初始化过程中，需要给所有的处理器都加载引导载入程序（以及你的应用软件的1.0版本）。此后自我编程（self-programming）机制才可以用来更新你的应用软件。为方便起见，程序存储器被分为两个独立的区：引导区和应用程序区。内存空间被划分为128或256字节大小的页（具体大小由处理器类型决定）。一次只对内存的一个页进行擦除和再编程。在编程过程中，Z寄存器作为指针，用以存放页地址，而 `r1` 和 `r0` 寄存器则一起用来存放待编程的数据字。ATMEL的应用参考（AVR109: 自编程），可以从公司的网站上查阅，它给出了引导载入程序的源代码示例，并给予了详细的说明。

无论你使用的是何种类型的处理器，芯片的制造商都会给出相应的技术参数，用以说明如何将代码下载到处理器中。

更强大的AVR处理器

到目前为止，我们所介绍的都只是小型的AVR处理器，它们的功能都很有限。本书的第三部分将要介绍的是在嵌入式系统中常见的各种输入输出形式。因此我们需要功能更多的处理器。我们已经详细介绍了ATtiny15处理器，现在将要看到更为复杂的处理器。在第七章开始详细介绍I/O特性之前，我们先来了解一下这些处理器，并且学习如何将它们添加到设计中。

首先要介绍的是 ATMEL AT90S8535，这是一款中型的 AVR 处理器，有大量内置的 I/O。除了数字 I/O 端口，它有各种各样的端口，比如：串行端口、串行外设端口、SPI、模拟输入端、定时器以及计数器。在后面我们将选择几种予以详细地介绍，现在还是先把注意力放到处理器本身上来吧。

此处理器有 512 字节的内部 RAM 和 8K 的闪存用来存储程序。它的一个更小的姊妹产品 AT90S4434，除程序存储空间更小（4K 的存储程序空间以及 256 字节大小的 RAM）外，其他方面完全相同。但如果只从电器特性的角度来看，AT90S8535 和 AT90S4434 处理器没有什么差别。

图 6-10 所示为基于 AT90S8535 芯片的计算机的电路图，其中没有任何额外的设备。可以看到，除了有更多的引脚之外，别的方面它与 ATtiny15 没有什么大的不同。在 **$\overline{\text{RESET}}$** 端有一个 10K 的外部上拉电阻。处理器还有一个外部晶体（X1），这需要有两个小的旁路电容 C1 和 C2。处理器还有 4 个电源引脚，每个引脚都连接有一个 100nF 的陶瓷电容用以去耦合。这些电源输入端中有一个 **AVCC**，是用来给芯片内的模拟部件供电的，一个100 的电阻R2将其与数字电源输入端隔离。它为模拟部件与开关噪音之间提供了一个小的屏障，而这些开关噪音也许来自数字电路。和 ATtiny15 芯片一样，其余的引脚都作为通用目的的 I/O 端口。不过与 ATtiny15 的不用之处在于这些引脚都具有两种不同的功能。它们可以在软件的控制下重新设置，用以实现另一种 I/O 功能。处理器的技术手册上给出了如何在软件的控制下去设置这些功能的全部信息。

你会发现对大部分的 AVR 处理器而言，基本的 AVR 设计思想都是一样的。引脚或许有所不同，但其所需的基本支持相同。无论如何，查阅合适的技术手册是必要的，它将给出你所用特殊处理器的具体细节。

总线接口

在这一节要介绍的是如何通过连接基于总线的存储器和外设来扩充处理器的功能。在开始介绍之前，我们先来快速浏览一下技术手册中的那些高深莫测的时序图表，并理解它们的含义。

时序（timing）

时序图表是设备输入输出信号的表示方法，而且从中还能体现出输入输出信号间的关系。实质上它表明的是什么时候一个信号需要确认，什么时候你会从设备得到一个期望的响应。对于两个相互作用的设备而言，它们之间的时序信号必须是兼容的，否则你必须增加额外的电路以使其兼容。

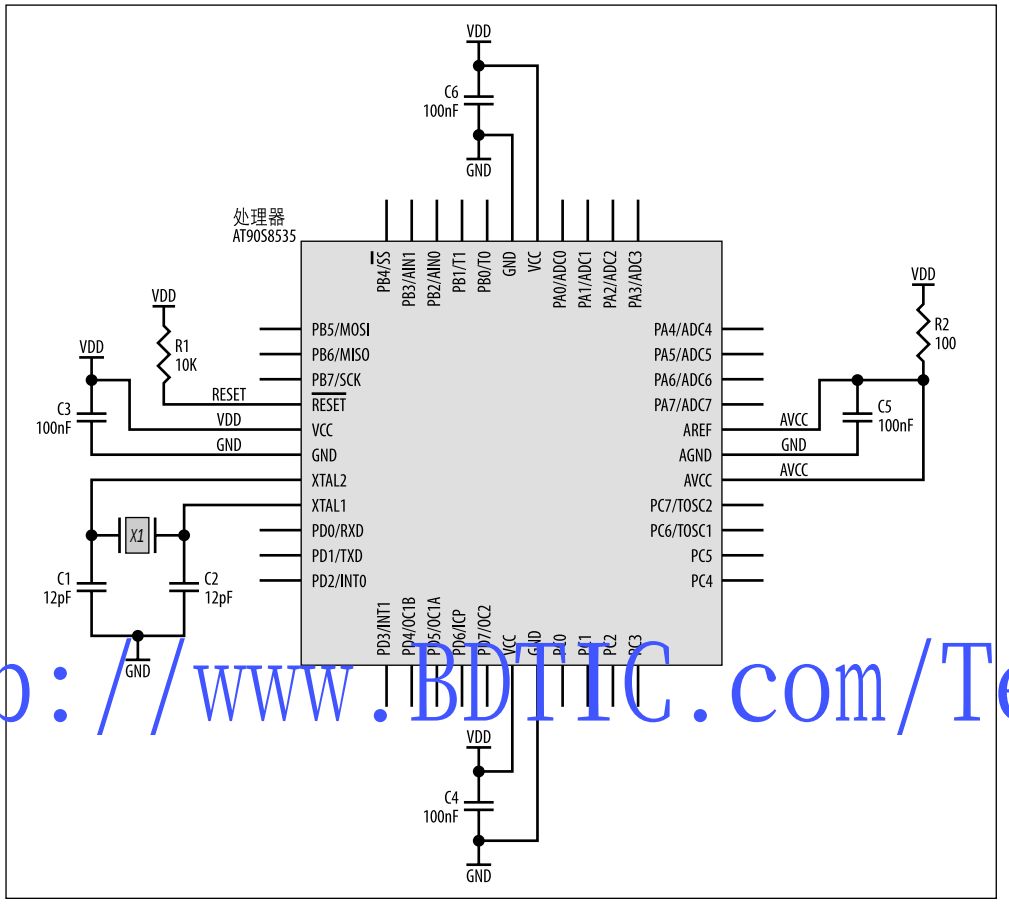

图 6-10：AT90S535 处理器及其支持元件

由于时序图表看起来的确会让很多人感到迷惑和畏惧，所以它常常被完全忽略。不过可以肯定的是，如果你忽略它，那么你设计的系统将很难工作。更何况时序图并非像看起来那样难以理解和使用。如果你想设计和构建一个可靠的系统，那么时序必不可少。

数字信号有三种状态：高电平、低电平和高阻态（三态）。图 6-11 所示的即是数字设备时序图表中的这三种状态。

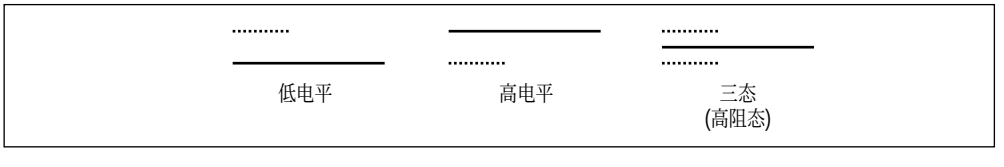

图 6-11：数字信号状态图

图 6-12 所示为一种状态到另一种状态间的转换。

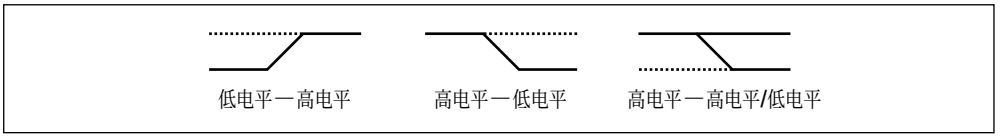

图 6-12：状态间的转换

图 6-12 中最后一种波形（高—高 / 低）表明如果一个信号为高电平，那么在时间轴上某个特定的点，信号既可能保持高电平，也有可能转变为低电平。同样，一个处于高阻态的信号线，也可以转换成高电平或低电平，具体如何转换由系统的状态决定。一个具体例子就是数据线，它在信息开始传输之前都将保持高阻态，而一旦信息开始传输，它就变为高电平（data=1），或变为低电平（data=0），如图 6-13 所示。

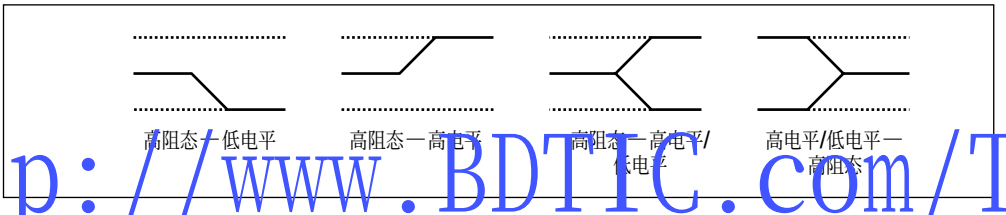

图 6-13：高阻态的转变

图 6-14 中的波形表示的是从高阻态转换到高 / 低电平又回到高阻态。这表明状态的变化可以发生在给定时间段上的任何一点，但必须发生在时间轴上的指定点。

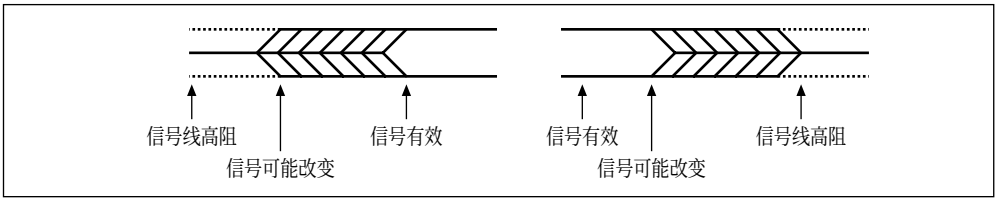

图 6-14：时序的转变

图 6-15 中的波形表明，一个信号可能在时间图上的任意给定点改变状态，信号可能处于高电平，并将持续这一状态或是变为低电平。或者，信号可能处于低电平，并将持续这一状态或是变为高电平。

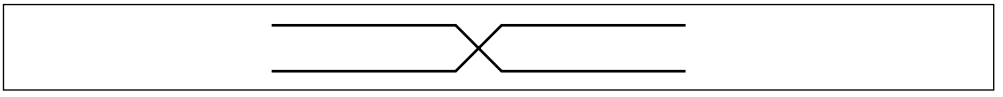

图 6-15：信号状态的改变

也许我们从很多关于数字电路的参考书上都得到这样的印象 :数字电路中信号状态的改变都是瞬时发生的。其实不然。这种变化从来都不是瞬时发生的 ;它可能会持续几纳秒，并且持续的时间会随设备的不同而差别很大（如图 6-16 所示）。每个元件的技术手册上都会详细列出设备的信号转换时间。

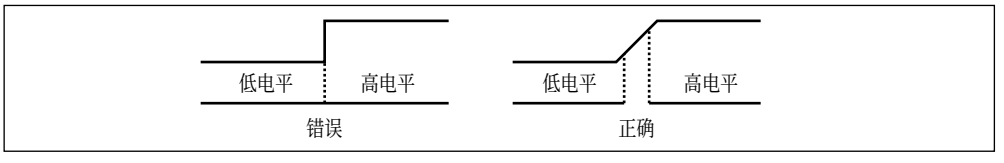

图 6-16：信号变换时序

元件制造商的技术手册中都列出了所生产的设备的时序信息，图6-17所示为我们虚构的一个设备的时序图表。

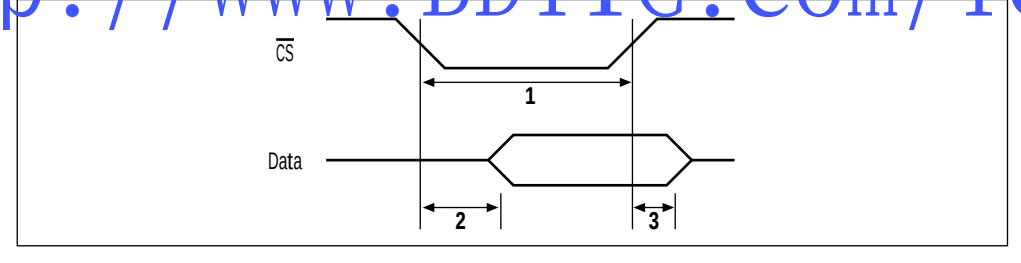

图 6-17：时序图表示例

时序图表还体现了设备的输入信号（如 **$\overline{\text{CS}}$**）与设备输出信号（如 **Data**）之间的关系，图中的编号参见时序信息列表，它们没有直接表示电路的时序。表 6-2 给出了一些技术手册中常见的时序参数信息。

表 6-2：时序参数示例

参考	功能描述	最小值	最大值	单位
1	CS 持续时间	60		ns
2	CS 到有效数据输出		30	ns
3	Data 持续时间	5	10	ns

时序参考 1（见表 6-2 第一行）给出了 $\overline{CS}$ 必须持续为低电平的时间。本例中，最小的持续时间为 60ns。这意味着 $\overline{CS}$ 必须持续低电平至少 60ns 才能为设备所识别。$\overline{CS}$ 持续时间没有最大值的限制，持续时间超过最低要求的 60ns 对系统没有什么影响。

时序参考 2 给出的是当 $\overline{CS}$ 为低电平时，设备响应这种变化所需要的时间。从 $\overline{CS}$ 变为低电平直至设备开始输出数据的时间最多为 30ns，这意味着在 $\overline{CS}$ 变为低电平后的 30ns，设备将把有效的数据输出到数据总线上。当然，输出数据也可能早于 30ns，可以确定的就是设备的响应时间不会超过 30ns。

时序参考 3 说明，一旦 $\overline{CS}$ 信号反转，设备应该在什么时候停止数据的传输。其中给出的参考值最小为 5ns，最大为 10ns。这意味着，当 $\overline{CS}$ 反转后，数据传输将持续 5ns，但不应该超过 10ns。

一些制造商使用数字作为时序参考，另外一些则使用标签，如图 6-18 所示。

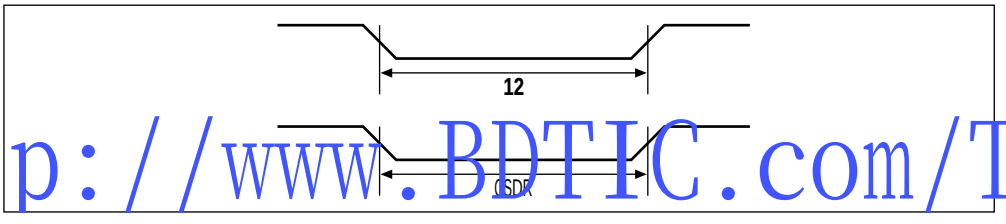

图 6-18：时序参考

一些制造商把时序长度定义为从一个信号从变为有效直至它变为无效的时间段，还有一些制造商则定义时序为两次相邻变化中间时刻的时间间隔（如图 6-19 所示）。

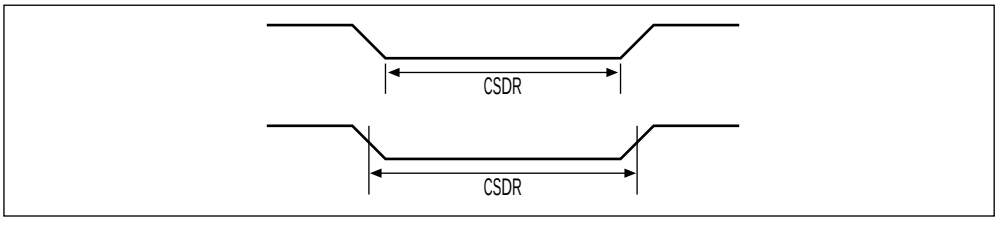

图 6-19：时序长度

现在我们将使用所学的知识去看一下真正的处理器的时序。不同的处理器结构会有不同的信号和时序，但只要你掌握了其中一种，那么基本的原理也适用于其他种类。由于绝大多数的小型微控制器都没有外部总线，所以我们的选择非常有限。我们将要讨论的是

AT90S8515 芯片，它是惟一一款带有外部总线的 AVR 处理器。而在 PIC 世界中，PIC17C44 可以使用基于总线的接口。

AT90S8515 存储器周期

存储器周期（也称机器周期或处理器周期）被定义为处理器从发起存储器（或外设）访问，经过数据传输，再到访问终止的时间。存储器周期由处理器产生，通常是一个固定的时间段（或一个给定时间段的倍数），可能需要若干个（处理器）时钟周期来完成。

存储器周期通常分为两类：读周期和写周期。对存储器和外设而言，当数据被选中（且读或写周期被确认）后，要求数据必须持续一段时间。这就给系统设计者提出了要求。胶合层逻辑（介于处理器和其他设备之间的接口逻辑电路）必须在一定的时间内执行自身的功能，如选择正在访问的外设。设置的时间必须要满足胶合层逻辑的时间要求，否则计算机将无法工作。胶合层逻辑电路监测来自处理器的地址，并选择一个设备作为地址解码器。稍后我们将更详细地介绍地址解码器。

在计算机的设计中，对时序的处理也许是最为关键的一个环节。举个例子，如果一个给定的处理器有一个 150ns 的时钟周期，而存储设备需要其中的 120ns 来完成发起访问到数据传输的工作，那么只将时钟周期最初的 30ns 留给胶合层逻辑电路，用以处理处理器信号。一个 74LS 系列的 TTL 门电路典型的传输延迟为 10ns。因此，在本例中可以使用两个以上的 74LS 门电路（依序）来实现地址解码电路，以便满足时间的要求。

同步处理器的存储器周期大小固定，所有处理器时序与时钟直接相关。假定系统内的所有设备都能被访问，而且能在给定的存储周期内作出响应。如果系统中某一设备的响应时间大于存储器周期所允许的时间，就需要增加逻辑电路来暂停处理器的访问，以使慢速设备有足够的响应时间。处于暂停状态时的每个时钟周期都称为等待状态。一旦慢速设备准备就绪，处理器再由逻辑电路释放，继续执行存储器周期。这种暂停处理器来同步慢速设备的方法称为插入等待状态 ;使处理器处于等待状态的电路就被称为等待状态产生器，它可以由一组触发器（触发器的功能是一个简单的计数器轻易地产生）。该产生器由处理器的输出激活，以表明存储器周期的开始，并且在每个存储周期结束时复位，从而返回一个已知状态。（还有一些处理器内置有可编程的等待状态产生器。）

而对于异步处理器来说，在给定的几个时钟周期内，它不会终止其存储器周期。相反，它等待来自外设或辅助逻辑电路的传输确认，以判断它所访问的外设是否有足够的时间在存储周期内完成操作。换句话说，处理器能够在存储周期内自动插入若干等待状态，直到待访问的设备准备就绪。如果处理器没有收到确认，就会无限期地等待。因而，很

多计算机系统在使用异步处理器时，都会用一个附加的逻辑电路来协同工作。当一个存储周期太长（即有可能不能终止）时，它就将处理器重启。异步处理器还能作为同步处理器使用，只要把确认线固定为激活状态即可，之后它假定所有的设备都能与之同步，即在运行时无需插入等待状态。

绝大部分的微控制器都是同步的，而大部分大型处理器则是异步的。AT90S8515是同步处理器，内部有一个等待状态产生器，可以插入单个等待状态。

总线信号

图6-20所示为一个带有配套元件的AT90S8515处理器。AT90S8515用来和外界接口的总线包括：一个地址总线、一个数据总线以及一个控制总线。由于处理器的引脚数目有限，所以这些总线通常会与处理器的数字I/O端口（端口A和C）共享引脚。控制寄存器中的一个比特位被用来区分引脚是用于I/O，还是连接总线。现在一个16位的地址总线加上一个8位的数据总线一共是24位，但端口A与端口B却只有16位。那么处理器是如何使24位的总线来适合16位的端口的呢？采用的方法就是将地址总线的低8位与数据总线复用。当存储器访问开始时，端口A输出地址位A0到A7。处理器还提供了一个控制线——地址锁存使能（Address Latch Enable，ALE），用来控制锁存器，例如74HCT573（如图6-20右侧所示）。当**ALE**为低电平时，锁存器获得并保持低8位地址。同时，端口B输出高8位地址A8到A15。这些地址位在整个存储器访问过程中都将保持有效。一旦锁存器将低位地址锁存，那么端口A将成为处理器与外部设备之间的信息传输数据总线。图6-20所示的电路中还包括：晶体振荡电路、系统内编程端口、使处理器电源退耦的电容以及用于其他重要信号的网络标识。

图6-21所示为AT90S8515的时序图表。只有当处理器的等待状态产生器被激活时，周期T3才存在。

现在，让我们更详细地看一下这些信号。（在后面你还将看到如何实际运用这些信息，但现在我们要做的只是“浏览”一下信号的时序图。）时序信息的数字可以查询ATMEL公司的技术手册得到，该手册可以从其网站获得。图6-22所示的即是ATMEL公司技术手册上的时序信息，并且具有完备的时序参考说明。

参考信息可以通过查找处理器技术手册中相应表格而得到（见表6-3）。

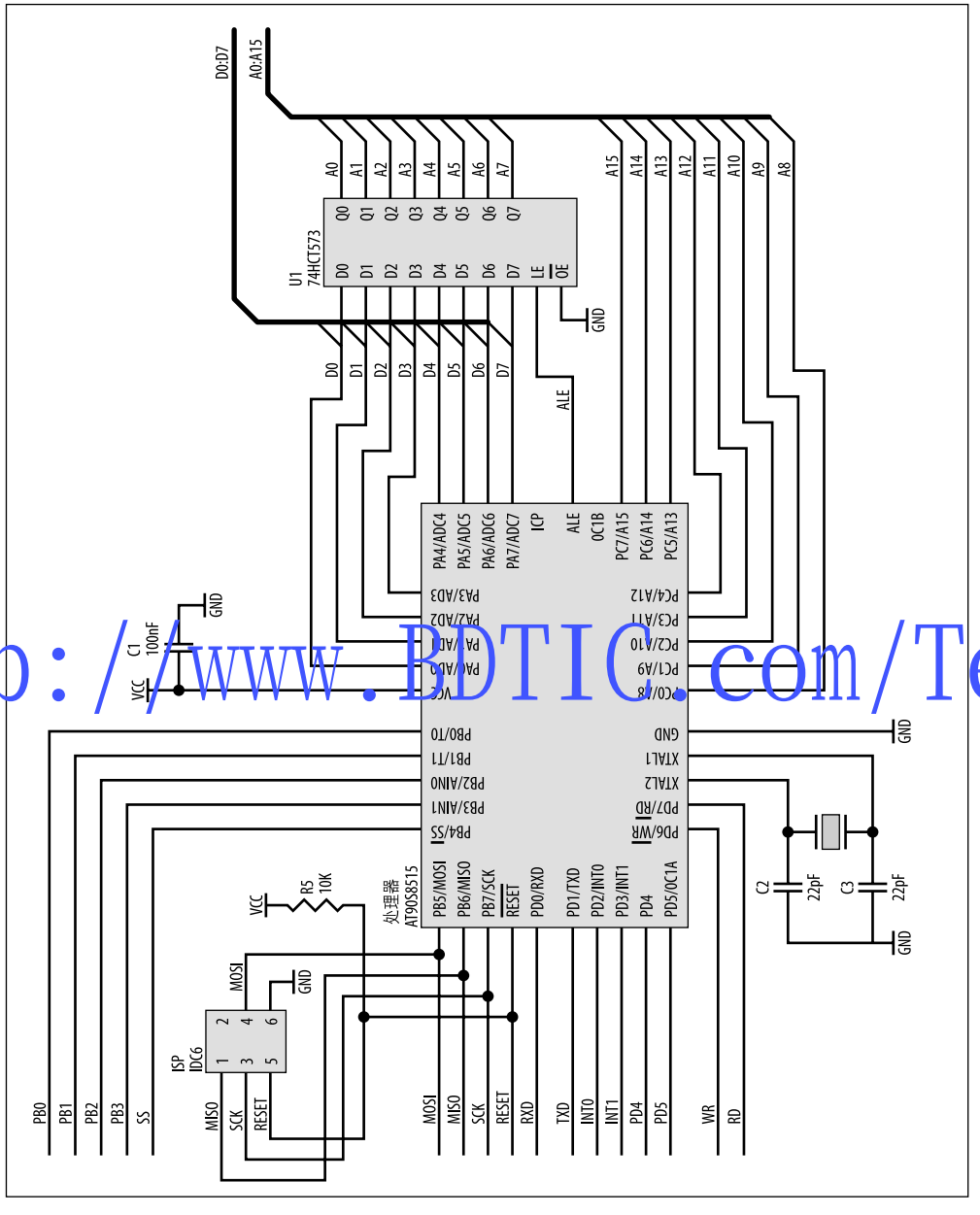

图 6-20：地址总线的多路复用

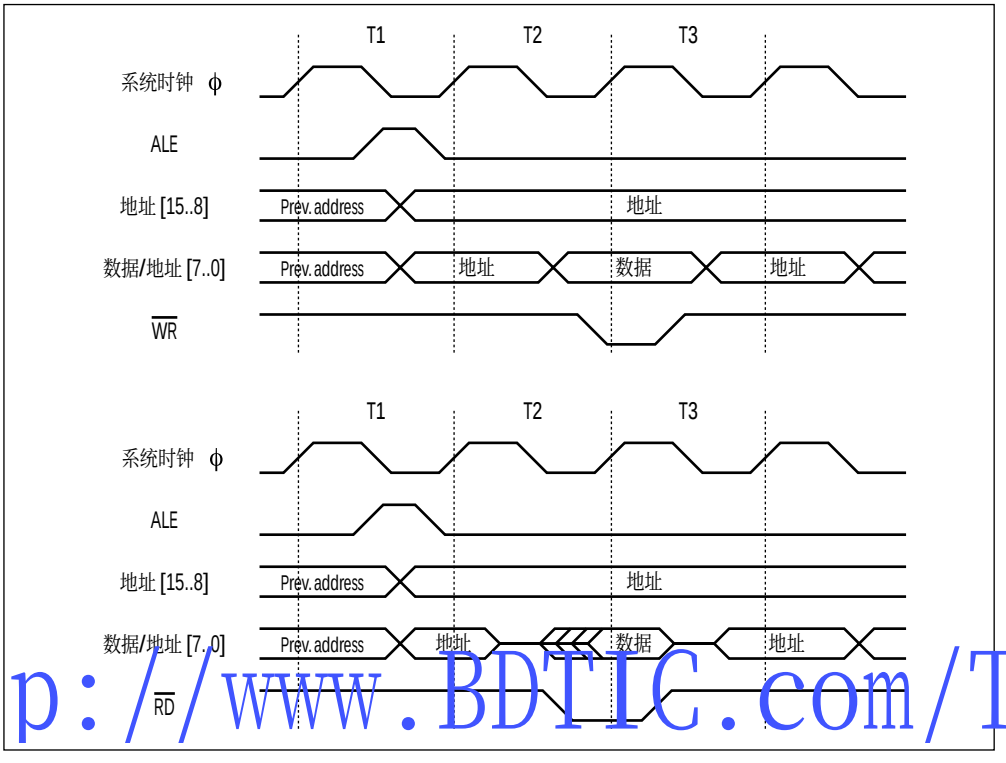

图 6-21：AT90S8515 存储器周期

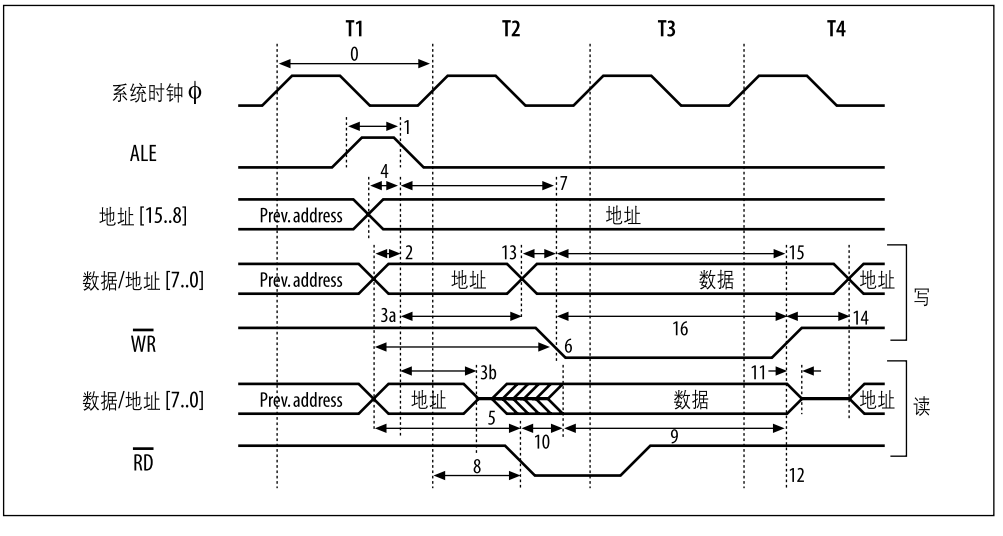

图 6-22：带有时序参数的 AT90S8515 存储器周期

表 6-3：时序参数表

	标号	参数	8MHz 振荡器		可变振荡器		
			最小值	最大值	最小值	最大值	单位
0	$1/t_{CLCL}$	振荡频率			0.0	8.0	MHz
1	t_{LHLL}	ALE 脉冲宽度	32.5		0.5_{tCLCL}-30.0		ns
2	t_{AVLL}	地址 A 有效直到 ALE 为低电平	22.5		0.5_{tCLCL}-40.0		ns
3a	t_{LLAX_ST}	ALE 为低电平后地址保持，ST/STD/STS 指令	67.5		0.5_{tCLCL}-50.0		ns
3b	t_{LLAX_LD}	ALE 为低电平后地址保持，LD/LDD/LDS 指令	15.0		15.0		ns
4	t_{AVLLC}	地址 C 有效直到 ALE 为低电平	22.5		0.5_{tCLCL}-40.0		ns
5	t_{AVRL}	地址有效直到RD 为低电平	95.0		1.0_{tCLCL}-30.0		ns
6	t_{AVWL}	地址有效直到 WR 为低电平	157.5		1.5_{tCLCL}-30.0		ns
7	t_{LWL}	ALE 为低电平直到 WR 为低电平	105.0	145.0	1.0_{tCLCL}-20.0	1.0_{tCLCL}+20.0	ns
8	t_{LLRL}	ALE 为低电平直到 RD 为低电平	42.5	82.5	0.5_{tCLCL}-20.0	0.5_{tCLCL}+20.0	ns
9	t_{DVRH}	Data 设置直到 RD 为高电平	60.0		60.0		ns
10	t_{RLDV}	Read 为低电平直到 data 有效		70.0		1.0_{tCLCL}-55.0	ns
11	t_{RHDX}	RD 为高电平后 Data 保持	0.0		0.0		ns
12	t_{RLRH}	PD 脉冲宽度	105.0		1.0_{tCLCL}-20.0		ns
13	t_{DVWL}	Data 设置直到 WR 为低电平	27.5		0.5_{tCLCL}-35.0		ns
14	t_{WHDX}	WR 为高电平后 Data 保持	0.0		0.0		ns
15	t_{DVWH}	Data 有效直到 WR 为高电平	95.0		1.0_{tCLCL}-30.0		ns
16	t_{WLWH}	WR 脉冲宽度	42.5		0.5_{tCLCL}-20.0		ns

由于所有处理器的操作都以时钟为基准，所以在两个波形图的上方都有系统的时钟φ作

为参考。在ATMEL的技术手册中，时钟周期用t_{CLCL}（注1）标识，其值为频率的倒数。对一个8MHz的时钟来说，时钟周期等于125ns。T1、T2和T3的时钟宽度都为t_{CLCL}。

处理器的时钟周期不会孤立地存在，它总是（注2）有一个前导周期和一个后继周期。我们能从时序图中看到这一点。在周期开始时，上一次访问的地址仍驻留在地址总线中。在T1周期的下降沿，地址总线上的地址才变换成本次访问的有效地址。端口A表示地址位A0～A7，而端口B表示地址位A8～A15。同时，**ALE**变为高电平，释放外部地址锁存器，以为获得来自端口A的地址作准备。**ALE**维持高电平的时间为$0.5 \times t_{CLCL} - 30ns$。举个例子，如果一个AT90S8515运行在8MHz，则**ALE**保持高电平时间为32.5ns（0.5×125ns－30ns=32.5ns）。**ALE**变为低电平时，外部锁存器便会得到并保持新的低8位地址位。在**ALE**下降之前，地址位保持有效的时间为$0.5 \times t_{CLCL} - 40ns$，换句话说，就是要在T1周期末系统时钟上升之前保持40ns。当**ALE**下降时，而且在端口A变为数据位之前，对于写周期，低地址位需要在端口A保持$0.5 \times t_{CLCL} + 5ns$，但对于读周期，则只需要保持15ns。在读周期时保持的时间短得多，其原因是处理器希望只要有可能就释放信号引脚；因为这是一个读周期，需要外部设备作出响应，这就意味着处理器必须尽可能地为响应信号释放出通道。

对一个写周期来说，在**ALE**变为低电平后的$t_{CLCL} - 20ns$，写选通信号$\overline{\mathbf{WR}}$变为低电平。这对于外设来说，表明处理器已把有效的数据输出到数据总线上。$\overline{\mathbf{WR}}$将保持低电平$0.5 \times t_{CLCL} - 20ns$，这段时间允许外设作好将数据读入锁存器的准备。在$\overline{\mathbf{WR}}$的上升沿，外设被允许将数据总线上的数据锁存。此时，写周期结束，下一周期将开始。

对于一个读周期来说，在**ALE**变为低电平后的$0.5 \times t_{CLCL} - 20ns$，读选通信号$\overline{\mathbf{RD}}$变为低电平。$\overline{\mathbf{RD}}$低电平的持续时间为$t_{CLCL} - 20ns$，在这段时间内，外设应当将有效的数据放到数据总线上。只要能保证在$\overline{\mathbf{RD}}$再次变为高电平之前，数据能够稳定地保持至少60ns，那么外设便可以在$\overline{\mathbf{RD}}$下降后的任意时刻发送数据。此时，处理器将锁存来自外设的数据，读周期终止。需要注意，很多处理器可能没有一个单独的读使能信号，因而它必须由外部逻辑电路产生。这基于一个前提，即如果一个周期不是写周期，那么它一定是读周期。

上述就是AT90S8515处理器如何访问其总线上连接的任意外部设备，不论这些设备是存储器芯片还是外围设备。但在实际应用中它又是如何工作的呢？下面我们将要看到的就

注1： 参考手册里术语的语义常常模糊不清，术语*CL*出自时钟clock。由于Atmel采用四个下标字符来描述时序参考，因此他们就用两个*CL*来对其标注。你的确没有必要去了解下标字符究竟代表什么意思，你只需知道这些下标字符所代表的信号及实际相关号码即可。

注2： 在此，我忽略了复位或断电前一刻所产生的情形。

是带有一些外设的基于 AT90S8515 的计算机的设计（注 3）。在这个例子中，我们将把处理器连接到一个静态 RAM 和一些用以驱动几组 LED 的简单锁存器上。

内存映射和地址解码

对于处理器而言，它的地址空间是一个很大的线性区域。虽然处理器的地址空间也许由很多内部和外部设备组成，但在地址空间中它们并没有什么差别。对处理器来说，同样都是简单地执行存储访问操作。系统设计者（也就是你）的工作就是给每个设备分配存储区域，并提供地址解码逻辑电路。在对外访问期间，处理器将地址传送给地址解码器，由它来惟一地选择正确的设备（如图 6-23 所示）。例如，如果我们的 RAM 占用了存储空间的一段地址区域，那么只要处理器访问该地址区域， RAM 就会被选中（而不会是其他什么设备）。简言之，就是当访问的地址不在这一区域时，RAM 就不应当被选中。

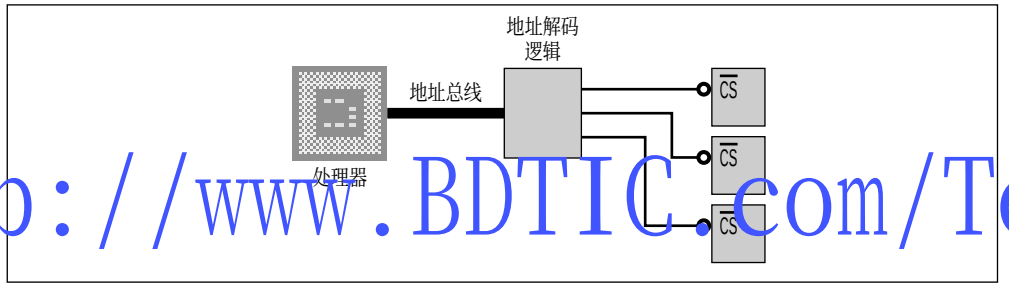

图 6-23：使用地址选择设备的地址解码器

在地址空间中对设备进行分配的过程就是所谓的内存映射或是地址映射过程。图6-24所示的是AT90S8515处理器的地址空间。我们所连接到处理器上的任何设备都必须映射到数据存储空间中，这样我们就可以忽略处理器内部的程序存储器。由于处理器采用的是哈佛结构，因而程序空间是一个完全独立的地址空间。有 64K 的数据空间用于处理器的内部资源：工作寄存器、I/O 寄存器以及 512 字节的内部 SRAM。它们占用了数据空间中的最低地址部分。任何大于 0x0260 的地址都可以由我们自由分配使用。（注意：并非所有的处理器都采用这种内存映射方式，有些处理器的外围设备可以使用整个存储空间。）

现在，我们首要的任务是把剩下的地址空间分配给外部设备。由于 RAM 的大小为 32K，所以可以将它放在地址空间的高位地址部分（0x8000~0xFFFF）。如果设备地址处于整数边界，那么地址解码就会容易得多。把高位地址 0x8000~0xFFFF 分给 RAM，则低位地址空间用来存放锁存器和处理器的内部资源。目前，一个锁存器只需要一个字节的地

注 3： 由于我们先前已经介绍过振荡器和电路内编程，在此就不对其进行讨论了。但这并不意味着你应该从你的设计中去掉它们。

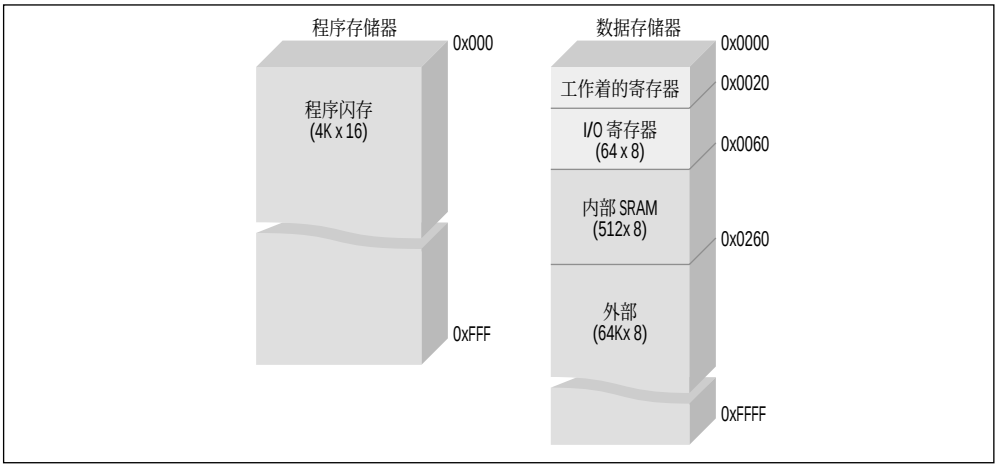

图 6-24：ATMEL AT90S8515 内存映射

址空间。因此，如果有三个锁存器，那么我们只需给它们分配 3 个字节的地址空间即可。这就是所谓的显式地址解码。然而，可以通过一个很典型的例子来说明这种地址分配方式并不总是很有效的。假如，要对 3 个字节进行解码，将需要使用 14 位的地址解码器。而使用这么多的逻辑电路去选择三个设备显然没有必要。一个更好的解决方法是简单地将剩余的地址空间划分为四部分，其中的三部分分别用作锁存器，另一部分暂不使用（留给处理器的内部资源）。这种方法被称为部分地址解码，显然它的效率更高。它的技巧就是使用最少量的地址信息来为你的设备解码。

图6-25所示的是SRAM和三个锁存器的地址映射。我们注意到：最低地址部分0x0260～0x1FFF 没有被使用。

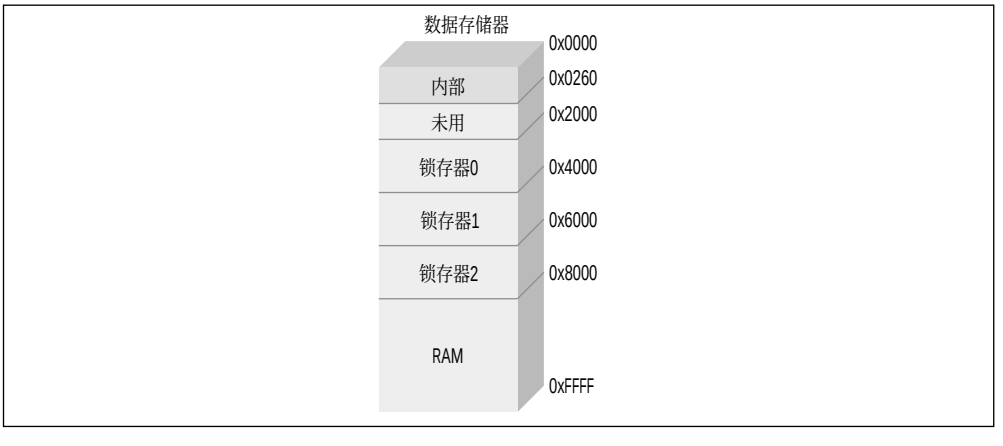

图 6-25：已分配的内存映射

0x2000～0x3FFF 区域内的任何地址都将选择锁存器 0，尽管锁存器只需要 1 字节的空间。因而，这个设备便被镜像到那个地址空间。为了便于编程，你通常会选择只选择一个地址（如 0x2000），并且在你的代码中使用它。但同样你也可以仅简单地使用地址 0x290F，这并不会影响你的工作。

现在我们有了自己的内存映射，接下来我们需要设计一个地址解码器。首先列出设备以及它们相应的地址（见表 6-4）。我们需要查找对于不同的设备，它们地址位中哪些是不同的；对于一个给定的设备区域，它们的地址位中又有哪些是相同的。

表 6-4：地址列表

设 备	地址范围	A15…A0
未用	0x0000~0x1FFF	0000 0000 0000 0000 0000~0001 1111 1111 1111 1111
锁存器 0	0x2000~0x3FFF	0010 0000 0000 0000 0000~0011 1111 1111 1111 1111
锁存器 1	0x4000~0x5FFF	0100 0000 0000 0000 0000~0101 1111 1111 1111 1111
锁存器 2	0x6000~0x7FFF	0110 0000 0000 0000 0000~0111 1111 1111 1111 1111
RAM	0x8000~0xFFFF	1000 0000 0000 0000 0000~1111 1111 1111 1111 1111

那么对每个设备来说，什么是地址惟一性的关键呢？通过上表我们可以发现，对 RAM 来说，地址位（以及地址信号）**A15** 是高电平，而对其他设备来说 A15 是低电平，因而我们可以使用 **A15** 作为选择 RAM 的关键。对于锁存器而言，地址位 **A15**、**A14** 和 **A13** 是决定性的。于是我们可以重新绘制上表，使其更为清晰（这也是更常用的构造地址列表的方法，见表 6-5）。表中“ x ”的意思是无关位（即置 0 或置 1 均可）。

表 6-5：简化的地址列表

设 备	地址范围	A15…A0
未用	0x0000~0x1FFF	000x xxxx xxxx xxxx xxxx
锁存器 0	0x2000~0x3FFF	001x xxxx xxxx xxxx xxxx
锁存器 1	0x4000~0x5FFF	010x xxxx xxxx xxxx xxxx
锁存器 2	0x6000~0x7FFF	011x xxxx xxxx xxxx xxxx
RAM	0x8000~0xFFFF	1xxx xxxx xxxx xxxx xxxx

因此，要解码 RAM 地址，只需要使用 **A15**。如果 **A15** 为高电平，则 RAM 被选中。如果 **A15** 为低电平，那么其余的设备中将有一个被选中，但 RAM 不会再被选中。现在，RAM 有一个片选信号 $\overline{\mathbf{CS}}$，是低电平触发。因而当 **A15** 上升时，$\overline{\mathbf{CS}}$ 应该为低电平。所以，RAM 的地址解码器只是简单地将 **A15** 反转，这可以通过使用一个反转芯片如

74HCT04（如图 6-26 所示）来完成。通常在设备被选择后，片选信号都被标识，所以 RAM 的片选信号被标识为 **$\overline{\text{RAM}}$**。

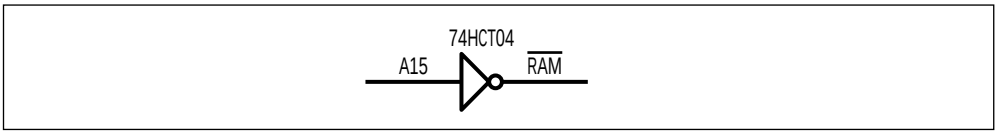

图 6-26：RAM 的地址解码器

注意，为了使RAM作出响应，需要有一个片选**$\overline{\text{CS}}$**以及一个来自处理器的读或写选通器。处理器的所有其他的地址线被直接连接到 RAM 的相应地址输入端（如图 6-27 所示）。

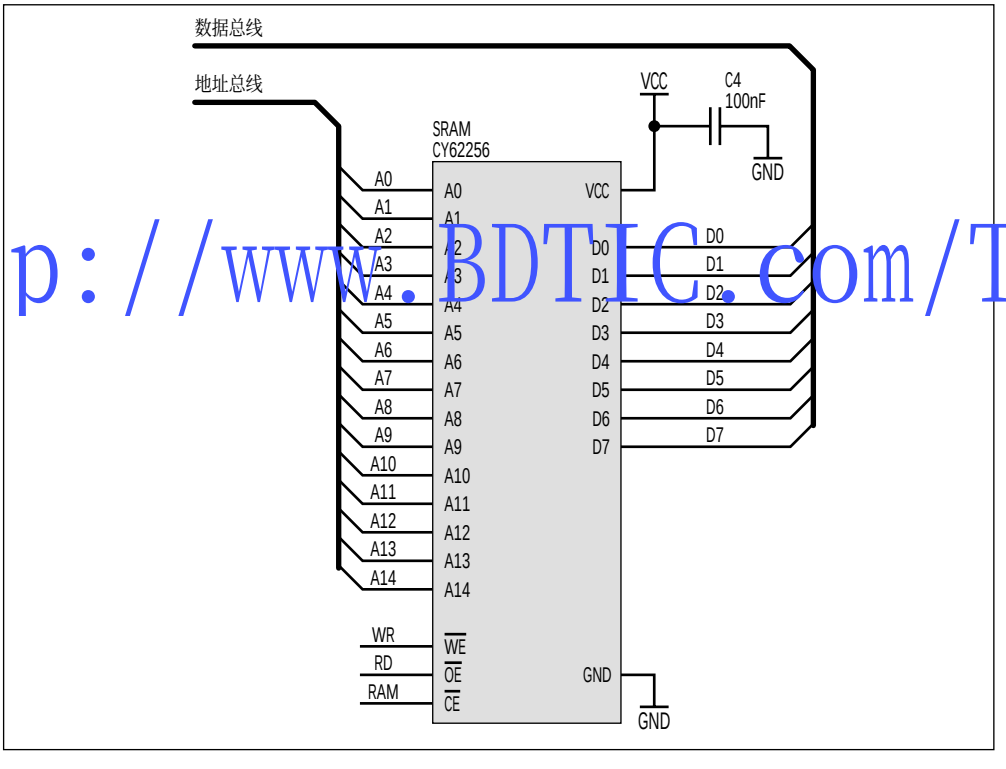

图 6-27：连接到 SRAM

现在来看一下其余的四个区域。**A15**必须为低电平，而**A14**和**A13**足以用来区分那些设备。如果我们的解码器使用离散逻辑电路，则还需要一些门电路的支持，这可能会使电路变得很混乱。这里还有一个更简单的方法 —— 我们可以使用一个 74HCT139 解码器

(注 4)，它有两个地址输入（A 和 B），能够产生四个惟一的低电平触发的片选信号输出（标记为 Y0...Y3）。因此，图 6-28 所示为计算机中所使用的完整地址解码器。

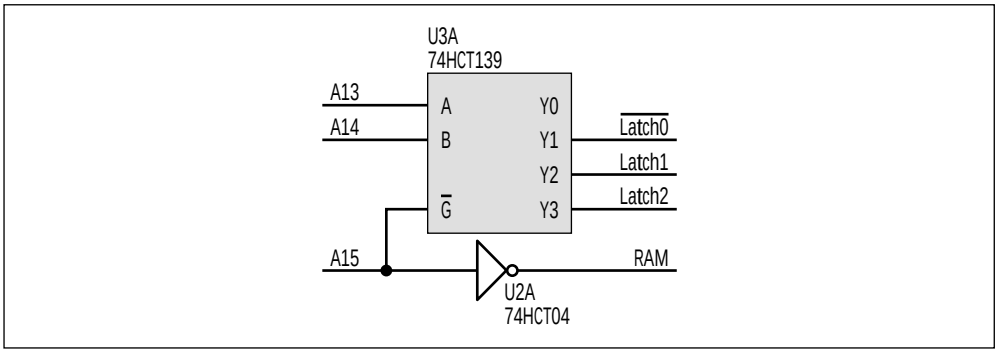

图 6-28：完整的地址解码器

74HCT139 解码器使用 **A15**（低电平）作为触发信号（输入端 $\overline{\mathbf{G}}$），因此，**A15** 也被用来作为地址解码的一部分。如果我们要解码 8 个区域而非 4 个区域，那么我们需要使用 74HCT138 解码器，它有 3 个地址输入端，可以产生 8 个片选信号。

处理器与输出锁存器之间的接口很简单。我们可以使用同种类型的锁存器（74HCT573），用来实现对地址的多路复用。这种输出锁存器可以应用于任何需要额外数字输出的环境中。如图 6-29 所示的示例电路中，我使用 74HCT573 锁存器去控制 8 个一组的 LED 灯。

来自 74HCT139 地址解码器的输出被用来驱动 74HCT573 锁存器使能输入端 LE（Latch Enable）。只要处理器访问那些分配给这种设备的存储区域，那么地址解码器将会触发锁存器以获取数据总线上的信息。因此，处理器会把一个字节的信息写入到锁存器地址区域中的任意地址上，接着这一字节的信息便会被输出到 LED 组。（对应比特位为 0 时 LED 灯亮；为 1 则 LED 灯灭。）

需要指出的是，锁存器的输出使能端 $\overline{\mathbf{OE}}$ 恒接地。这意味着锁存器总是显示最后被写入的字节信息。这样做很重要，因为我们希望 LED 灯在处理器对它们访问的整个期间一直处于显示状态，而不仅仅是闪一下。

使用锁存器 74HCT139 相对于使用离散逻辑门可以简化我们的设计，但仍有更好的方法实现系统的集成。

注 4： 在每一个 74HCT139 芯片中实际有两个独立的解码器，我们只用到其中一个。

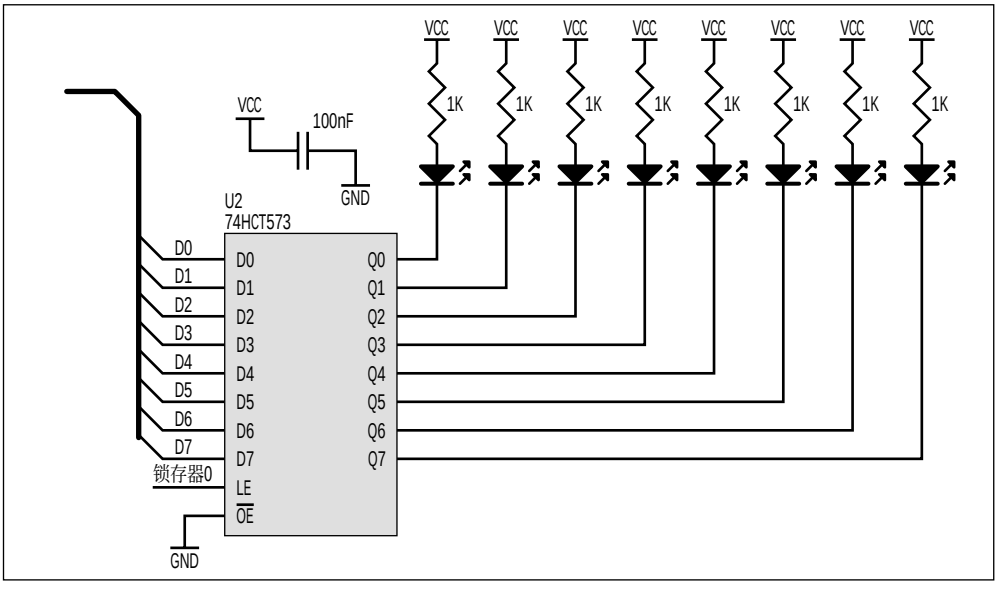

图 6-29：使用 74HCT573 锁存器控制 LED 组

可编程逻辑电路 PAL

支持逻辑电路很少是由单独的门电路来实现的。更常见的方法是使用可编程逻辑电路（PAL、LCA 或 PLD）（注 5）来实现计算机系统所要求的各种各样的混合功能。这种设备的特点是快速、紧凑、功耗低。而且由于它们可以再编程，因而使系统设计更加容易，功能更多。

有很多种设备都是可用的，从可以用来实现胶合层逻辑电路的简单的芯片（也是我们将要介绍的），到带有成百上千个门电路的大规模设备。

注意：Altera（*http://www.altera.com*）、Xilinx（*http://www.xilinx.com*）、Lattice 半导体公司（*http://www.latticesemi.com*）以及 Atmel 都是一些大规模可编程逻辑设备的开发商；生产这些大规模芯片的技术已相当成熟，以至于你甚至都可以将整个计算机系统都集成到一个芯片中。软件核心是用门电路实现的处理器设计，是可以集成到这些可编程逻辑设备中的。你还可以将串行接口、磁盘控制器、网络接口以及一系列的外围设备都集成到这些大规模的设备中。当然，从底层开始设计和试验你自己的处理器也别有乐趣。

每一系列的芯片都有自己的一套开发工具，这使你可以创建自己的设计（使用电路图或是使用某种编程语言如VHDL）来进行系统的仿真，并最终把自己开发的程序装载到芯片上。

注 5：分别为 Programmable Array Logic、Logic Cell Array 和 Programmable Logic Devices。

> 你甚至还能在这些芯片装载C编译器来执行算法，并将其转换为门电路逻辑（而不是机器代码）。在这种情况下，算法是在软件，而不是在硬件上运行，可效果是一样的（译注1）。这听起来很酷，但是用以实现这种功能的工具却很昂贵。如果你仅想实现一个小的嵌入式系统，它们显然超出了你的预算。而且对于我们的胶合层逻辑电路而言，这些芯片显然有点大材小用。

由于我们需要的逻辑电路并不复杂，所以一个简单（而且廉价）的PAL就足够了，它还可以使用公共的、免费的软件进行程序设计。

PAL被设计使用等式来表示内部逻辑：“+”表示OR，“*”表示AND，“/”代表NOT。（这些操作符最初用于布尔逻辑中，如果你有程序设计的背景，那么这些符号对你来说可能有些古怪，因为你可能更习惯于使用“|”、“&”和“!”。）这些等式使用诸如PALASM，ABEL或CUPL等软件编译，生成一个JED文件。它使用了一个PAL烧录器去配置PAL。在很多情况下，标准EPROM烧录器也用来给PAL编程。

PAL有输出引脚，输入引脚，还有一些引脚既可用作输出，也可用作输入。你可以使用PAL的大部分引脚。在你的PAL源代码文件（PDS文件）中，你可以声明正在使用的引脚，并标识它们。这与你在程序的源代码中声明变量很类似，惟一不同之处在于你不是给变量分配RAM上的存储单元，而是给芯片分配物理引脚。然后，你可以在等式中使用这些引脚标识去定义内部逻辑。我们的地址解码器（在PAL内部实现）将使用如下等式定义解码逻辑：

```
RAM = /A15
LATCH0 = /(/A15 * /A14 * A13)
LATCH1 = /(/A15 * A14 * /A13)
LATCH2 = /(/A15 * A14 * A13)
```

上述等式的表达方式可以使你很容易地将其和前面列出的地址列表相比较。你完全可以简化这些等式，但没有这个必要。正如一个优化的C编译器将会简化（并且加速）你的程序代码那样，PALASM也会重构你的等式，使其更适于PAL。

在22V10 PAL上编程实现前面所述地址解码功能的PDS文件的源码如下：

```
TITLE decoder.pds          ; 文件名
PATTERN
REVISION 1.0
AUTHOR John Catsoulis
COMPANY Embedded Pty Ltd
DATE June 2002

CHIP decoder PAL22V10      ; 定义正在使用的PAL设备
                           ; 并为其命名("decoder")
```

译注1： 即它已经成为具有软件功能的硬件，就是所谓的固件。

```
PIN   2   A15              ; 引脚声明及分配
PIN   3   A14
PIN   12  LATCH0
PIN   13  LATCH1
PIN   14  LATCH2
PIN   15  RAM

EQUATIONS                  ; 等式开始处

RAM=/A15
LATH0=/(/A15*/A14*A13)
LATH1=/(/A15*/A14*/A13)
LATH2=/(/A15*/A14*A13)
```

使用PAL设计系统逻辑电路的优点有两方面。当要改正bug或设计有变化时，PAL等式可以作出相应的改变。通过PAL的传输延迟相对固定而且短暂（无论是何种等式），这使得分析系统的时序信息变得简单多了。对于非常简单的设计而言，使用PAL或独立的芯片没有什么差别。但对于更复杂的设计来说，可编程逻辑电路是你惟一的选择。如果有可能，请尽量地使用可编程逻辑设备，而不是离散逻辑芯片，因为可编程逻辑设备将使一切都变得更加简单。

时序分析

现在我们已经完成了逻辑设计，但问题是：它确实能工作吗？下面我们就来分析一下它们的时序。这是最让人乏味，却也是计算机设计过程中最重要的一个环节。

我们首先要看到的是处理器的信号（和时序），然后是胶合层逻辑电路的作用，最后再看它们是否符合我们接口设备的要求。我们以SRAM为例进行分析，对于其他的设备，也可以用同一种方式进行分析。图6-30所示为SRAM读周期的时序图表。在此，我选用的RAM是由Cypress半导体公司生产的CY62256-70（32K）SRAM。大部分32K的SRAM都遵守JEDEC标准，这意味着它们的引脚和信号都是完全兼容的。所以，对一种SRAM适用的操作就对其他的SRAM也同样适用。不过我还是要再次强调：在使用前，应该先查阅所用单个设备的技术手册。

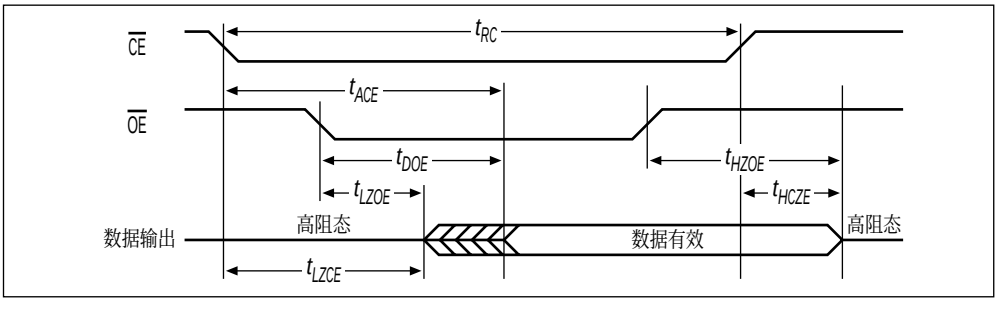

图6-30：RAM读周期的时序图表

型号中“ -70 ”意味着这是一个 70ns 的 SRAM，简单地说，就是对该芯片的访问时间为 70ns。现在从 CY62256-70 的技术手册（可以从网站 *http://www.cypress.com* 获得）中，我们可以看到 t_{RC} 的最小值为 70ns 。这意味着芯片使能 **$\overline{CE}$** 可以保持低电平的时间要不少于 70ns。因为 **$\overline{CE}$** 正是来自地址解码器的片选（**$\overline{RAM}$**）信号，所以我们必须保证地址解码器能够使 **$\overline{RAM}$** 维持低电平至少 70ns。要使 SRAM 在读周期内输出数据，必须有一个有效地址、一个激活的芯片使能信号以及一个激活的输出使能信号 **$\overline{OE}$**。而输出使能信号 **$\overline{OE}$** 就是处理器的读选通信号 **$\overline{RD}$**。上述三个条件都必须满足，芯片才可以响应数据。数据要在 **$\overline{CE}$**（t_{ACE}）变为低电平后的 70ns 或 **$\overline{OE}$**（t_{DOE}）被变为低电平后的 35ns（等两个条件都符合）才输出。现在，我们的 **$\overline{CE}$** 信号由地址解码器生成（反过来，地址解码器使用的又是来自处理器的信息），**$\overline{OE}$**（t_{DOE}）则来自处理器。在读周期的过程中，处理器将输出读选通信号以及一个地址，它们将依次触发地址解码器。在读周期的后半段，处理器将等待 RAM 的数据送到数据总线上。有一点非常关键：使 RAM 输出数据的信号必须保证处理器希望得到数据是有效的。这就对 RAM 的数据输出触发信号提出了一个很严格的要求。如果能满足这个要求，那么你就可以得到一个能读取外部存储器的处理器；否则，你的处理器将毫无用处。

现在我们开始讨论处理器。我们假定它的等待状态产生器已失效。对于一个 AT90S8515 处理器来说，它的每个动作都是 **ALE** 的下降沿触发。在 8MHz 的 AT90S8515 芯片上，用以输入到地址解码器的高位地址将在 **ALE** 电平降低之前的 22.5ns 变为有效。如果我们使用的地址解码器对输入变化的响应时间为 40ns（注 6），那么 RAM 的片选信号会在 **ALE** 下降后的 17.5ns 变为有效，如图 6-31 所示。

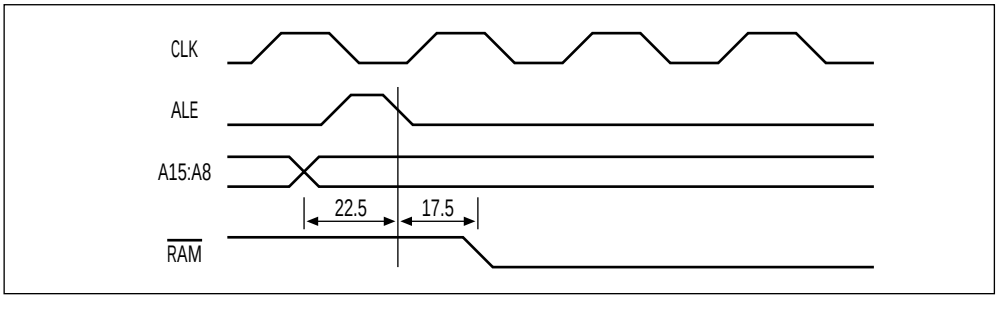

图 6-31：RAM 片选时序

而 **$\overline{RD}$** 将在 **ALE** 下降后的 42.5ns～82.5ns 之间变为低电平。由于 RAM 直到 **$\overline{RD}$**（**$\overline{OE}$**）为低电平才输出数据，所以我们在此取最坏的情况，即 82.5ns，如图 6-32 所示。

注 6： PAL 可能在 15ns 或更少的时间内响应，这也就是 PAL 较离散逻辑成为更好选择的另外一个原因。

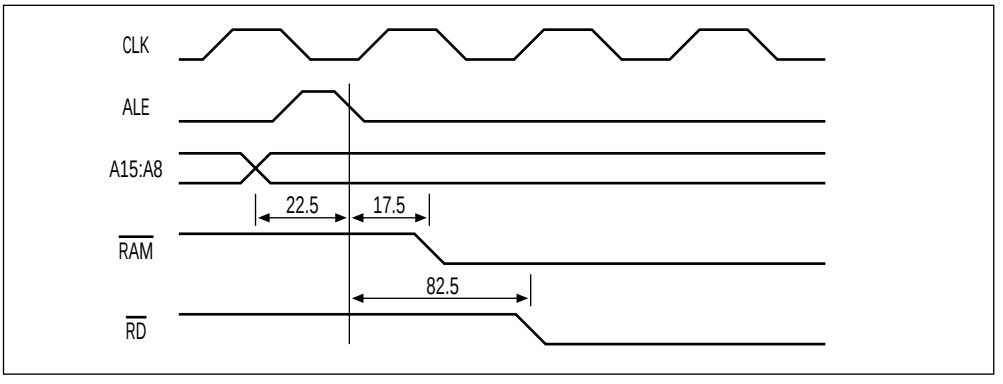

图 6-32：RAM 的读选通与片选信号

由于 **RAM** 要在 $\overline{\textbf{RAM}}$ 处于低电平状态 70ns 后以及 $\overline{\textbf{RD}}$ 处于低电平状态 35ns 之后才作出响应。于是，$\overline{\textbf{RAM}}$ 处于低电平 70ns 时 **ALE** 已变为低电平 87.5ns，而 $\overline{\textbf{RD}}$ 的低电平持续 35ns 时 **ALE** 已变为低电平 117.5ns。所以，在这种情况下 $\overline{\textbf{RD}}$ 是最终的控制信号。这意味着 SRAM 将在 **ALE** 变低后的 117.5ns 输出有效数据，如图 6-33 所示。

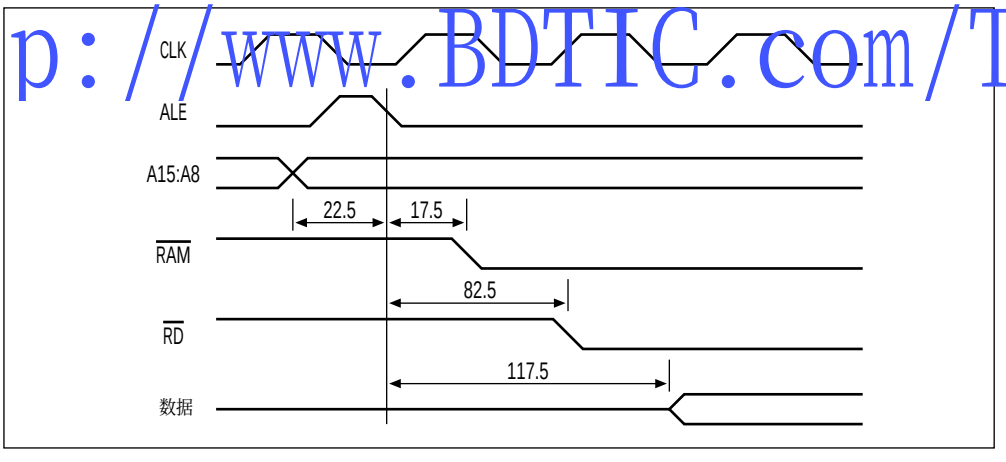

图 6-33：SRAM 的有效数据

现在，在 **ALE** 下降后 147.5ns 的读周期内，一个 8MHz 的处理器会将有效的数据锁存。因而，SRAM 将保留有效数据 30ns 以便处理器共享。到目前为止，一切设计都很完美。但在周期结束时又会怎样呢？这时，处理器希望数据总线能够被释放，以便在 **ALE** 下降 200ns 之后的下一次访问时可用。RAM 被 $\overline{\textbf{RD}}$ 释放直至它停止将数据发送到总线上，所需要的时间为 25ns。这意味着数据总线将在 142.5ns 时被 RAM 释放，这不会造成任何影响。

对写周期的分析也与上述过程类似。对每种不同的存储器访问周期以及对每一连接到处理器上的设备作同样的时序分析非常重要。如果技术手册没有列出相关信息，或是虽有涉及但说得很不直接，这时候分析时序是耗时和令人沮丧的，而且还很容易出错。然而我们却不得不做时序分析。否则我们只能寄希望于自己盲目的运气来使计算机运转，而且无论如何它都将不能算是一个好的设计。

存储器管理

在大部分的小规模嵌入式应用中，处理器与外部存储芯片都是直接连接的，然而，有时保持事物的原有次序是很有好处的，这就涉及到存储器的管理技术。

存储器管理要解决的问题是逻辑地址到物理地址的转换以及相反的过程。逻辑地址是处理器输出的地址，而物理地址则是存储器中将要被访问的真实地址。在小型计算机系统中，逻辑地址和物理地址往往是相同的，换句话说，系统不需要地址变换，如图 6-34 所示。

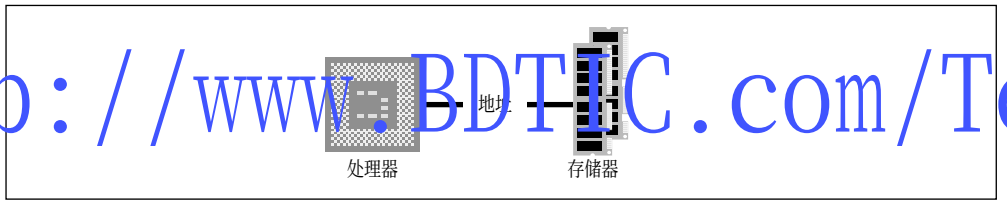

图 6-34：无地址变换

对小型计算机系统而言，没有存储器管理机制并不妨碍系统的运行。然而，在复杂的系统中，某种形式的存储器管理就可能很有必要了。以下四种情况就需要存储器管理：

物理存储器 > 逻辑存储器

当处理器的逻辑地址空间（由地址总线的个数决定）小于系统的物理地址空间时，处理器的逻辑空间必须被映射到系统的物理存储器空间之上。这有时被称为后备存储器（banked memory），听起来有些奇怪和陌生。通常，为了特定的目的而选择适用的处理器很必要，不过这种处理器也许地址空间有限，对于你的应用而言也许太小了。通过后备存储器的使用，处理器的地址空间会扩大到超过逻辑地址范围的限制。

逻辑存储器 > 物理存储器

当处理器的逻辑地址空间非常巨大时，用物理存储空间去填充有时是不切实际的。磁盘上的一些空间也许被用作虚拟存储器，这样处理器看起来就拥有比芯片内部更多的存储空间。存储管理的任务是：判断存储访问的存储器是物理存储器还是虚拟

存储器；将磁盘中虚拟存储空间上的内容转储到真正的内存中并完成相应的地址变换。

内存保护

你也许想阻止一些程序访问某些存储区域。保护机制能够阻止那些恶意代码去破坏操作系统，使计算机崩溃。它也是使所有的I/O访问都通过操作系统来进行的一种方式，因为保护机制可以阻止软件（操作系统除外）对I/O空间的访问。

任务隔离

在多任务系统中，任务间彼此不应该受到干扰（比如：越界访问其他任务的内存空间）。而且两个不同的任务应该能够使用相同的逻辑地址，可以通过内存管理机制将逻辑地址变换为各自的物理地址。

存储器管理的基本思想很简单，但实现起来却很复杂。而且存储器管理技术也很多，可以说有多少种使用存储器管理的计算机，就会有多少种存储器管理技术。

存储器管理由存储器管理单元（Memory Management Unit，MMU）实现。基本形式如图6-35所示。MMU可以是商用芯片，也可以是定制芯片（或逻辑电路），甚至还可以是处理器内部的一个集成模块。大多数现代的高速处理器几乎都将MMU集成在CPU芯片上。

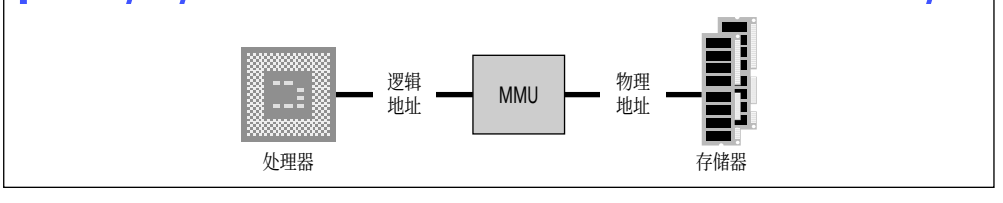

图6-35：使用MMU的地址转换

页面映射

在所有实用的存储器管理系统中，内存中的字按页分组，因而一个地址也可以看作是一个页号和字号的页内地址。MMU将逻辑页转换成物理页，而字号改变（如图6-36所示）。实际上，整个地址就是页号与字号的拼接。

来自处理器的逻辑地址被分为页号和字号。页号由MMU变换而得，再复合字号就可以形成存储器内的物理地址。如图6-37所示。

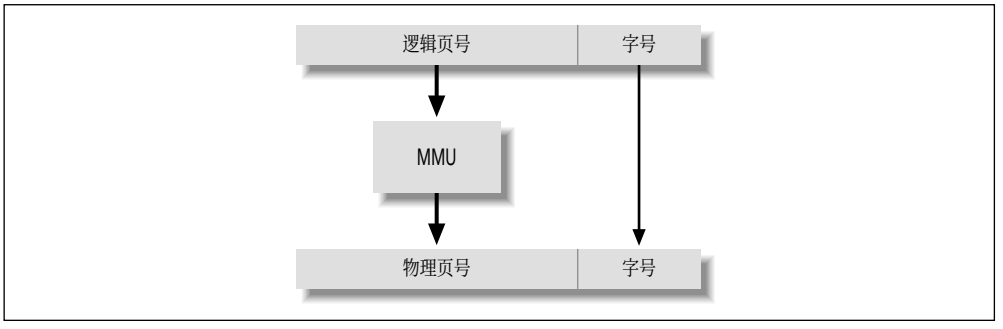

图 6-36：地址转换

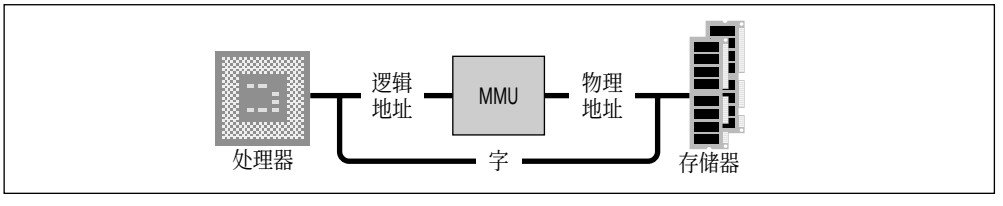

图 6-37：使用页内地址变换的系统

后备存储器（banked memory）

当逻辑地址空间小于物理地址空间时，存储器管理形式最为简单。如果系统将每一页的大小设计成等于逻辑地址空间，那么只要由 MMU 提供页号，就可以把逻辑地址映射为物理地址，如图 6-38 所示。

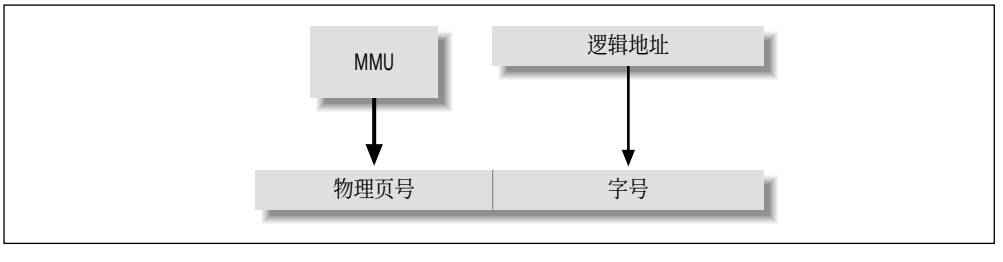

图 6-38：MMU 生成页号

图 6-39 所示的是用这种存储器管理方式的有效地址空间。逻辑地址空间可以被映射到（并重映射）到物理地址空间的任何地方。

图 6-40 所示的是这种存储器管理方式的系统配置。这种技术通常应用在 16 位地址（64K 逻辑空间）的处理器中，以便它们能访问更大的存储空间。

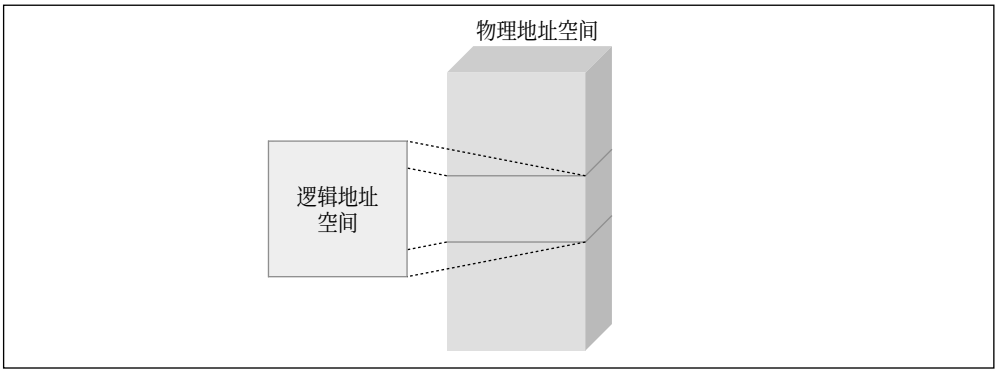

图 6-39：一个较小的逻辑地址空间到一个较大的物理地址空间的映射

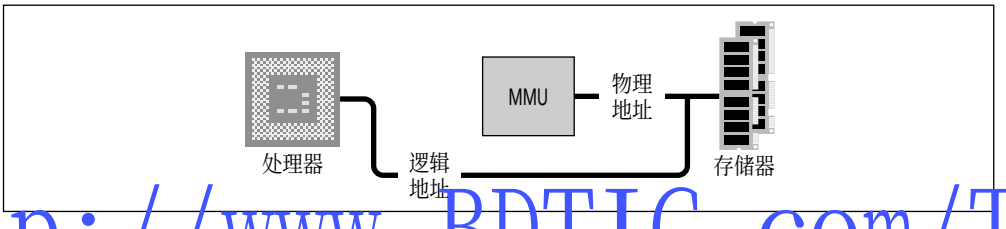

图 6-40：更大的物理地址空间的生成

对很多小型系统来说，后备存储也可以通过锁存（请求并占有）数据总线，将数据总线作为物理内存的附加地址位来实现，如图 6-41 所示。对于处理器的逻辑空间来说，锁存器可以被看作另一个 I/O 设备。为了选择合适的存储体，处理器将存储体比特位放到锁存器中锁存。所有对逻辑地址空间的访问都发生在被选择的存储体内。在本例中，处理器的地址空间作为一个 64K 的窗口以映射到更大的 RAM 芯片上。如你看到的那样，虽然存储管理看上去也许很复杂，但实现起来可以很简单。

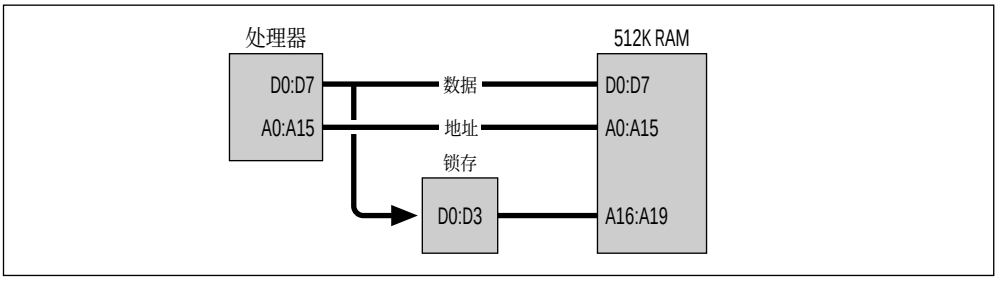

图 6-41：简单后备存储器的实现

注意：这种技术也被应用于桌面系统。老式的苹果机带有一个256K的内存，不过它的6502处理器的地址空间却只有64K。

图6-42所示的是在AT90S515 AVR系统中，实现后备存储器的布线要求，在本例中用一个512K的RAM代替了32K的RAM。

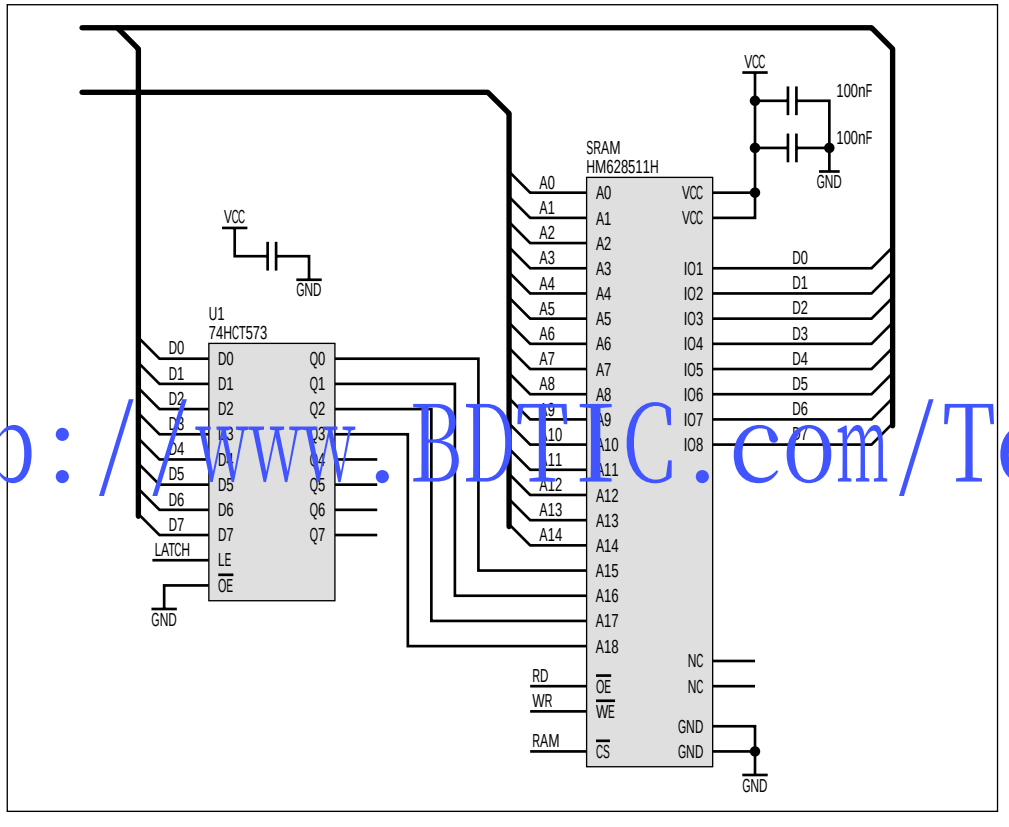

图6-42：AVR计算机的后备存储器

其中使用的RAM是由日立公司生产的HM628511H。在本例中，我们仍和以前一样将RAM分配在处理器的高32K地址空间。换句话说，处理器的高32K地址空间可以映射到RAM的512K地址空间。同样，处理器的低32K地址空间被留给I/O设备。地址位**A0**到**A14**仍被连到RAM上，而数据总线（**D0**～**D7**）则被连到SRAM的数据引脚（**IO1**到**IO8**，注7）。

注7： 由于这些引脚为设备完成数据输入/输出工作，存储芯片生产厂商经常标注数据引脚为I/O引脚。

现在，就像我们在应用LED锁存器时那样，我们也将74HCT573锁存器映射到处理器的地址空间。处理器可以写锁存器，并且锁存器将写入的数据保持在输出端。锁存器的低半字节被用来给 RAM 提供高阶地址位。

让我们看一下处理器访问地址 0x1C000 时的情况。地址 0x1C000 写成二进制形式为 %001 1100 0000 0000 0000。低 15 位地址（**A14**～**A0**）由处理器直接提供，而其余的地址位则必须被锁存。所以，处理器首先将字节0x03锁存，也即锁存RAM的**A18**、**A17**、**A16**和**A15**（见%0011，即0x03）地址引脚。RAM的这个区域现在被用作处理器的32K窗口。当处理器访问地址 0xC000 时，高阶地址位（**A15**）被地址解码器使用，以选择 RAM（通过使其片选信号 $\overline{\mathbf{CS}}$ 输出低电平）。而其余的 15 位地址位（**A0**～**A14**）再加上锁存器的输出即可选择地址 0x1C000。

NC 引脚的意思是无连接（No Connection），没有线路连接它。

地址变换

对于具有更大地址空间的处理器，MMU 可以提供地址总线上半部分的变换，如图 6-43 所示。

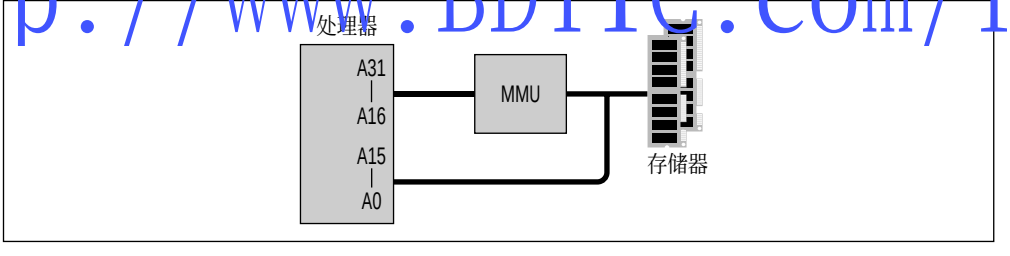

图 6-43：逻辑页号的变换

MMU 包含一个地址变换表，可以把输入地址重新映射到不同的输出地址上。为了改变地址变换表，处理器必须能够访问 MMU（如果地址变换表不能更改，那么 MMU 也就没有存在的必要了）。一些处理器被特别设计为可以与专门的外部MMU协同工作，而另一些处理器则把MMU集成在芯片内。但如果处理器在设计时没有考虑到要使用MMU，那么它就没有对 MMU 的特定支持。这样一来，处理器与 MMU 的通信方式就和其他的外设通信没有区别，同样要使用标准的读 / 写周期。这也就要求 MMU 必须出现在处理器的地址空间。看起来最简单的解决方法是将 MMU 映射到系统的物理地址空间中，遗憾的是，在实际中这并不可行。因为一旦（有意或无意地）MMU 不能映射到当前的逻辑地址空间（原因可能是MMU所在的物理页面不在当前逻辑地址空间的地址范围内），

那么 MMU 就不能够再被访问到。这也可能发生于系统上电时，因为这时 MMU 的地址转换表的内容还是未知的。

解决的办法是 MMU 的片选信号由处理器的逻辑地址总线直接解码。因此，MMU 将位于逻辑地址空间的一个恒定地址。不过，这种方法虽然避免了 MMU 的“丢失”，却引入了一个新的问题。由于 MMU 直接处于逻辑地址空间，那么不可避免的，它将受到突发性事件的影响（由系统崩溃所引起）或是在多任务的环境中受到一些非法的、恶意的干扰。为了解决这一问题，很多大的处理器都使用了两种状态的操作方式：管理态和用户态。每种模式分别使用了不同的堆栈指针。这就在操作系统（以及它的特权程序）和系统运行的其他任务之间形成了屏障。MMU 通过处理器上专用的状态引脚获知处理器的状态。只有当处理器处于管理态时，MMU 才能够被修改，这就限制了用户程序对其进行修改。对不同的工作态，MMU 使用不同的逻辑－物理变换表。管理态变换表通常在系统初始化时被配置，之后就保持不变。用户任务（用户程序）通常运行在用户态，而操作系统（用来执行任务切换和处理 I/O）则运行于管理态。中断也会把处理器置于管理态，这就使得向量表和服务例程可以不在用户的逻辑地址空间。而在用户态，操作系统可以使某些特殊的物理页不被用户程序所访问。