1.导言

Rabbit 半导体器件是专门为应用于中小型控制器而设计的一种高性能微处理器。Rabbit2000 是其首创产品，其设计者都有着多年在小型控制器中使用 Z80、Z180 和 HD64180 微处理器的经验。Rabbit 与这些微处理器有着相似的结构和高度的兼容性，而且与之相比又有重大的改进。

Rabbit 的开发设计是与 Z—World 公司共同合作完成的，Z—World 是一家长期从事低价位单扳机制造的公司,其产品的支持语言是一种改进的 C 语言开发系统(动态 C—Dynamic C)。Z—World 公司为 Rabbit 提供了软件开发工具。

Rabbit 使用简便，其硬件及软件界面都最大程度的实现了安全简洁，运算速度在 8 位总线微处理器中处于领先地位，这是因为源于 Z80 的指令集合非常简洁，而且存储器的接口设计允许最大限度的使用内存带宽。Rabbit 通过指令运行。

Rabbit 设计者从用户利益出发，简化了传统的微处理器软硬件开发方式。开发商重视 Rabbit 的开发方式，设计时就考虑了其开发不需要单片机开发系统。由一条电缆连接 PC 机串行口和基于 Rabbit 的目标系统完成软件开发。

1．1 特点及说明；

- 100 引脚 PQFP 封装。工作电压范围 2.7 V—5 V。时钟速度可达 30MHz。说明书将给出工业和商业用的适宜温度和电压范围。中等批量购进 Rabbit 微处理器单价低于$10。
- 工用级适用的电压波动为 10%，温度范围为-40°C到+85°C。商用级适用的电压波动为 5%，温度范围为 0°C到 70°C。
- 为 C 程序留有 1 兆字节的空间，最多可写 5 万多行程序代码。扩展的 Z80 式指令集合对于多数 C 操作来说简洁快速，并且对 C 语言是友好的。
- 具有四个级别的中断优先级，使得在实际工作中对关键应用能够做到快速响应。在时钟速度为 25MHz 的条件下，对于第一个中断程序指令的最大响应时间大约为 1μｓ。
- 访问 I/O 设备可通过使用带有 I/O 前缀的存储器存取指令来完成。因而，与专用 I/O 指令集的处理器相比，访问 I/O 设备更加快捷、简便。
- 系统硬件设计简单。总共可有 6 个静态存储器芯片（比如 RAM 和 flash EPROM）直接与微处理器连接而不需要额外的译码逻辑（glue logic)。通过使用并行 I/O 口线作为高位地址线，还可处理更多的存储器。每一次存储器访问需两个时钟周期。在 24MHz， Rabbit 处理器无等待状态下，存储器存取时间为 70ns。多数 I/O 设备也可实现无译码逻辑的直接连接。
- 存储器读取周期为两个时钟周期长度。清晰的存储器和 I/O 读写逻辑能够完全避免相互冲突的可能。外围 I/O 设备通常使用可编程接口作为 I/O 芯片选择信号、I/O 读选通信号或 I/O 写选通信号等来实现无译码逻辑接口。内置时钟倍频分频器允许使用降频方式工作以减少高频辐射。
- Rabbit 可经由串行口或并行口的访问实现冷启动。这一功能可以在未有任何现存程序或 bios（基本输入/输出系统）时能够重新编程。作为从处理器的 Rabbit 与易失性 RAM 能够在主处理器的冷程序启动下载程序的情况下正常工作。
- 共有 40 条并行 I/O 口线（与串行口共用)。其中一些 I/O 口是定时器同步的，这就允许在组合软硬件控制下精确地产生边沿和脉冲。
- 共有 4 个串行口。这 4 个串行端口都可以在多种操作模式下实现异步工作。其中两个口还可

以同步工作，实现与串行 I/O 设备的接口。通讯波特率可以很高，在异步方式时为时钟速度的 1/32，同步方式时为时钟速度的 1/8。在异步工作方式时，Rabbit 与 Z80CPU 相同，都支持发送标志字节来标记一个消息帧的开始。标志字节有 9 个数据而不是 8 个；额外的一位数据在前 8 个数据之后，用以标记一个消息帧的开始。

- 从端口方式允许 Rabbit 作为智能外设从属于一个主处理器。8 位从端口有 6 个 8 位寄存器，其中 3 个用于通讯的方向。另外的选通信号和中断信号从控制从端口双向工作。如果时钟信号和复位信号都与主处理器共用,那么只需要一个 Rabbit 和一个 RAM 芯片就可以构成一个完整的从系统。
- 内置电池供电的时间/日期时钟部件，使用一个外置 32.768KHz 晶振。时间/日期时钟也可用于提供每 488μs 一次的周期性中断。当使用推荐电池供电电路时，电池电流消耗仅为 25μA。还有另一种可选的电路能进一步减小这一电流。
- 很多定时器和计数器（共 6 个）可用于产生中断、波特率发生和计数器工作。
- 内置主时钟振荡器使用的是一个外部晶体，也可以使用陶制谐振器。典型的晶振频率范围在 1.8MHz 到 29.5MHz 之间。由于可通过单独的 32.768KHz 晶振实现精确定时，因此用价格便宜的陶瓷振荡器也能得到满意效果。系统时钟允许倍频或 8 分频，来动态动态改变工作速度或降低功耗。为定时器提供的时钟是独立的，以保证当处理器时钟分频或倍频时不影响波特率和定时器。极低功率方式时，处理器时钟可由 32.786KHz 振荡器驱动，并把主振荡器断电。这时电流大约为 100μA，而处理器仍能保持每秒 10，000 条指令的执行速度。这要优于其他处理器的休眠模式。在 25MHz/5V 时，电流大约为 65mA。电流、电压和时钟速度成比例——3.3V/7.68MHz 时电流为 13mA，而在频率为 1MHz 时电流降至小于 2mA。在极低功率模式下，应使用带有自动降低功率功能（通过 AMD）的 flash 存储器。
- Rabbit 有着卓越的浮点数处理性能，原因在于它有着严格的代码库和强大的处理能力。例如，25MHz 的时钟通常需要 14μs 进行浮点加运算，13μs 进行乘运算，40μs 进行开方运算。相比较而言，一个 8 位总线、25MHz 的 386EX 处理器使用 Borland C 的速度要慢十倍。
- 有一个内置的看门狗定时器。
- 拥有标准的十针编程端口，因而避免了使用 CPU 仿真器的必要。利用 Dynamic C 和 PC 的简单连接，可以使用一个非常简单的 10 脚连接器来下载和调试软件。而由于占用编程口所增加的成本极小。

图 1 所示为 Rabbit 微处理器的部件图。

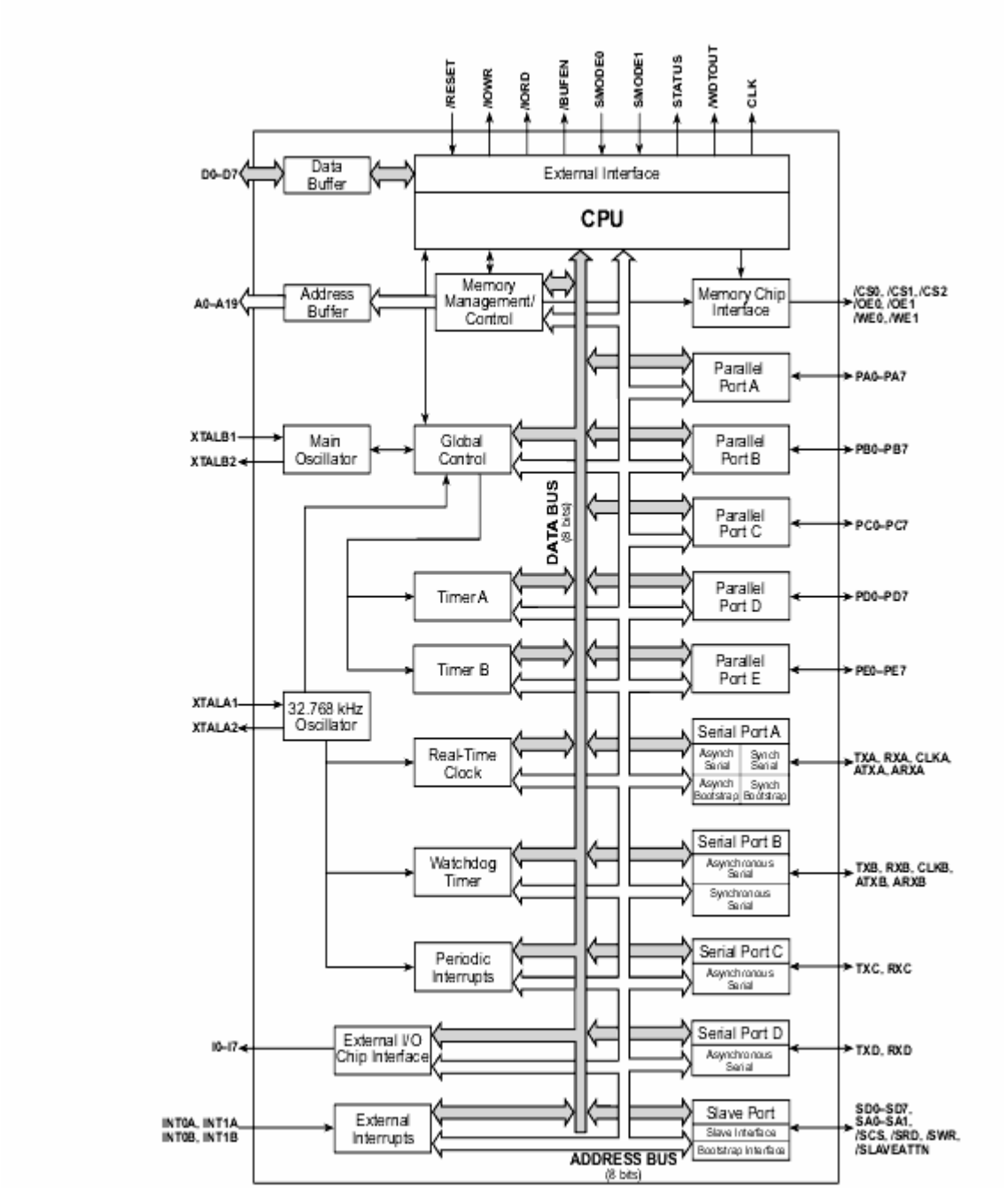


图 1.Rabbit 微处理器的部件图

1．2 Rabbit 优越性概述：

- **具有无译码逻辑的体系结构，使得硬件系统设计很简便。**
- **串行口数目众多并且通讯快捷。**
- **精确的脉冲宽度产生和准确时间隔输出脉冲沿。**
- **具有多种中断优先级。**
- **处理器速度和功率消耗可由程序控制。**
- **即使是在频率为 32KHz 的情况下，由于处理器仍能继续工作，超低功率模式也可以进行计算和逻辑测试。**

- Rabbit 可用于构成智能外围处理器或从处理器。例如，协议栈可卸载到一个 Rabbit 从处理器，并且主处理器可为任意处理器。
- Rabbit 能够实现冷启动，因而可以在适当的地方焊接未编程闪存（flash memory）。
- 可以编写长度为 1,000 到 50,000 行 C 代码的软件。Rabbit 提供所需工具而且费用低廉。
- 如果您熟悉 Z80 或 Z180，您就已经掌握了 Rabbit 的大部分功能。
- 使用一个 10 脚的编程接口来取代 CPU 仿真器及 PROM 编程器。
- 包含一个电池供电的时间/日期时钟。
- 标准 Rabbit 芯片按照工业温度和电压的规范而制作。

2 Rabbit 设计特点：

Rabbit 是一种改进 Z80 和 Z180 CPU 设计的 CPU，指令集和寄存器与 Z80 和 Z180 相同。指令集增添了许多新的指令。Rabbit 弃用了 Z180 中一些过时或多余的指令，将一些单字节操作码分配给了重要的新指令，使指令系统效率提高（参见“ Rabbit 与 Z80/Z180 指令区别 ”）。这一改进的优点在于熟悉 Z80 或 Z180 的用户能够立即对 Rabbit 有所了解。已有的源代码通过少量修改，进行重新汇编或编译就可为 Rabbit 所用。

技术进步使得 Z80/Z180 系列的某些特性已经过时，因而已被 Rabbit 所废弃。例如，Rabbit 不支持动态存储器，但完全支持静态存储器。这是因为静态存储器的已经非常便宜，使之成为了中型嵌入式系统的首选。Rabbit 不支持 DMA（直接存储器存取），原因在于传统的 DMA 使用不适用于嵌入式系统，或能够通过其他方法更好的完成，比如使用快速中断程序，外部状态机或从处理器。

从编写 C 编译器的经验可以看出，Z80 指令集执行 C 语言的缺点。主要问题在于缺少直接处理 16 位字指令和在所计算地址中访问数据的指令（特别是在堆栈中存存取数据时）。新的指令集解决了这些问题。

许多 8 位处理器所面临的另一个问题是执行速度慢并缺少数值整合（number-crunching）能力。在较小的系统中，良好的浮点运算能力是一项重要的特性。有了良好的浮点运算能力，就能更简便的解决编程问题。改进的 Rabbit 指令集合提供了快捷的浮点数及整数的运算能力。

Rabbit 支持四个级别的中断优先级。这是 Rabbit 的一项重要特性，这样在处理实时任务时就能有效的使用高速中断程序。

2.1 Rabbit8 位处理器与 16 位及 32 位处理器的比较

8 位处理器 Rabbit 有 8 位外部数据总线和 8 位内部数据总线。由于 Rabbit 构造了其大部分外部 8 位总线并且具有简洁的指令集合，因而其性能与许多 16 位处理器一样棒。所以 Rabbit 可执行许多 16 位操作。

我们不愿将 Rabbit 与 32 位处理器相比较，但毫无疑问的是，在有些情况下用户可以使用 Rabbit 来代替 32 位处理器并节省一大笔费用。很多 Rabbit 指令长度为 1 个字节。与之相比，绝大多数 32 位 RISC 处理器的最短指令为 32 位。

2.2 片内外围部件概述：

根据以往的经验，在小型嵌入式系统中选择了最有效的片内外围部件。片内部件主要有串行口，系统时钟，时间/日期振荡器和实时时钟，并行 I/O 口，从端口以及定时器等。其描述如下。

2.2.1 串行口

Rabbit 共有 4 路串行接口 A、B、C 和 D。这 4 个串行接口都可在异步模式下工作，波特率可达系统时钟的 1/32。异步端口能处理 7 位或 8 位数据，并且支持有一个标志位来标志地址帧和数据帧的 9 位通讯模式。用软件可判断出输出移位寄存器完成传送消息最后一个字节的时间，这弥补了 Z180 的一个明显缺点。这一改进对 RS—485 通讯非常重要，因为在最后一位字节发送完毕之前，线路驱动器不能转换传输方向。在许多 UART(通用异步收发器)，包括在 Z180 上的，都很难在最后一位字节发送完毕之后产生中断。Rabbit 不直接支持奇偶校验位及多个停止位，但可通过适当的驱动软件来完成。

在时钟同步串行模式下，可选择使用串行口 A 和 B。这时，由一个时钟信号同步传送输入或输出的数据。同步通讯两端的通讯器件中的任何一个都可用来提供时钟。当由 Rabbit 提供时钟时，波特率可达系统时钟频率的 1/4，或在 29.5MHz 时钟速度下超过 7,375,000bps。

串行口 A 具有特殊的性能，它可用于在复位后冷启动系统。串行口 A 也是 Dynamic C 软件开发的标准端口。

2.2.2 系统时钟

主振荡器使用一个频率范围为 1.8MHz 到 29.5MHz 的外部晶体。处理器时钟正是由这一振荡器驱动的，振荡器的输出或者直接使用 2 倍频，或使用 8 分频。在用于超低功率操作时，处理器时钟也可由 32.768kHz 振荡器驱动，此时主振荡器可在软件控制下关闭。

表 1 给出了根据操作功率选择时钟速度的初步估计。

表 1.选定时钟速度下操作功率的初步估计

时钟速度 (MHz)	电压 (V)	电流 (mA)	功率 (mW)	时钟速度 (MHz)	电压 (V)	电流 (mA)	功率 (mW)
25.0	5.0	80	400	6.0	2.5	10	25
12.5	5.0	40	200	3.0	2.5	5	12
12.5	3.3	26	87	1.5	2.5	2.5	6
6.0	3.3	13	42	0.0032	2.5	0.054	0.135

2.2.3 时间/日期振荡器

32.768kHz 振荡器可用于驱动外部 32.768KHz 石英晶体，以及由电池供电的内部 48 位计数器（有一个单独的供电电源引脚）。该计数器可用作实时时钟（RTC），由软件设置并读取，用来记录日期及时间。系统中有足够的空间来记录超过 100 年的日期。32.768kHz 晶体还可用于驱动看门狗定时器，或在冷启动顺序过程中为端口 A 提供波特时钟。

2.2.4 并行 I/O 口

从 5 个标记为 A 到 E 的 8 位端口中，共有 40 根并行输入/输出口线。大多数口线有后备功能，比如串行口或芯片选择选通信号。并行端口 D 和 E 有定时器同步输出的能力。输出寄存器是级联的。

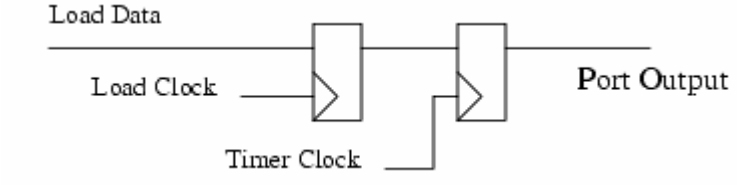


图 2．用于并行端口 D 和 E 的时钟同步输出寄存器

端口存储装载于一级寄存器之中。这一寄存器的内容根据所选择的定时器信号被依次传输到输出寄存器中。该定时器信号还能够产生一个中断，用它来建立下一位输出的时间。这一特性用于产生精确控制脉冲，其边沿定位具有很高的时间准确度。这种并行 I/O 主要应用于通讯信号发送、脉冲宽度调制及步进电机驱动。

2.2.5 从端口

这个功能是为了使 Rabbit 能够成为其它处理器的从处理器，主处理器也可以是另一个 Rabbit CPU。从端口与并行端口 A 共用，而且是双向数据传输端口。主端口可以读取三个寄存器中的任一个，通过两条选择线和读选通脉冲将寄存器内容输出到端口。从 Rabbit 可将这些寄存器作为 I/O 寄存器执行写操作。另有 3 个附加寄存器向相反方向传送数据。通过两条选择线和一个写选通脉冲可实现主处理器对它们的写操作。

图 3 所示为从端口中的数据通路。

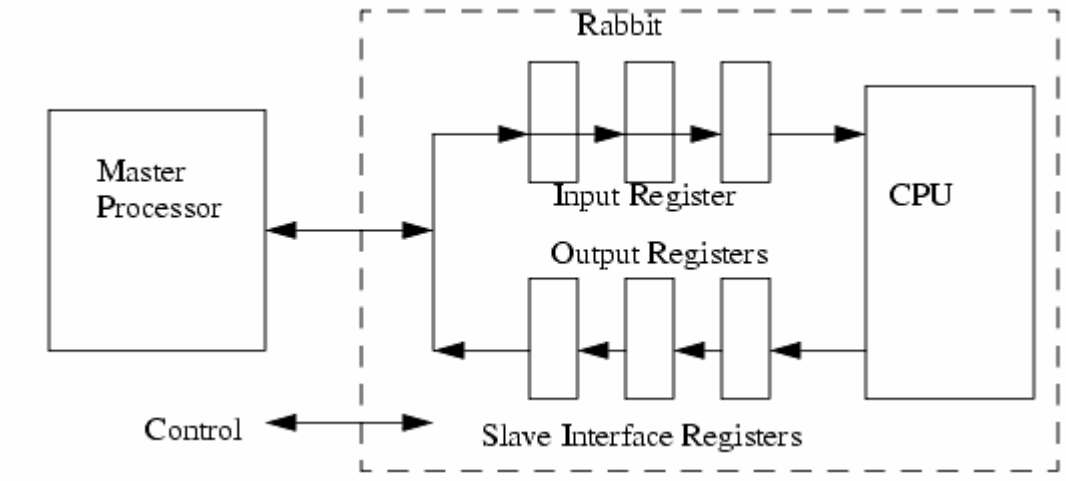


图 3.从端口中的数据通路

从 Rabbit 能够把同一寄存器作为 I/O 寄存器读取。当输入数据字节写入其中一个寄存器时，将由状态字节显示所写入的寄存器。用户可设计一个可选的中断，让它在写操作时发生。当从端口对向外携带数据的寄存器之一执行写操作时，使能一条注意信号线（attention line），因而主处理器可检测到数据变化并在需要时被中断。当从端口已读取全部输入数据之后，由一条线路通知主处理器。当新的输出数据就绪而主处理器还没有读取时，由另一条线路通知主处理器。从端口可在其上使用多种通讯协议指导主处理器执行任务。

2.2.6 定时器

Rabbit 有多个定时器系统。由一个 16 分频的 32.768KHz 晶体生成周期性中断，如果被使能，能够实现每 488μs 产生一个中断，可用作普遍用途的时钟中断。定时器 A 由五个 8 位递减计数且可重复装载的寄存器双层级联组成。并且每个递减寄存器都可设置以被 1 到 256 中任意一个数分割。其中 4 个定时器的输出用于为串行端口提供波特时钟。其中任何一个寄存器都可产生中断或为时钟同步的并行输

出端口计时。定时器 B 是一个只可读不可写的 10 位计数器。共有两个 10 位匹配寄存器和比较器。如果匹配寄存器与计数器相匹配则输出一个脉冲。这样就可对计数器编程以实现当计数到预先设定数时输出脉冲。该脉冲可用于为定时器同步的并行端口输出寄存器计时，也可用于产生中断。在程序控制下，用定时器 B 在将来某个精确时间产生事件（event）很方便。

图 4 所示为 Rabbit 定时器。

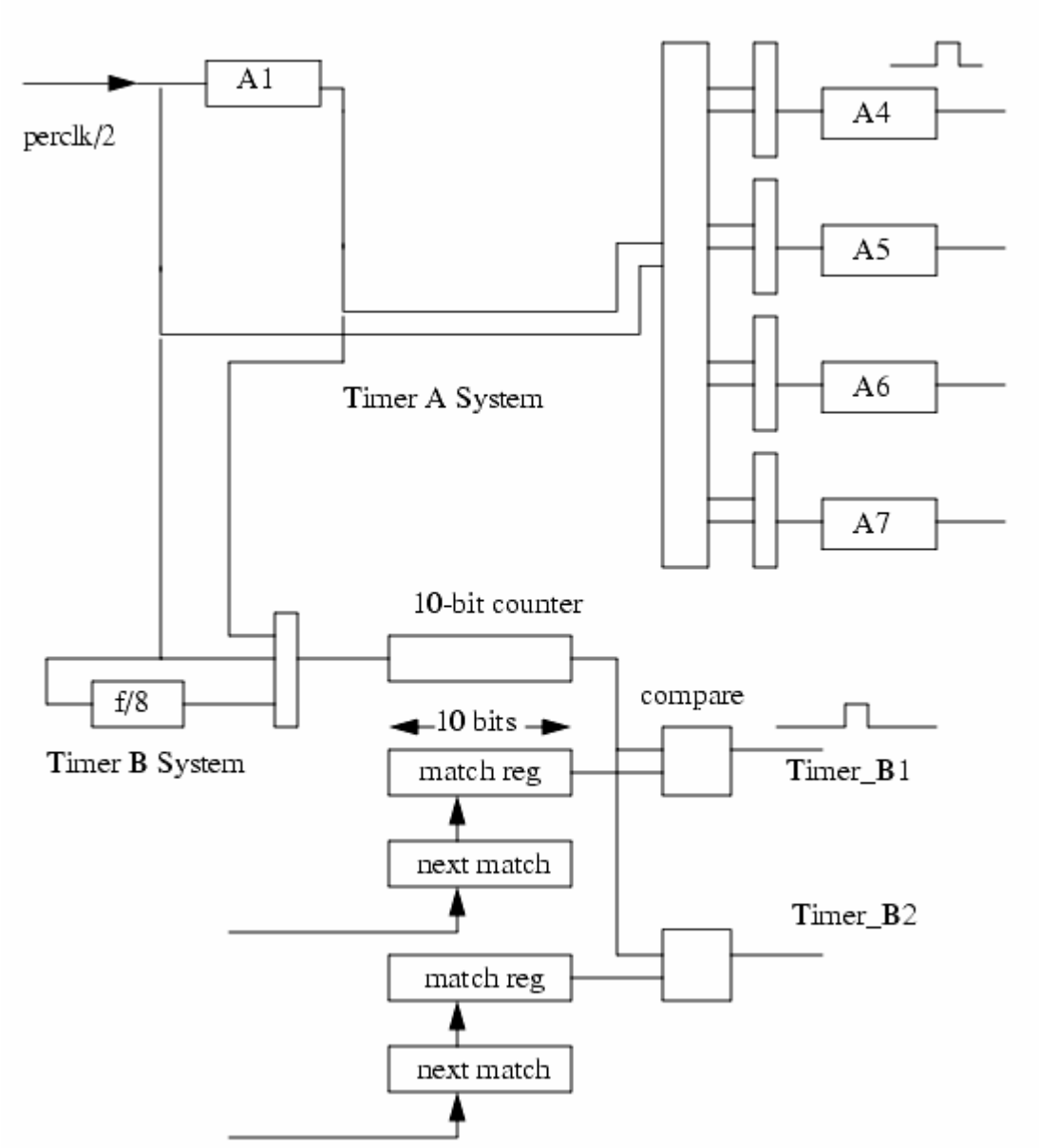


图 4.Rabbit 定时器

2.3 设计标准

用 Rabbit 可用多种方法实现同一种功能。通过公布设计标准或实现通常目的的标准方法，可以使软硬件支持更为简易。

2.3．1 可编程端口

Rabbit 半导体公司公布了一项标准可编程端口规范（参见：“ Rabbit 可编程端口 ”）并提供了一种转换电缆，该电缆可用于连接 PC 串口与标准可编程接口。用一个在 2mm 中心有两排引脚的连接器实现这个接口。该端口可连接到 Rabbit 串行端口 A 上，Rabbit 启动模式引脚上，Rabbit 复位引脚上，或产生要求 PC 注意的信号的可编程引脚上。通过在设计及软件上使用适当的防范措施，串行端口 A 即可被用作编程端口，又可作为用户自定义的串行端口，然而在大多数情况下并不需要这种功能。

Rabbit 半导体支持标准编程端口和标准编程电缆，用作诊断故障的诊断和配置或现场的安装系统。

2．3．2 标准 BIOS(基本输入输出系统)

Rabbit 半导体公司为 Rabbit 提供了一套标准 BIOS。该 BIOS 是基本输入输出程序，可管理、启动、关闭，并为运行在 Rabbit 上的软件提供基本服务。

2．4 Rabbit 的动态 C 软件支持

Dynamic C 是 Z—World 公司的交互式 C 语言开发系统，可在安装有 Windows95/98 或 Windows NT 的 PC 机上运行。它提供了集成开发环境，集编译器，编辑器和调试器于一体。调试基于 Rabbit 的目标系统的一般方法是用标准转换电缆连接 PC 机串行口到目标的 10 脚编程接口。Dynamic C 的程序库包含有高度完善的软件为 Rabbit 提供应用支持，其中包括驱动程序，应用程序，数学运算程序及动态 C 调试 BIOS 的程序。

另外，国际知名的实时操作系统 uC/OS-II 也将移植到 Rabbit 系统，并带有动态 C 的某些版本。

3．Rabbit 微处理器功能详细介绍

3．1 处理器的寄存器

Rabbit 寄存器与 Z80 或 Z180 寄存器十分相似。图 5 所示即为 Rabbit 寄存器的分布图。其中 XPC 及 IP 均为新寄存器。而 EIR 与 Z80 的 I 寄存器相同，用于指向一个外部中断的中断向量表。IIR 寄存器与 Z80 的 R 寄存器在指令集中占据相同的逻辑位置，但 IIR 寄存器的功能是指向一个内部中断的中断向量表。

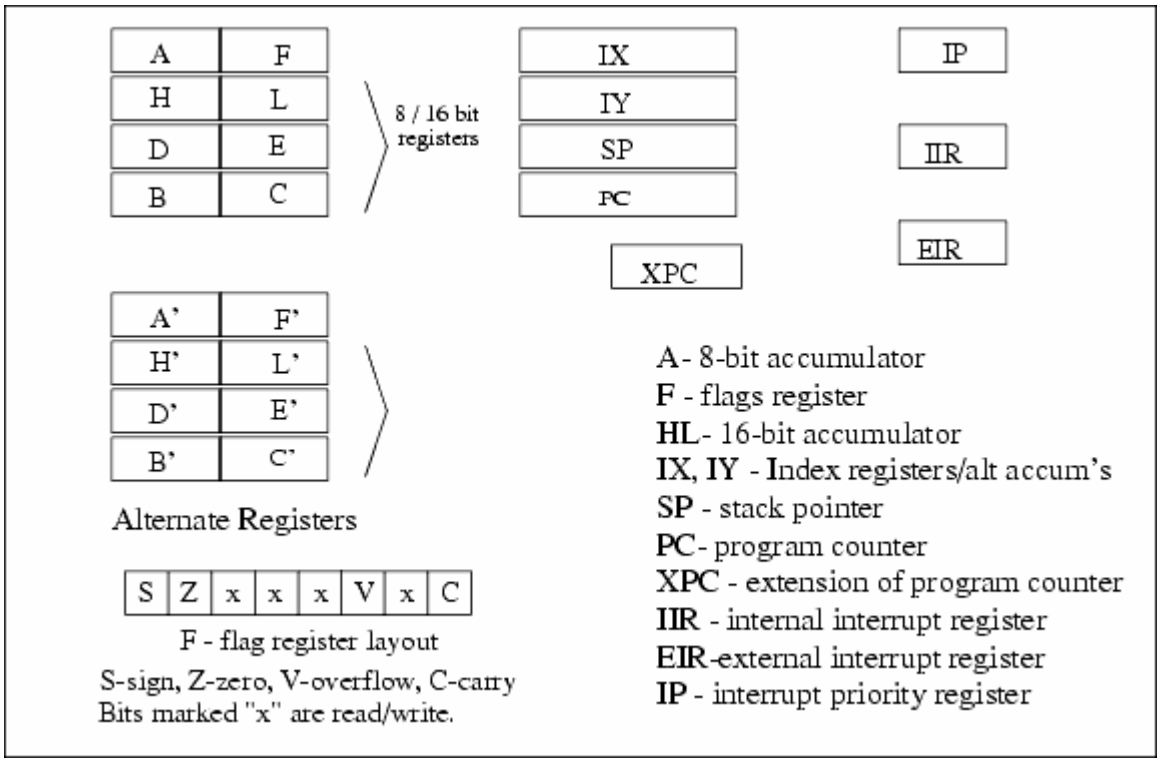


图 5.Rabbit 寄存器

Rabbit（以及 Z80/Z180）处理器有两个累加器，寄存器 A 作为 8 位累加器执行 8 位运算，如 ADD 或 AND；16 位寄存器 HL 用作 16 位累加器执行 16 位运算，如 ADD HL，DE，实现将 16 位寄存器 DE 与 16 位累加器 HL 相加。在许多操作中也可由 IX 或 IY 代替 HL 作为累加器。

F 寄存器是标志寄存器（flag register）或状态寄存器，存储着一些最新操作信息的标志。标志寄存器只能通过 POP AF 和 PUSH AF 指令来进行操作，而不能直接访问。通常条件跳转指令测试这些标志。这些标志用于纪录算术或逻辑运算的结果。在标志寄存器中有 4 个未用的读/写位，用户可使用 PUSH AF 和 POP AF 对其进行操作。但对于这些读/写位的使用应十分小心，因为新一代 Rabbit 处理器将为他们定义了新的功能。

寄存器 IX、IY 及 HL 还可用作索引寄存器（index register）。它们用于指向所存取的数据的内存地址。虽然 Rabbit 可寻址 1 兆或更多的内存地址，但索引寄存器只能直接寻址 64K（除非使用特定的扩展寻址指令 LDP）。寻址范围可通过用内存地址映射物理地址（详见“内存映射”）和特殊指令来实现扩展。对于大多数嵌入式系统来讲，64K 数据存储器（相对于代码存储器）已经足够了。Rabbit 可有效使用 1 兆字节的存储空间。

SP 寄存器指向堆栈，此堆栈用于子程序和中断程序的断点纪录和数据通讯以及一般用途的数据存储。

Rabbit（以及 Z80/Z180）有辅助寄存器（alternate registers）集。两条特殊的指令可实现常规寄存器和辅助寄存器的交换，指令 ex af,af′交换了 AF 和 AF′的内容，指令 EXX 则将 HL、DE、BC 与 HL′DE′BC′作交换。在原来的 Z80 体系结构中，常规寄存器和辅助寄存器之间的通讯比较困难，原因在于交换指令是实现这一通讯的唯一方法。新型 Rabbit 指令大大改进了常规寄存器和辅助寄存器之间的通讯，相当于将寄存器数量增加了一倍，方便了程序员的使用。这并不意味着要将这些辅助寄存器作为另外一个寄存器组来在中断程序中使用，而且动态 C 也不支持这种使用，因为在任何时候都可随

意的同时使用两套寄存器。

寄存器 IP 是中断优先权寄存器。它包括四个 2 位域，用来存储处理器中断优先权的历史信息。Rabbit 支持的 4 级中断优先权，有的只在某些 Z80 或 Z180 的受限制的地方才使用。

3．2 存储映射

除了一些特殊指令外（参见："16 位装载并存储 20 位地址"），Rabbit 指令可直接寻址 64K 的数据存储空间。就是说 Rabbit 指令的地址域的长度是 16 位，并且可用作地址的指针（索引寄存器（IX,IY），程序计数器以及堆栈指针（SP））的寄存器的长度也是 16 位。

由于 Rabbit 使用的是 16 位地址，因而与使用 32 位地址相比，指令更短并且执行更快。同样程序代码也是非常简洁的。尽管这些 16 位地址有它的实用性，但也确实产生了一定的复杂度，原因在于必须有内存映射单元才能使 C 程序访问合理数量的存储空间。

Rabbit 与 Z180 的存储映射单元十分相似，但功能更强大。图 6 所示为与内存编址相关的主要部件之间的关系。

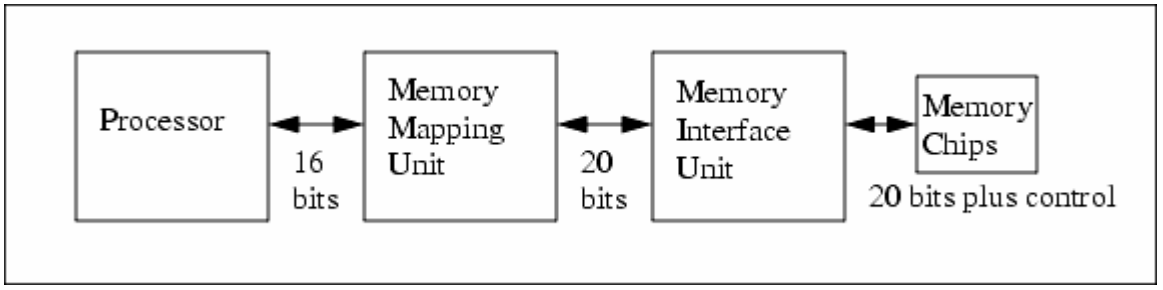


图 6.寻址存储部件

存储映射单元接收 16 位地址输入，输出 20 位地址。处理器涉及的空间是 16 位（特定的 LDP 指令除外），它仅能直接寻址 65536 个字节，在这一范围内其指令有效。由 3 个段寄存器来实现这 16 位空间与 1 兆空间之间的映射。这个 16 位空间将被分为 4 个分离的区域。除了首尾区域外，每一个区域都有一个段寄存器，该段寄存器被加入到各区域的 16 位地址中以产生 20 位地址。每个段寄存器有 8 位，这 8 位被加入到 16 位地址的前 4 位上，就形成了一个 20 位的地址。这样，在 16 位存储器中的每一个分离区域就都成为了映射 20 位地址空间中一段内存的窗口。16 位空间中 4 个段的相对大小由 SEGSIZE 寄存器控制。SEGSIZE 寄存器是一个包含有两个 4 位寄存器的 8 位寄存器。由它控制着第一段与第二段以及第二段与第三段之间的边界。另两个可移动段边界的位置由一个 4 位的数值决定，这个数值指明了边界所在位置地址的前 4 位。这一关系如图 7 所示。

各段的名称说明了该段的用途。根段用于映射 flash 存储器的基址并包含有启动代码和其它偶尔存储进来的代码。数据段的用途依照建立内存的策略的不同而改变，可能是根段的扩充，也可能包含数据变量。堆栈段的长度通常是 4K，用于保存系统堆栈。XPC 段通常用于执行不存储在根段或数据段中的代码。有特殊的指令用于执行 XPC 段中的代码。

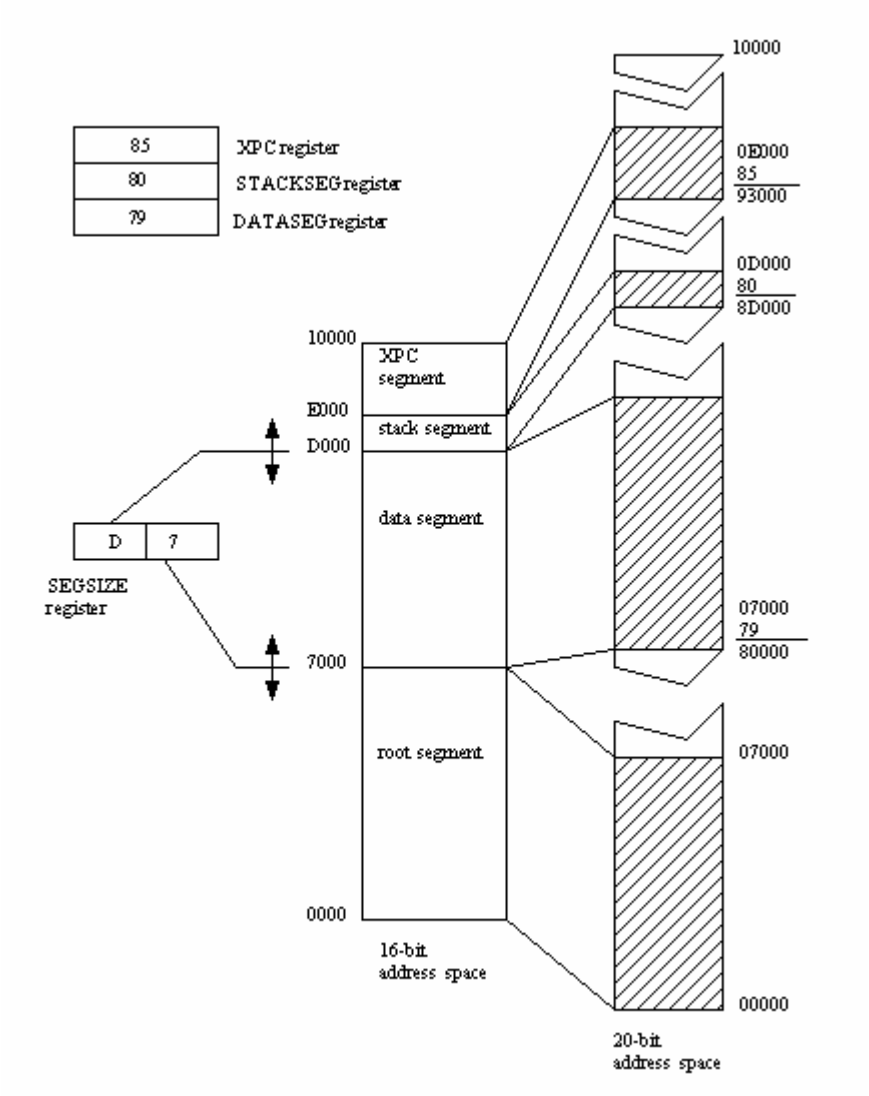


图 7．内存映射操作实例

存储器接口单元接收由存储器映射单元产生的 20 位地址。存储器接口单元有条件地修改地址线 A16、A18 和 A19。其它地址线无条件传递。存储器接口单元为外部存储芯片提供控制信号。这些信号包括片选信号（/CS0，/CS1，/CS2）、输出使能信号（/OE0/OE1）和写使能信号（/WE0/WE1）。基于静态存储器芯片的控制序列与这些信号相对应（片选或/CS、输出使能或/OE 以及写使能或/WE）。为了产生这些存储器控制信号，将 20 位地址空间分为 4 个象限，各 256K。每一象限中都有一个体控制寄存器（bank control register），用来决定发生内存读写操作时的哪些片选、输出使能和写使能有效。例如，如果一个 512K*8 的 flash 存储器在 20 位地址空间中作为前 512K 地址被访问，可使前两象限的/CS0、/WE0 和/OE0 使能。

图 8 所示为一个存储接口单元。

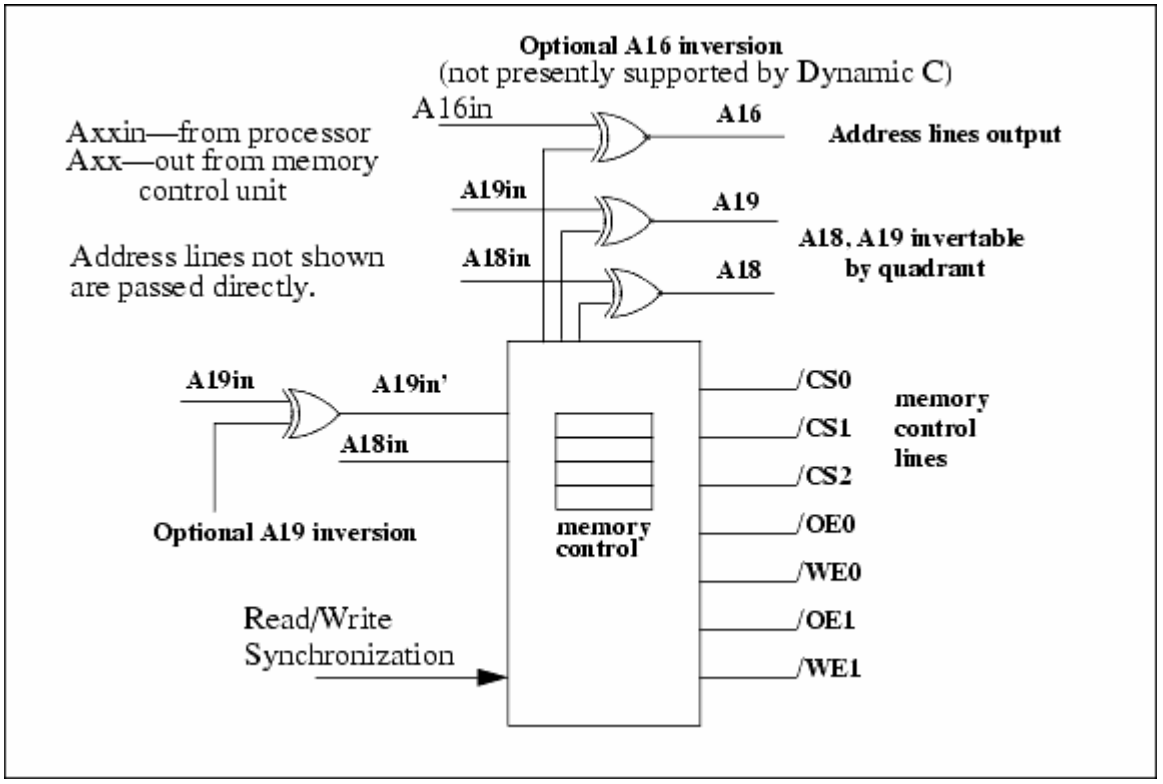


图 8.存储器接口单元

3．2．1 扩展代码空间

Rabbit 能够有效的执行代码最长达 1 兆字节的程序，这是其内存映射设计中至关重要的一项功能，也是单纯的 16 位地址处理器所做不到的，即使使用 Z180 内存映射单元也较难实现。对于页式处理器（如 8086），则在代码空间中进行页面分离，使代码存储在许多分离的页面之中，而实现这一功能。8086 的页面大小为 64K，因此在给定页面上只使用 16 位地址进行跳转、调用和返回就可以访问所有代码。在使用分页技术时，由另外一个独立寄存器（在 8086 中使用 CS 寄存器）决定当前活动页在整个存储空间中的位置。通过使用特殊指令可实现页面之间的跳转、调用或返回。这些特殊指令称为长跳转、长调用和长返回，用以区别仅用于 16 位变量的相同操作。

Rabbit 还使用分页技术来实现 16 位地址可达的空间之外的代码空间的扩展。Rabbit 分页技术引入了滑动页面的概念，其长度通常为 8K，也就是 XPC 段。8 位 XPC 作为页面寄存器指明窗口所指向的内存部分。当在 XPC 段中执行程序时，窗口内部的大多数跳转使用通常的 16 位跳转、调用和返回。这些通常的 16 位跳转、调用和返回也用于访问 16 位地址空间里其余三个段中的代码。如果需要对窗口以外的代码进行传输控制，就要使用长跳转、长调用及长返回了。这些指令将修改程序计数器（PC）和 XPC 寄存器，使得 XPC 窗口指到内存中的另一个不同的位置上，也就是长跳转、长调用或长返回的目标位置上。XPC 寄存器的长度一直为 8K。将 XPC 段置于内存中的间隔尺寸为 4K。由于窗口能在大小为其自身尺寸一半的范围内滑动，因此能够在没有非占用内存空隙的情况下实现连续编译。

由于编译器产生的代码驻留在 XPC 窗口里，当代码长度超过 F000 时，窗口下滑 4K。这是通过一个长跳转将窗口重新定位使其降低 4K 来实现的。如图 9 所示。编译器在页面的结尾看不到明显的边界，这是因为当代码长度超过 F000 时窗口空间还有剩余，除非在窗口下滑之前又加进了 4K 代码。XPC 窗口中编译的代码都有 24 位地址，其中包括 8 位 XPC 和 16 位地址。假如源指令及目标指令具有相同的 XPC 地址，则可使用短跳转和短调用。通常情况下，这意味着每一条指令都属于某一个窗口，这个窗口的

长度大约是 4K，其地址范围在 E000+n 到 F000+m 之间，这里 m 和 n 的长度可为几十个字节，但最长可达 4096 个字节。由于窗口被限制在 8K 以内，因此编译器不能编译需要 8K 以上左右代码空间的单个表达式。这其实不在实际的考虑范围之内，因为超过几百字节的表达式的性质与特技一样，根本就不是什么实用程序.

程序代码可存放在根段或 XPC 段中，也可存放在数据段中。代码可以在堆栈段中执行，但一般仅限于特殊情况下。根段中的代码，可通过短跳转和短调用实现对根段中其他代码的调用，这同时也意味着除 XPC 段之外的任何段一般都可以这么做。XPC 段这么做则不好。XPC 段的代码，虽然也可通过短跳转和短调用来调用根段中代码，然而，调用 XPC 段中的代码却必须使用长调用。最好能将函数放入根段中，这是因为长调用及长返回的执行需要 32 个时钟，而短调用及短返回的执行仅需要 20 个时钟。这一区别虽然很小，但对于简短的子程序非常重要。

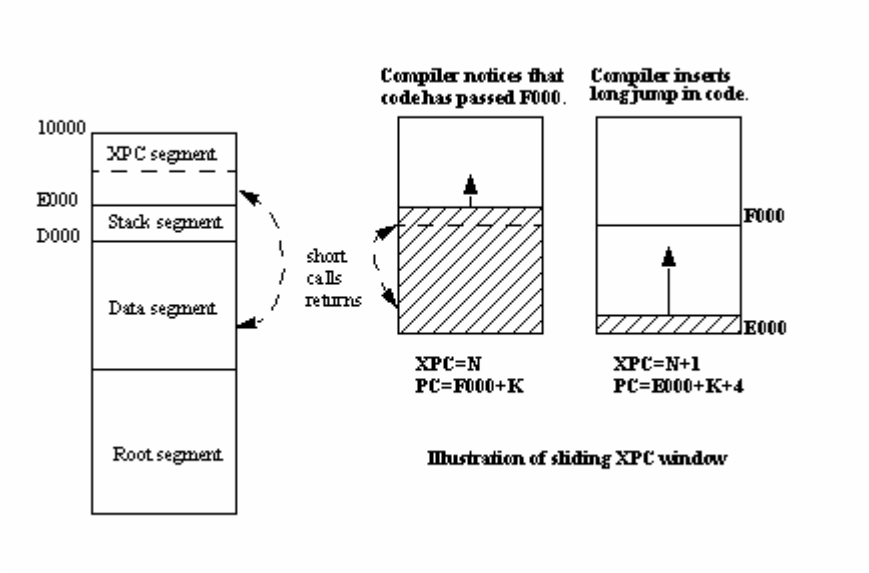


图 9.XPC 段的使用

3．2．2 数据存储扩展

在通常的内存模型中，数据空间必须与根代码、堆栈和 XPC 窗口共享 64K 的空间。通常情况下，这种做法留出了 40K 或更少的潜在数据空间。XPC 窗口需要 8K，堆栈需要 4K，而且大多数系统都需要至少 12K 的根代码。对于绝大多数嵌入式应用来讲，这些数据空间足够了。

如果要获得更多的数据空间，可以将数据存入没有映射到 64K 空间里的 RAM 或闪存中，然后可利用函数调用访问这些数据，或在汇编语言里时用 LDP 命令，该命令可使用 20 位地址来访问内存。使用这些方法访问简单的数据结构或缓存效果较好。

扩充数据内存的另一种方法是使用堆栈段来访问数据，将堆栈设置在数据段中以释放堆栈段。这种方法适用于一些软件系统，这些软件系统的特点是使用自包含的数据编组，并且一次只能访问其中的一个分组而不是从中随机选取。一个例子就是与 TCP/IP 通信协议连接相关联的软件结构，同一代码访问每个连接关联的数据结构时，方式由每个连接上的通信量决定。

这种方法的优点是能够使用常规的 C 数据访问方法，例如 16 位指针。在使用数据结构之前，必须修改堆栈段寄存器并将该数据结构引入堆栈段中。由于必须把堆栈移入数据域中，所以当使用堆栈段查看

数据时，将所需堆栈的数目减小到最少是十分重要的。当然，如果不需要查看数据结构，可把堆栈保留在堆栈段中。除此之外，还有另外一种可行的方法，就是将数据结构和堆栈同时放在堆栈段中，不同的任务使用不同的堆栈段，而每个任务都有绑定的自身的数据域和堆栈。

这种方法如图 10 所示。

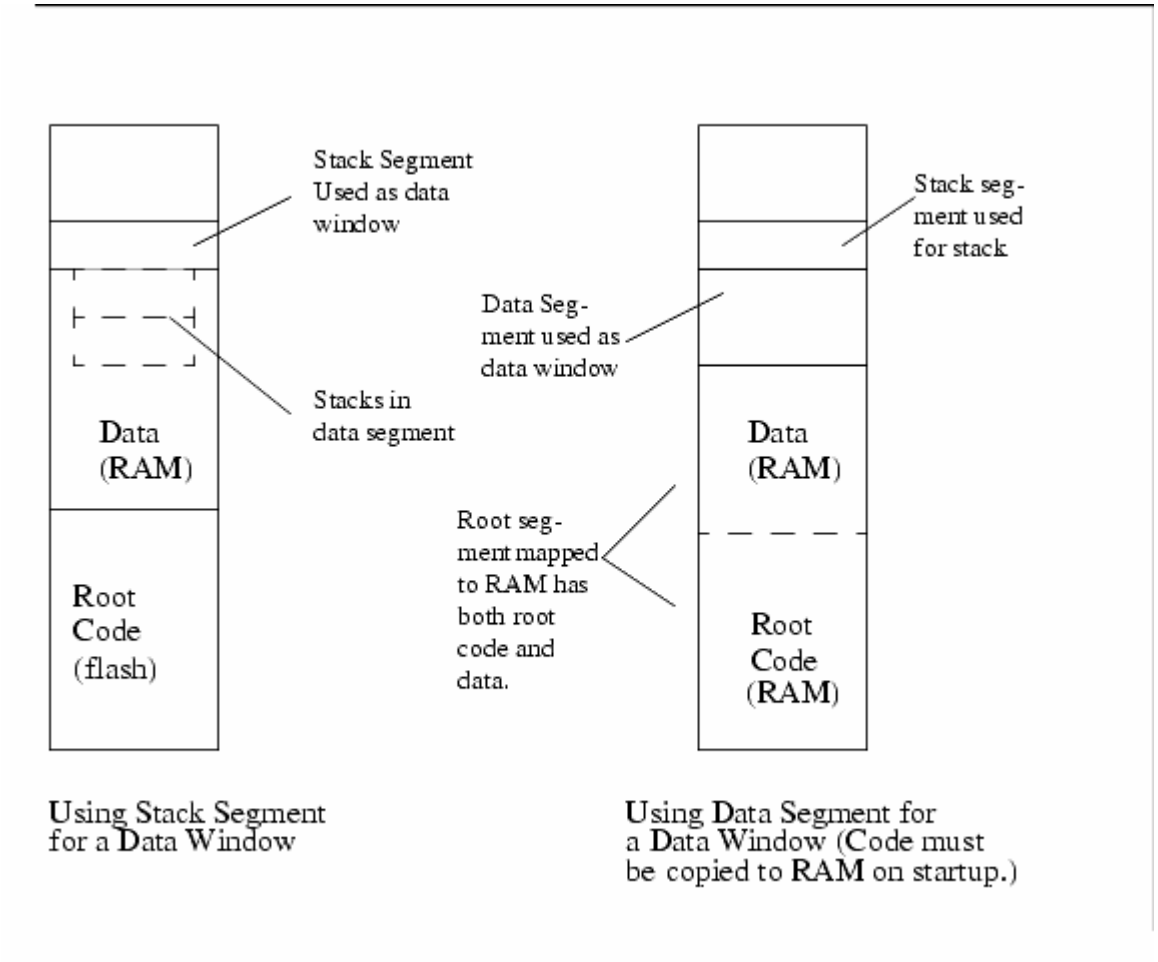


图 10.数据内存窗口方案

第三种方法是将 RAM 中的数据和根代码都放入根段中，释放数据段而作为扩展内存的一个窗口。这种方法要求在启动时将根代码拷贝到 RAM 中。将根代码拷贝到 RAM 中不一定很繁琐，因为所要求的 RAM 很少，比如说 12K。

通过程序设计可将位于内存顶端的 XPC 段也用作数据段，要求这些程序被编译进入根内存。需要访问大量数据的小型程序使用这种方法十分方便。

3．2．3 实际中的存储器使用

结构最简单的 Rabbit 由一个 flash 存储芯片（用/CS0 接口）和一个 RAM 存储芯片（用/CS1 接口）组成。在实际应用中，最小的 flash 容量是 128K，RAM 容量是 32K。Rabbit 也可支持更小型的芯片，但这些小型的静态存储器已经过时，因此没有提供支持。

虽然 Rabbit 所支持的代码大小可达 1 兆，但在大多数应用中最好少于 250K，相当于 10,000-20,000 条 C 语句。这不但反映了 Rabbit 代码的精简，也符合多数嵌入式应用的大小。

Rabbit 限制能直接访问 C 变量的为大约 44K 内存，在存储在 flash 和 RAM 之间的数据之间分离出来。对大多数嵌入式应用来说，这些内存空间已经足够了。也许有些应用需要额外的数据内存来存放庞大的数组或列表，因此动态 C 提供了对扩展数据内存类型的支持，它扩展额外数据内存数量到远远大于 1 兆。

是否需要堆栈内存取决于应用的类型，特别是是否使用了抢先多任务。如果使用了抢先多任务，则每项任务都需要有自己的堆栈。由于堆栈在 16 位地址空间中有其自己的段，因此可以很方便的使用 RAM 存储器来实现大量的堆栈。当环境的抢先变化发生时，STACKSEG 寄存器将做相应的改变，以把堆栈段映射到 RAM 的某个部分，这个部分的 RAM 里包含将要运行的新任务的关联堆栈。通常堆栈段的大小为 4K，足够为几个堆栈（一般为 4 个）提供空间。当需要大于 4K 的堆栈时，也可以扩展堆栈段。如果只需要一个堆栈时,可以删去整个堆栈段,并将这个单独的堆栈放入数据段。这种方法适用于仅有 32K RAM 并且不需要多个堆栈的系统。

3．3 指令集合纵览

这一部分主要介绍了以下指令：

- 立即数载入寄存器指令
- 在常量地址中载入或存储数据指令
- 使用索引寄存器载入或存储数据指令
- 寄存器之间的传送指令
- 寄存器交换指令
- Push 和 Pop 指令
- 16 位算术及逻辑操作
- 输入/输出指令——这包括对一个 BUG 的修正。当一条 I/O 指令（前缀为 IOI 或 IOE）后跟随 12 条单字节操作码（用 HL 作为索引寄存器）中的一条时，这个 BUG 就会发生。

在以下的讨论中，我们将给出各类指令的例子，并将 Rabbit 与 Z80/Z180 的指令作比较。每条指令的详细说明请参见“ Rabbit 指令 ”。

Rabbit 执行指令所需的时钟数目少于 Z80 或 Z180。通常情况下，Z180 执行 1 字节的操作码最少需要 4 个时钟，执行多字节操作码时每个字节需要 3 个时钟。另外，每个数据字节的读或写还需要 3 个时钟。许多 Z180 指令还需要很多的额外时钟。而 Rabbit，执行操作码的一个字节和读取数据的一个字节一般需要 2 个时钟，写一个字节则需要 3 个时钟。当需要计算一个存储地址或使用索引寄存器寻址时，还需要一个额外时钟。只有少数指令不遵循这些原则，比如 mul——16*16 位有符号二进制补码乘法，是一条 1 字节操作码，但在执行时需要 12 个时钟。与 Z180 相比，Rabbit 不仅需要的时钟较少，而且通常其时钟速度更快，指令功能更强大。

与 Z180 相比，Rabbit 指令集合的主要改进如下：

- 提取和存储数据（特别是 16 位字）相关于堆栈指针或索引寄存器 IX、IY 及 HL。
- 16 位算术及逻辑操作，包括 16 位与、或、移位和 16 位乘。
- 使用新指令后，常规寄存器与后备寄存器、索引寄存器与常规寄存器之间的通讯更为方便了。在 Z180 中，辅助寄存器的使用比较困难，而 Rabbit 中辅助寄存器与常规寄存器集成的很好。
- 长调用、长返回以及长跳转方便了 1M 代码空间的使用。这就避免了 Z180 中对超过 64K 代码的较长程序使用低效存储方案（banking schemes）。

- 现在通过普通存储访问指令就可实现输入/输出，这些指令带有操作码字节前缀，用以显示所访问的 I/O 空间。共有两个 I/O 空间：内部 I/O 和外部 I/O。

Rabbit 删除且不支持一些 Z80 及 Z180 指令（参见：19 小节“ Rabbit 与 Z80/Z180 的指令区别 ”）。大多数已被删除的指令十分陈旧或是用处很小并能由一些 Rabbit 指令模拟。同时也有必要删除一些指令来释放硬件空间，以保证有效的执行新指令。这些指令不再实现为 2 字节操作码，目的在于避免次要指令浪费片内资源。除了 EX (SP),HL 指令外，所有保留下来的 Z180 指令的原始 Z180 操作码的二进制编码都予以保留。

3．3．1 立即数载入寄存器指令

通常，指令组中位于操作指令之后的常数可以载入除 PC、AF、IP 和 F 以外的任何寄存器（载入 PC 寄存器需使用跳转指令）。其中也包括 Rabbit 中的辅助寄存器，但不包括 Z180 辅助寄存器。下面给出一些指令示例。

```
LD A,3
LD HL,456
LD BC',3567   ; 不可用于 Z180
LD H',4Ah     ; 不可用于 Z180
LD IX,1234
LD C,54
```

执行向寄存器传数操作时，装载一个字节需要 4 个时钟，一个字需要 6 个时钟。载入 IX、IY 存储器或备用存储器时，通常还需要两个额外时钟，这是因为操作码还包括一个 1 字节的前缀。

3．3．2 向（从）直接地址单元写入或读取数据指令

```
LD A,(mn)     ; 从地址 mn 中读取 8 位数据
LD A',(mn)    ; 不可用于 Z180
LD (mn),A
LD HL,(mn)    ; 从地址 mn 中读取 16 位数据
LD HL',(mn)   ; 对辅助寄存器操作，不可用于 Z180
LD (mn),HL
```

除了对 DE、BC、SP、IX 和 IY 以外，16 位的存取与之相似。

可以将直接地址存储器中的数据读入辅助寄存器，但不允许将辅助寄存器中的数据直接存储到存储器中。

```
LD A',(mn)      ; 可行
** LD (mn),D'   ; ****非法指令！
** LD (mn),DE' ; **** 非法指令!
```

3．3．3 使用索引寄存器存取数据指令

索引寄存器 IX、IY、SP 或 HL 是 16 位寄存器，用于指示存取存储器的字节或字的地址。有时还会给这个地址加上一个 8 位有符号数或无符号数的偏移量。在这个指令中，8 位的偏移量是一个字节。BC 和

DE 只有在以下的特殊情况下才能用作索引寄存器。

```
LD A,(BC)
LD A',(BC)
LD (BC),A
LD A,(DE)
LD A',(DE)
LD (DE),A
```

其它的 8 位载入和存储指令如下：

```
LD r,(HL)      ; r 是寄存器 A，B，C，D，E，H，L 中的任意一个
LD r',(HL)     ; 同上，但这里的目的地是辅助寄存器
LD (HL),r      ; r 是上述 7 个寄存器之一或一个立即数
** LD (HL),r' ;**** 非法指令!
LD r,(IX+d)    ; r 是上述 7 个寄存器之一，d 是从-128 到+127 的偏移量
LD r',(IX+d)   ; 同上，但这里的目的地是辅助寄存器
LD (IX+d),r    ; r 是上述 7 个寄存器之一或一个立即数
LD (IY+d),r    ; IX 或 IY 可以有偏移量 d
```

以下为 16 位索引存取指令。这些指令在 Z180 或 Z80 中都不存在。这些存取操作的一个操作数必须是 HL 或 HL'。

```
LD HL,(SP+d)    ; d 是一个从 0 到 255 的偏移量，16 位被存储入 HL 或 HL′中
LD (SP+d),HL    ; 相应的存储
LD HL,(HL+d)    ; d 是一个从-128 到+127 的偏移量,使用原始的 HL 值寻址，
                ; l=(HL+d)，h=(HL+d+1)
LD HL',(HL+d)
LD (HL+d),HL
LD (IX+d),HL    ; 在 IX 加上-128 到+127 偏移量所指向的地址处存储 HL
LD HL,(IX+d)
LD HL',(IX+d)
LD (IY+d),HL    ; 在 IX 加上-128 到+127 偏移量所指向的地址处存储 HL
LD HL,(IY+d)
LD HL',(IY+d)
```

3．3．4 寄存器之间的传送指令

任何一个 8 位寄存器 A、B、C、D、E、H 和 L 的内容，都可向其它任何 8 位寄存器传送，例如：

```
LD A,C
LD D,B
LD E,L
```

8 位辅助寄存器可作为目标寄存器，例如：

```
LD A′,C
LD D′,B
```

这些指令是 rabbit 所特有的，由于带有前缀，因而需要两个字节,4 个时钟周期。不允许使用指令 LD A,d'或 LD d',e'。
还有一些 16 位寄存器之间的传送指令。除了有提示的以外，这些指令都要求两个字节,4 个时钟周期，列于下：

```
LD dd',BC    ; 这里 dd'是 HL', DE', BC'中的任意一个（2 个字节，4 个时钟）
LD dd',DE
LD IX,HL
LD IY,HL
LD HL,IY
LD HL,IX
LD SP,HL     ; 1 个字节，2 个时钟
LD SP,IX
LD SP,IY
```

其它 16 位寄存器传送可使用 2 个传送指令。

3．3．5 寄存器交换指令

由于通过一条指令就可以完成 2 步（或更多）的传送，因而交换指令的功能十分强大。下面是寄存器交换指令。

```
EX af,af'     ; af 与 af'之间进行交换
EXX           ; HL，DE，BC 与 HL', DE', BC′之间进行交换
EX DE,HL      ; DE 与 HL 之间进行交换
```

以下是 Rabbit 特有的指令：

```
EX DE',HL     ; 1 个字节，2 个时钟
EX DE，HL'    ; 2 个字节，4 个时钟
EX DE', HL'   ; 2 个字节，4 个时钟
```

下面的特殊指令（用于 Rabbit 和 Z180/Z80）实现堆栈顶部 16 位的一个字与 HL 寄存器之间的交换。这三条指令，每条需要 2 个字节和 15 个时钟。

```
EX (SP),HL
EX (SP),IX
EX (SP),IY
```

3．3．6 PUSH 和 POP 指令

寄存器 AF，HL，DE，BC，IX,和 IY 的入栈和出栈可由指令实现。寄存器 AF'，HL'，DE' 和 BC'只能实现出栈操作。只有 Rabbit 支持辅助寄存器出栈，在 Z80/Z180 中是不允许。

例如：

```
POP HL
PUSH BC
PUSH IX
PUSH AF
POP DE
POP DE'
```

```
POP HL'
```

3．3．7 16 位算术、逻辑操作

HL 是基本 16 位累加器。在许多 16 位操作中，IX、IY 也可作为备用累加器。Z180/Z80 的 16 位操作功能不强，实际应用时程序员不得不将 8 位操作结合起来，用以完成 16 位操作。Rabbit 为 16 位操作提供了许多新指令，克服了 Z180/Z80 指令的一些弱点。

Z180/Z80 的基本 16 位算术指令如下：

```
ADD HL,ww    ; 这里 ww 是 HL，DE，BC，SP 之一
ADC HL,ww    ; 带进位加
SBC HL,ww    ; 带借位减
INC ww       ; 寄存器加 1（不影响标志位）
```

在上述指令中，可用 IX、IY 代替 HL。ADD 和 ADC 指令可使 HL 带进位左移。上述指令中可使用备用目标前缀（ALTD），使结果和其标志位存放在相应的辅助寄存器中。当 IX 或 IY 为目标寄存器时，如果使用了 ALTD 标志，则备用标志寄存器中仅存入标志位。

以下是 Rabbit 新添加的指令。

```
;移位指令：
RR  HL      ; 将 HL 带进位循环右移,1 个字节，2 个时钟。注意：用 ADC HL,HL 实现
              带进位循环左移,不需要进位时使用 add HL,HL。
RR  DE      ;1 个字节，2 个时钟
RL  DE      ;DE 带进位循环左移，1 个字节，2 个时钟
RR  IX      ; IX 带进位循环右移，2 个字节，4 个时钟
RR  IY      ; IY 带进位循环右移

;逻辑运算：
AND HL,DE   ; 1 个字节，2 个时钟
AND IX,DE   ; 2 个字节，4 个时钟 2 bytes, 4 clocks
AND IY,DE
OR HL,DE    ; 1 个字节，2 个时钟
OR IX,DE    ; 2 个字节，4 个时钟
OR IY,DE
```

BOOL 指令是用于帮助测试 HL 寄存器的特殊指令。当 HL 不为 0 时，BOOL 将 HL 的值置为 1；否则，HL 的值保持为 0。同时，依照这一结果设置标志位。BOOL 指令也可操作于 IX 和 IY 寄存器上。

```
BOOL  HL        ; 若 HL 不为 0，将其值为 1，设置 HL 的相应标志位
BOOL  IX
BOOL  IY
ALTD BOOL HL    ; 根据 HL 设置 HL'和 f'
ALTD BOOL IY    ; 依据结果的标志位修改 IY 并设置 f'
```

SBC 指令可与 BOOL 指令联合使用来完成比较功能。SBC 指令从一个寄存器中连同进位减去另一个寄存器。如果对被减数进行了非或与操作，会有进位；对比这个进位，执行结果取反（inverted）。下面的

例子说明了 SBC 指令和 BOOL 指令的用法。

```
                ; 测试 HL>=DE – HL 以及 DE 是否 0-65535 的无符号数
        OR a    ; 清除进位
        SBC HL,DE ; 如果 C=0 那末 HL>=DE；如果 C=1 那末 HL<DE

                ; 将进位位转换为 HL 中的 boolean 变量
        SBC HL,HL ; 如果 C=0，将 HL 设置为 0，如果 C=1，将 HL 设置为 0ffffh
        BOOL HL   ; 如果 C 被置位，则 HL=1，否则 HL=0

                ;将非进位位转换为 HL 中的 boolean 变量
        SBC HL,HL ; 如果 C=0 则 HL=0，C=1 则 HL=ffff
        INC HL    ; 如果 C=0 则 HL=1，C=1 则 HL=0;注意：设置了进位标志位,但零/符号标志位翻转
(reversed)
```

为了使用 SBC 指令比较有符号数，在执行比较之前，程序员可以将每一个有符号数映射为相应的无符号数，做法是反转符号位。这种映射将最小的负数-08000h 映射为最小的无符号数 0000h，将最大的正数+07FFFh 映射为最大的无符号数 0FFFFh。一旦数据转换完毕，就可以按照无符号数来进行比较。这种方法比起需要测试符号位和溢出位的跳转来说要快得多。

例子：

```
                ; 测试 HL>=DE，这里 HL 和 DE 都是有符号数;
                ;符号位都反转
        ADD HL,HL ; 左移位
        CCF       ; 进位取反
        RR HL     ; 右循环移位
        RL DE
        CCF
        RR DE     ; DE 符号取反
        SBC HL,DE ; 如果 HL>=DE，则不进位，
                ;如果 HL>=DE，生成值为 true 的 boolean 变量
        SBC HL,HL ; 如果没有进位，为 0，否则为–1
        INC HL    ; 如果没有进位，为 1，否则为 0
        BOOL      ; 需要时使用该指令设置标志
```

SBC 指令也可以用于符号扩展。

```
                ;扩展符号 L 到 HL
        LD A,L
        RLA       ; 要携带的符号
        SBC A,A   ; 如果符号为负，则 a 所有位都是 1
        LD H,A    ; 符号被扩展
```

乘法指令执行有符号数乘法时，产生的结果是一个 32 位有符号数。

```
      MUL     ; 有符号数 BC 和 DE 相乘，结果存放在 HL:BC 中—— 1 个字节，12 个时钟
```

如果两个 16 位数相乘得到一个 16 位的结果，那末只能使用 32 位结果的低位部分（BC）。无论两个乘数是有符号整型数还是无符号整型数，这个结果都是正确的。下面介绍的方法可以用来实现两个 16 位无符号整型数乘法，并得出一个 32 位无符号结果。这用到了一个结论——如果乘数之一是负数时，负号将导致从乘积中减去另一个乘数。下面所示的方法将减去的数字又重新加了两次，因而消除了前述结论的影响，此时该符号位被看作一个导致加运算的正号。

```
LD BC,n1
LD HL',BC ; 将 BC 保存在 HL'中
LD DE,n2
LD A,b    ; 保存 BC 的符号
MUL       ; 在 HL:BC 中产生结果
OR a      ; 检测乘数 BC 的符号
JR p,x1   ; 如果是正的，则继续
ADD HL,DE ; 调整 BC 中的负号
x1:
RL DE     ; 检测 DE 的符号
JR nc,x2  ; 如果不是负的，则从 HL 中减去的另一个乘数
EX DE,HL'
ADD HL,DE
x2:           ; 最后，在 HL:BC 中产生 32 位结果
```

该方法可以经过修改进而实现有符号数和无符号数相乘。在这种情况下，只需判断无符号数的符号是否开启，如果是的话，将有符号数加到乘积的高位部分上去。

乘法指令还可用于实现左、右移位。一个数向左移动 n 位可以通过将该数乘以无符号数 2^n 来实现。这通过 n # 15 实现，而且这个数可以是有符号数也可以是无符号数。要想完成一个数向右移动 n 位（0<n<16），这个数应该乘以无符号数 $2^{(16-n)}$，乘积的高半部分可以得到。如果数是有符号数，则有符号数乘以无符号数一定要执行；如果数是无符号数，或作为无符号数做逻辑右移，则无符号数乘以无符号数一定要执行。排除乘数是 2^{15} 的情形，移位问题会得到简化。

3.3.8 输入/输出指令

Rabbit 使用完全不同的方案访问输入/输出接口。任何存储器访问指令都使用两种前缀，一种表示内部 I/O 空间，另一种表示外部 I/O 空间。加上前缀的存储器指令可以访问与 16 位存储器地址重叠的 I/O 地址表示的空间。例如

```
IOI LD A,(85h)  ;地址 85h 的内部 I/O 寄存器的内容装入 A 寄存器
LD IY,4000h
IOE LD HL,(IY+5) ; 从外部 I/O 地址 4005h 获得一个字
```

通过使用前缀，所有 16 位存储器访问指令都可以读写 I/O 区域。当执行 I/O 操作时，内存映射被忽略。对内部 I/O 寄存器的写操作仅需要两个时钟周期，而写到存储器或外部 I/O 设备至少要用 3 个时钟周期。

在某种情形下，当 I/O 操作后跟随一条特殊的单字节指令时，Rabbit 2000 的一个 BUG 导致 I/O 访问

取代了内存访问操作。如果一个 I/O 指令（前缀是 IOI 或 IOE）后面跟随 12 条单字节操作码（用 HL 作为索引寄存器）之一，这个 BUG 出现。这 12 条指令是：

```
ADC A,(HL)        SUB (HL)
ADD A, (HL)       XOR (HL)
AND (HL)          DEC (HL)
CP (HL)           INC (HL)
OR (HL)           LD r,(HL)
SBC A,(HL)        LD (HL),r
```

此处 r 是一个 8 字节寄存器，代表 A,B,C,D,E,H 或 L 之一。

在用户书写的汇编语言程序中很容易出现的唯一联合是 LD (HL),r 后面跟随 I/O 指令。

失败（failure）的特征是内存地址转换没有执行，所以第二条指令所要求的正确的内存片选没有被使能。外部 I/O 操作时，端口 E 的 I/O 选通可被使能，I/O 片选（I/O 选通）会取代内存片选。如果上述指令之一置于一个内部 I/O 操作之后，而且存储器访问发生在地址转换没有执行的基区，内存操作会得到正确执行，因为内部 I/O 操作所要求的正确的内存片选得到使能。

在 I/O 指令和上述所列指令之间加上一条 NOP 指令可以很容易的避免这个 BUG。

Rabbit 用户不太可能遇到这个问题，因为用动态 C 编译器或在任何标准库中，都不会出现导致 BUG 发生的指令序列。

从 6.57 发行版之后，动态 C 编译器和汇编器将在产生的代码中必要时加入 NOP 指令以修正这个异常。

3.4 汇编语言如何处理-技巧和窍门

3.4.1 使 HL 在四个时钟周期变 0

```
BOOL HL ; 2 个时钟周期,清除进位,HL 是 1 或 0
RR HL ; 2 个时钟周期，共 4 个时钟周期--除去可能的 1
```

这个序列要求 4 个时钟周期，而 LD HL,0 需要 6 个时钟周期。

3.4.2 不直接实现交换

HL<->HL’ 需要 8 个时钟周期

```
EX DE’,HL ; 2 个时钟周期
EX DE’,HL’ ; 4 个时钟周期
EX DE’,HL ; 2 个时钟周期，共 8 个时钟周期
```

DE<->DE’ 需要 6 个时钟周期

```
EX DE’,HL ; 2 个时钟周期
EX DE,HL ; 2 个时钟周期
EX DE’,HL ; 2 个时钟周期,共 6 个时钟周期
```

BC<->BC’ 需要 12 个时钟周期

```
EX DE’,HL ; 2 个时钟周期
EX DE,HL’ ; 4 个时钟周期
EX DE,HL ; 2 个时钟周期
EXX ; 2 个时钟周期
EX DE,HL ; 2 个时钟周期
```

在 IX，IY 和 DE，DE’之间移动内容
IX/IY->DE / DE->IX/IY

```
;IX，IX --> DE
EX DE,HL
LD HL,IX/IY / LD IX/IY,HL
EX DE,HL ;共 8 个时钟周期，DE --> IX/ IY
EX DE,HL
LD IX/IY,HL
EX DE,HL ; 共 8 个时钟周期
```

3.4.3 布尔变量的处理

当 HL 是一个逻辑变量，取值 1 或 0，逻辑操作使用 HL，这对 C 语言很重要，C 语言中用最少 16 位的整数表示一个逻辑结果。

逻辑非操作——HL 的 0 位取反需要 4 个时钟周期（对 IX，IY 需要 8 个时钟周期）。

```
DEC HL ;1 变为 0，0 变为-1
BOOL HL ; -1 到 1，0 到 0. 共 4 个时钟周期
```

逻辑异或操作——xor HL,DE 当 HL/DE 是 1 或 0 时.

```
ADD HL,DE
RES 1,l ; 共 6 个时钟周期，清除结果的位 1，如果 1+1=2
```

3.4.4 整数的比较

无符号整数的比较可以通过减操作后测试零或进位标志实现。如果两数相等，零标志置位。运用 SBC 指令，当被减数小于减数，被清除的进位位置 1。8 位无符号整数范围是 0 – 255。16 位无符号整数范围是 0 – 65535.

```
OR a ; 清除进位
SBC HL,DE ; HL=A 且 DE=B
A>=B !C
A<B C
A==B Z
A>B !C & !Z
A<=B C v Z
```

如果 A 在 HL 内且 B 在 DE 内，则指令执行如下。此时假定，HL 被置位或清零取决于比较结果是真或假。

```
; 计算 HL<DE
;无符号整数
; EX DE,HL ;没有注明 DE<HL
OR a ; 清进位位
SBC HL,DE ; C 置位，如果 HL<DE
SBC HL,HL ; HL-HL-C 为 -1， 如果进位位置 1
BOOL HL ; 如果有进位， HL 为 1,否则为 0
        ;否则 结果等于 0
;无符号整数
; 计算 HL>=DE 或 DE>=HL – 检查 !C
; EX DE,HL ; 不注明 DE<=HL
OR a ; 清进位位
SBC HL,DE ; !C 为 1， 如果 HL>=DE
SBC HL,HL ; HL-HL-C 等于 0，若无进位；等于-1 ，有进位
INC HL    ; 共 14 / 16 个周期 ，如果在第一次执行 SBC 结果为 1 后 C 置 1,
          ; 否则为 0
; 0，若 C 置位 ； 1， 若!C 为 1
;
； 计算 HL==DE
OR a ;清进位位
SBC HL,DE ;为 0，若相等
BOOL HL ; 强制为 0，1
DEC HL ; 逻辑取反
BOOL HL ; 共 12 个周期 –逻辑非,若输入相同，为 1
;
```

如果被比较的无符号数中有一个是常数，则可以简化。注意进位与 SBC 指令有相反的含义。

```
  ;测试条件 HL>B，B 是常数
  LD DE,(65535-B)
  ADD HL,DE ;若 HL>B 进位位置 1
  SBC HL,HL ; HL-HL-C  ——若进位位置 1 结果为-1,否则为 0
  BOOL HL ; 共 14 个周期 – 如果 HL>B ，为真

  ; HL>=B， B 是常数，非零
  LD DE,(65536-B)
  ADD HL,DE
  SBC HL,HL
  BOOL HL ; 14 个周期
  ; HL>=B 且 B 是 0
  LD HL,1 ; 6 个周期
  ;
  ; HL<B B 是一个常数,非 0 (如果 B==0 总为 FALSE)
  LD DE,(65536-B)
  ADD HL,DE ; 如果 HL<B 进位清 0
```

```
SBC HL,HL ;  如果进位置 1 结果为-1，否则为 0
INC HL ; 14 个时钟周期 –结果为 0 ，如果有进位，否则为 1
;
; HL <= B ，B 是常数，非 0
LD DE,(65535-B)
ADD HL,DE ; ~C 为真， 如果 HL<=B
CCF ; C 置 1，如果为真
SBC HL,HL ;如果 C 置 1，结果为 –1，否则为 0
INC HL ; 共 16 个时钟周期， -- 如果为真结果为 1,否则为 0
;
; HL <= B B 为零 -  如果 HL==0 则结果为真
BOOL HL ; 结果存放在 HL 中
;
; HL==B 且 B 是一个非零常数
LD DE,(65536-B)
ADD HL,DE ; 如果相等，结果为零
BOOL HL
INC HL
RES 1,l ; 16 个时钟周期
; HL==B 且 B==0
BOOL HL
INC HL
RES 1,l ; 8 个时钟周期
```

对于有符号整数常规方法是察看零标志，符号标志和溢出标志。有符号 8 位整数范围 – 128 到 +127 (80h 到 7Fh).有符号 16 位整数范围是 – 32768 到 + 32767 (8000h 到 7FFFh).如果没有溢出，通过符号和零标志可以判断减操作后哪个数大；溢出时，符号标志需要逻辑取反，也就是说，操作是错的。

```
A>B (!S & !V & !Z) v (S & V)
A<B (S & !V) v (!S & V & !Z)
A==B
A>=B
A<=B
```

有符号数比较的另一种方法是通过反转 D15 位把符号整数映射为无符号整数。参见 34 页图 11。一旦两个数的 D15 位反转，完成映射，比较可以像无符号整数那样进行。这样可以避免建立跳转分支以测试溢出和符号标志。例如。

```
; 对符号整数测试 HL>5
LD DE,65535-(5+08000h) ; 5 被映射为无符号整数
LD BC,08000h
ADD HL,BC ; 高位取反
ADD HL,DE ; 至此 16 个时钟周期
; 如果 HL>5，进位位置 1 --进位时有跳转的可能
SUBC HL,HL ; HL-HL-C ; 如果 C 为 1 结果为 -1,否则为 0
BOOL HL ; 共 22 个时钟周期 - 如果 HL>5 ，为真，否则为假
```

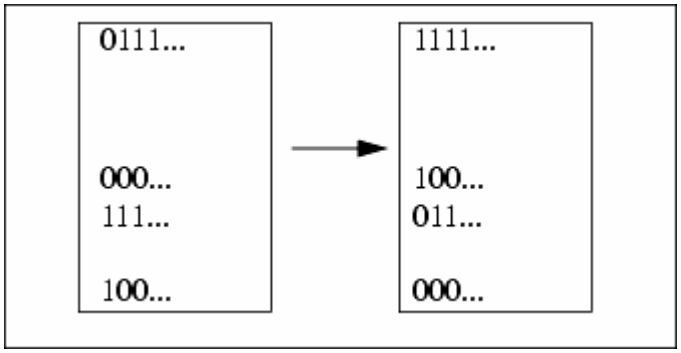


图 11.通过反转位 15 映射有符号数为符号数。

3.4.5 从存储器到 I/O 空间的微移动（Atomic Move）

为了避免在拷贝影子寄存器到它的目标寄存器时中断被禁止，最好有从存储器到 I/O 空间的微移。这可以通过 LDD 或 LDI 指令实现。

```
LD HL,sh_PDDDR ;指向影子寄存器
LD DE,PDDDR ; 设置 DE 指向 I/O 寄存器
SET 5,(HL) ;影子寄存器的位 5 置 1
           ;使用 ldd 指令做微传输
IOI ldd ; (io DE)<-(HL) HL--, DE—
```

当 LDD 指令使用 I/O 前缀，目标便成为 DE 指定的 I/O 地址。HL 和 DE 的递减是其副作用。如果使用了重复指令 LDIR 和 LDDR，在连续的重复之间可能发生中断。利用单条非可中断指令，通过到 I/O 空间的字存储，可以在相邻地址设置两个 I/O 寄存器。

3．5 中断结构

当 Rabbit 的一个中断发生时，返回的地址入栈，控制被传递到中断服务程序地址处。中断服务程序地址分为两部分：高字节来自专门寄存器，而低字节则由硬件固定分配给每一个中断。内部中断寄存器(IIR)和外部中断寄存器(EIR)来指明中断服务程序地址的高位字节。由专门指令访问这些寄存器。

```
LD A,IIR
LD IIR,A
LD A,EIR
LD EIR,A
```

由硬件设备或特定的 1 字节复位指令来初始化中断。

```
RST 10
RST 18
RST 20
RST 28
RST 38
```

RST 指令集与 Z80 和 Z180 中的相像，但一些特定的指令已经从指令集(00，08，30)中除去不用。RST

中断不受处理器优先级的约束。由于这些复位指令主要保留为动态 C 调试使用，建议用户谨慎使用复位指令。与 Z80 或 Z180 不同，Rabbit 中由内部中断寄存器 IIR 提供 RST 中断服务程序地址的高字节。

由于中断程序不影响 XPC，中断程序必须定位于根代码空间里。但是，XPC 存入栈后中断程序可以跳转到扩展代码空间。

3.5.1 中断优先级

Z80 和 Z180 有两个中断优先级：可屏蔽中断和非屏蔽中断。非屏蔽中断不能被禁止，有固定中断服务程序地址 66h。而在 Rabbit 中，有三个中断优先级，而处理器可工作于四个优先级。当一个中断被请求时，如果中断优先权高于处理器的级别，则在当前指令(除了特权指令)完成以后，中断发生。

多中断优先权为嵌入式系统程序提供了极快的中断响应。中断等待时间（interrupt latancy）指中断请求到中断得到相应的时间。一般情形下，在中断服务程序执行时，同一优先级中断被禁止。有时中断要保持禁止状态直到中断服务程序结束。其他时候中断则可以被重新使能，一旦中断服务程序至少禁止了自己中断的原因。在任何情形下，如果几个中断程序在同一优先级上运行，在某一中断程序等待前一中断程序允许更多中断发生时，也就引入了中断等待时间的问题。如果很多设备具有中断服务程序，且所有中断处于同一优先级，则挂起的中断要到至少运行的中断结束时才能发生，或者它改变中断优先级。作为一种经验原则，Z-World 通常建议基于 Z180 的控制器具有 100μs 的中断等待时间。这样，如果有 5 个激活的中断程序，则每一个中断程序关断至多 20 μs.

在 Rabbit 中，希望大多的中断设备使用优先级 1。需要极快速中断响应的设备使用优先级 2 和 3。由于运行在优先级 0 或 1 的代码永不会禁止优先级 2 和 3 的中断,这些中断会在 20 个时钟周期之内发生，这个时间长度是最长的指令或切合实际的最长特权指令(后面尾随一条非特权指令)要求的时间。用户要小心不要过度禁止关键代码区域的中断。*处理器优先级不该高于 1，除非在深思熟虑的场合。*

处理器优先级对中断的影响示于表 2。中断优先级常常由与产生中断的硬件相关联的 I/O 控制寄存器的位来设置。8 位中断寄存器(IR)在最低两位中保存处理器优先权。当中断发生时，IR 寄存器向左移位，最低两位被设置等于该发生中断的优先级。这意味着一个中断服务程序只能被更高级别的中断服务程序中断(除非其优先级由程序员明确设低)．IR 寄存器作为 4 字堆栈来保存和恢复中断优先级。此寄存器可以右移，通过专门的指令(IPRES)恢复以前的优先级。由于 IP 只能储存当前处理器优先级和 3 个刚使用过的优先级，我们提供了 PUSH 和 POP IP 指令且使用常规栈。一个新的优先级可以使用专门指令(IPSET 0，IPSET 1,IPSET 2，IPSET 3)推入 IP 寄存器。

表 2.处理器优先级对中断的影响

处理器优先级	对中断的影响
0	所有中断(优先级 1，2 和 3)，在当前非特权指令执行后发生。
1	只有优先级 2 和 3 中断发生。
2	只有优先级 3 中断发生
3	所有中断被抑制(除了 RST 指令)

3.5.2 多外部中断设备

Rabbit 有两个外部中断请求信号。如果有 2 个以上的外部中断源，则必须多个设备共用这两个中断源。如果中断源是边沿触发，当上升沿或下降沿发生时，向 Rabbit 发出中断申请，这可以通过设置寄存器来指定。中断引脚和并口 E 共用，所以通过读并口 E 也可以读取中断的状态。

如果几个中断源共用一个中断信号，所有中断请求都作“或”运算，这样任何设备均可以产生中断申请。如果几个设备同时请求中断，只能有一个中断产生，因为只有一个中断申请信号传送到中断引脚。为解决这种情形并保证不同设备的中断程序得以调用，一种好的做法是在软件上设立中断分配器，对每个设备提供分别的请求状态信号。中断申请信号在“或”运算之前，就是各自设备中断请求信号。中断分配器按照优先级顺序调用各自设备的中断程序，保证所有设备中断得到响应。

3.5.3 特权指令，关键部分和标志

通常中断发生在当前指令执行结尾处。但是，如果执行的指令是有特权的，中断就不能发生，要推迟到一条非特权指令执行之后。特权指令执行了一些特殊的操作，这些特殊操作后紧接着响应中断会产生问题。明确的禁止中断也许会延迟中断响应时间，因为特权指令的目的是操纵中断控制。特权指令的其他信息见 18.19“特权指令”.

载入堆栈的特权指令如下。

```
LD SP,HL
LD SP,IY
LD SP,IX
```

下面加载 SP 的指令是有特权的，因为这些指令后常跟着一条改变堆栈段寄存器的指令。如果一个中断发生在这两条指令之间，堆栈会变得混乱。

```
LD SP,HL
IOI LD sseg,a
```

操作 IP 寄存器的特权指令如下。

```
IP 0 ; 左移 IP 寄存器，在位 1,0 设置优先级 00
IP 1
IP 2
IP 3
IPRES ; 循环右移 IP2 位,恢复前一使用的优先级
RETI ;  IP 出栈，然后弹出返回地址
POP IP ;  IP 出栈
```

3.5.4 关键部分

某些库程序可能需要在关键部分禁止中断。当然，如果处理器中断优先级是 0 或 1，这些程序为合法调用。一个更高的优先级中断说明是用户手工编写的汇编程序，而不是使用通用的编译库。下面代码可以禁止优先级 1 中断。

```
IP 1 ; 保存前一使用的优先级，设置优先级为 1
....关键部分...
IPRES ; 恢复前一优先级
```

这段代码是安全的，如果已知道关键部分的代码没有再嵌入关键部分。如果这段代码有嵌套，就有溢出 IP 寄存器的危险。可以嵌套的部分形式则如下。

```
PUSH IP
IP 1 ; 保存前一优先级，设置优先级为 1
....关键部分...
POP IP ; 恢复前一优先级
```

下面指令也是有特权的.

```
LD A,xpc
LD xpc,a
BIT B,(HL)
```

3.5.5 使用指令 Bit B,(HL)的门控（Semaphores）

bit B,(HL)指令具有通过以下代码建立门控结构。

```
BIT B,(HL) ; 测试(HL)上字节的某位
SET B,(HL) ; 确保该位置 1，不影响标志
; 如果标志位为零，则设置该标志位使资源归我们所有;
; 否则其他程序拥有该资源
```

一个标志位可以用来获得对某资源的控制权，此资源在同一时刻只能属于一个任务或程序。这可以通过测试某一位是否为 1，如果为 1，表明有其他程序在使用此资源；如果为 0，则此程序立即设置该位，以宣称现在此资源归本程序所有。在测试该位和设置该位之间不允许发生中断，如果允许中断可能导致两个不同程序都认为自己拥有此资源。

3.5.6 计算长调用和跳转（computed long call and jumps）

设置 XPC 的特权指令可以进行计算长调用和跳转。这可以按下列指令序列实现。

```
LD xpc,a
JP (HL)
```

这时，A 拥有一个新的 XPC，HL 拥有一个新的 PC。这段代码一般应该在根段执行，这样防止 JP (HL) 指令执行时超出寻址范围。

下列代码完成对某一计算地址（computed address）的调用。

```
; A=xpc, IY=address
;
LD A,newxpc
LD IY,newaddress
LCALL DOCALL ; 调用根段里的实用程序
;
;  DOCALL 程序段
DOCALL:
LD xpc,a ; 设置 xpc
JP (IY) ; 跳转到程序
```

4. Rabbit 的性能

这部分描述 Rabbit 多方面的性能，这些性能在技术描述中可能不够详细。

4.1 精确的定时输出脉冲

Rabbit 在软件控制之下可以实现精确的脉冲输出。中断等待时间的影响得以避免，因为中断总是准备一个脉冲沿，此脉冲沿在下一个时钟周期才进入输出寄存器里。见图 12。

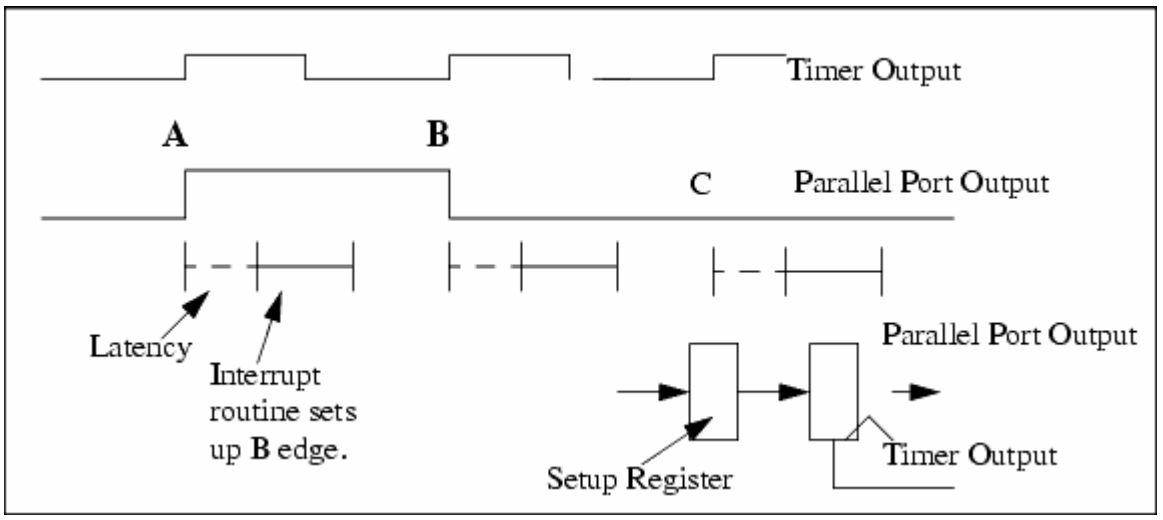


图 12 定时输出脉冲

图 12 的定时器输出是周期性的。只要中断程序能在一个定时器周期内完成，任意模式的同步脉冲都可以从并口输出。

中断等待时间依赖于中断优先级和抑制时间（其他的相同或更高优先级的中断程序占用的时间）。最高优先级中断程序的第一条指令在中断请求后 30 个时钟周期内开始执行。这包括执行最长指令需要的 19 个时钟周期和执行中断的 10 个时钟周期。每个 16 位寄存器入栈需要 10 – 12 个时钟周期。寄存器出栈需要 7 – 9 个时钟周期。中断返回需要 7 个时钟周期。如果有 3 个存储器入栈和恢复，并且要执行 20 条指令，每条指令平均需要 5 个时钟周期。整个中断程序需要约 200 个时钟周期，在 20 MHz 晶振下的 10 μs。按这个定时方式，以下特性成为可能。

脉宽调制输出——最小脉宽是 10 μs。如果重周期为 10ms，则具有 1000 个不同宽度的脉冲会以 100 次每秒的速率产生。

异步通讯串行输出——每 10 μs 由一个新脉冲产生异步输出数据。这符合波特率 100,000 bps.

异步通讯串行输入——为捕捉异步串行输入，输入轮询必须快于波特率至少 3 倍~5 倍更好。如果采用 5 倍轮询，可以接收 20,000 bps 的异步输入。

以精确的定时关系产生脉冲——两个事件的间隔可以控制在 10 μs 到 20 μs 之间.

用一个定时器产生的周期性时钟，可控制事件的精度接近 10 μs。但是，如果采用定时器 B 控制输出寄存器，精度约高 100 倍。这是因为定时器 B 有一个匹配寄存器，后者可编程在未来指定时刻生成脉冲。匹配寄存器有两个级联寄存器，即匹配寄存器和下一个匹配寄存器。当一个脉冲生成时，下一个匹配寄存器的内容装入此匹配寄存器。这使事件之间很紧密，定时器 B 每次计数产生一次事件。定时器 B 的时钟是 sysclk/2，分频数是 1 – 256 中的一个。定时器 B 在 20M Hz 系统时钟下，计数速率最高

达 10 MHz，这允许事件间隔为 100 ns。定时器 B 和匹配寄存器为 10 位。

使用定时器 B 输出脉冲的定位精度达 sysclk/2。结合使用外部中断信号，定时器 B 可以捕获外部事件的发生时间。可以编程选择中断信号在上升沿、下降沿还是同时在两个沿上发生中断。为捕获脉冲边沿时间，中断程序可以读定时器 B 计数器。可以从定时器值中减掉中断点到读定时器的时间点的执行时间。如果没有其他相同或更高优先级的中断，则边沿位置的不确定性减小为中断等待时间的不确定，或为最长指令的执行时间的一半。这个不确定性约为 10 个时钟周期，在 20 MHz 时钟下为 0.5 μs。这使得任何脉冲的脉宽测量精度达到 1 μs。如果多个脉冲需要同时测量，精度会有所下降，通过仔细编程可以把这个误差下降减到最小。

4.1.1 脉宽调制减少继电器功耗

典型继电器保持闭合所需电流要比最初闭合时小的多。例如，继电器吸合后，利用脉宽调制使继电器用 75%的占空比维持电压工作，维持电流（holding current）将约为恒定电压时电流的 75%，而功耗则降为 56%。

脉宽调制频率从 5 kHz 到 20 kHz。如果周期中断每 50 μs 中断一次，则 100 μs 的工作周期建立的占空度是 50%。200μs 工作周期时，25%，50% 或 75% 的占空比都可实现。而 250 μs 周期允许的占空比是 20%，40%，60% 或 80%。这种中断程序的代码如下。

```
push af ; 10
push hl
push de
ld hl,(ptr)      ; 11 获得数组里的位置指针
ld a,(maskand)   ; 9 获得屏蔽
and a,(hl)       ; 5 获得当前输出
ld e,a           ; 2
ld a,(maskor)    ; 9
or a,e           ; 2
ioi ld (port),a  ; 13 存储在端口里
inc hl           ; 2 指向下一个
ld a,(hl)        ; 5 检验是否是数阻尾
or a,a           ; 2
jr nz,step2      ; 2
ld hl,(beginptr) ; 11 重置 hl 为数组起使
step2:
ld (ptr),hl      ; 13 存储 hl
pop de           ;7
pop hl
pop af
reti             ; 7 中断返回
; 最差情况下为 153 个时钟周期-7.5 us （20 Mhz）
```

这个程序要占用处理器计算时间的 15%，假定中断之间间隔 50 μs。程序可以执行更快，其代价是程序变得复杂。不用“与”和“或”屏蔽，通过高级程序可以直接修改数组，数组的结尾可以通过测试

寄存器 HL 中的位方法得以判断。高级程序在修改数组的位时要抑制中断，或者要么防止修改数组时的不稳定输出。如果继电器在调制期间发出声音，声能会在周期性缺失"off"脉冲频谱上传出，产生 180°的相移。下述程序执行时间是上面程序的 2/3。

```
push af      ;10
push hl
ld hl,(ptr) ;11
ld a,(hl)   ;5
ioi ld (port),a ; 13 数据输出
inc hl
ld a,0fh ;4
and l ; 判断 hl 是否到了循环结尾
jr z,step2
ld (ptr),hl
pop hl
pop af
reti
step2:
ld a,(beginptr)
ld l,a
ld (ptr),hl ;13
pop hl ;7
pop af
reti
; 共 103 个时钟周期
```

4.2 用于键盘扫描的漏极开路输出

并口 D 输出可以被独自编程为漏极开路。这对扫描开关矩阵很有用，如图 13。把一行驱动为低电平，然后扫描各列检测是否有低电平线，有的话表示有键按下。每行都重复这么做。漏极开路输出的优点是如果同一列有两键同时按下，则一个驱动器驱动这条线输出高电平与另一个驱动器驱动这条线输出低电平不会冲突。

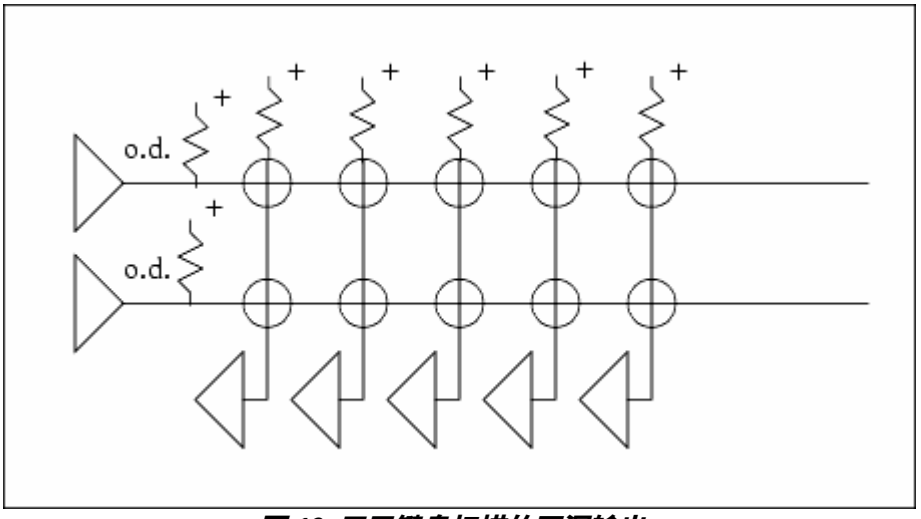


图 13.用于键盘扫描的开漏输出

4.3 冷启动(cold boot)

多数微处理器上电或复位后从一固定地址，常为 0000H 地址开始执行程序。Rabbit 有两个启动模式选择引脚 SMODE0，SMODE1（见图 14），这两引脚的逻辑状态决定复位后的启动过程。如果两脚接地，Rabbit 从地址 0 开始执行程序。复位后，内存起始地址从 0 开始，存储器连接在控制引脚/CS0 和/OE0 上。然而还有 3 种其他的启动方式。这些备用方法通过通信口接受数据流，该通信口用于向 RAM 中存储启动程序，而此启动程序又可进一步启动二级启动程序，在同一个通信口下载某个程序的细节描述见 7.9 节。

有 3 个通信通道可以用于自引导，分别为异步方式下的串口 A（2400bps），时钟同步方式下的串口 A，和（并行）从端口。

冷启动协议允许三个字节为一组的数据形式，两字节定义地址，一字节为数据。每个数据组将一个数据字节写入指定的存储器或内部 I/O 空间。地址最高位置位则指明是 I/O 空间，这样无论写到内存还是 I/O 空间，都只能达到前 32K 字节。对 I/O 空间某地址的存储操作会中止冷启动，因为它会致使程序又从地址 0 开始执行。由于内存芯片可以通过对某 I/O 地址的寄存器操作而被重新映射到 0 地址，所以，RAM 可以临时映射到 0 地址，以避免处理更加复杂的闪存的写协议。而通常 0 地址的默认存储器是闪存。

下面是冷启动特性的优势。

- 闪存可以焊接到微处理器主板上，并通过一个串口或并口编程。这样避免还要把这个部件插入主板或在焊接前用 BIOS 或引导程序对它编程。
- 对闪存的再编程可在现场完成。当开发平台可以在不考虑处理器中存在的软件的状态下执行完整的软件重装载操作时，这个特点在软件开发过程中非常有用。Dynamic C 标准编程电缆允许开发平台复位并冷启动目标系统,即基于 Rabbit 的微处理器主板。
- 如果 Rabbit 作为一个从处理器使用，主处理器可以通过从端口将它冷启动。这意味着从处理器可以在没有非易失性存储器的情况下工作，只需 RAM 即可。

4．4 从端口（slave port）

从端口允许一个 Rabbit 处理器作为另一个处理器的从处理器，而主处理器也可以是一个 Rabbit 处理器。从处理器必须要有一个处理器芯片和一个 RAM 芯片，时钟和复位信号可以由主处理器提供。主处理器可以冷启动从处理器，然后下载一个程序给它。主处理器不一定非要是 Rabbit 处理器，它可以是能够读写标准寄存器的任何类型的处理器。

要获得详细的信息，参看“Rabbit 从端口”。

从处理器的从端口连接到主处理器的数据总线。主从处理器之间的通信通过三个寄存器进行。在 Rabbit 中设计时，考虑到为双向通信服务，共有六个数据寄存器。此外，还有一个从端口状态寄存器，它可被主处理器或从处理器读取（参看图 38）。由主处理器使用两根从寄存器地址线选择被读或写的寄存器。当某个寄存器把数据从主处理器携带到从处理器时，对主处理器对它表现为写寄存器，而对从处理器表现为读寄存器。进行相反方向操作时，对主处理器是读寄存器，对从处理器是写寄存器。这些寄存器在双方来看都是读-写同一寄存器，但并不是真正意义上的读-写一个寄存器，因为写入和读出数据的地方不同。主处理器提供时钟信号或选通脉冲，向它所控制的三个写寄存器里存放数据。主处理器也可以写状态寄存器。主处理器能写的寄存器对从 Rabbit 处理器表现为读寄存器。主处理器提供一个使能选通脉冲来读取那三个读数据寄存器和状态寄存器。对从 Rabbit 处理器来说，这些寄存器是

写寄存器。这三对寄存器中第一个写寄存器比较特殊，可以通过主-从通信联络信号中断另一个处理器。当从处理器写零号寄存器时，一个输出信号有效，这个信号可中断主处理器。而当主处理器写零号从寄存器时，可通过对从处理器内部电路的设置来中断从处理器。

双方都能访问状态寄存器，该寄存器记录所有状态，并报告是否有一方有中断请求。状态寄存器跟踪每个寄存器的“满-空”状态。当一端对寄存器执行写操作，该寄存器为满。另一端读取它时，它变成空。通过这种方法，一端 CPU 可以检测另一端是否已经修改了寄存器，或两端是否向寄存器都存储了信息。主-从通信链路机制使“设置和忘却（set and forget）”通信协议成为可能。任一方都可以通过向某寄存器写数据来发出命令或请求,然后只管做自己的事情,而另一方根据其自身的时间安排处理这个请求。向零号寄存器存储时将产生中断唤醒另一方，也可以通过周期的查询状态寄存器来提醒自身。

这三个寄存器中，零号从寄存器有一个特别功能，写这个寄存器时在另一端产生一个中断（如果这个中断被开放）。有一种协议就是首先向 1 号和 2 号寄存器存储数据，最后才向零号寄存器写数据。这样，在链路的另一端，中断程序可以获得 24 位的数据。

可以使每个字节传输产生一次中断来进行成批数据传输，这与通常的串行口或 UART 相似。与通过 UART 所做的类似，这种情况下，也能够进行全双工传输。这种传输的开销是每传输一个字节需 100 个时钟周期，假定这是一个 20 条指令的中断程序（为把中断程序保持在 20 条指令以内，中断程序需要安排的非常紧凑）。有几种方法可以加快传输同时占用较少的计算机开销。每次中断时,可以用足够的寄存器来传输两个字节，这样几乎把开销减少了一半。如果在两个处理器之间集中进行数据传输，那么传输一个字节大约需要 25 个时钟。任一方查询状态寄存器，并等待对方来读/写一个数据寄存器，然后这个数据寄存器就被对方再次写/读。

4.4.1 从 Rabbit 作为一个协议 UART

Rabbit 作为从处理器主要应用就是创建一个 4 端口的能处理通信协议细节的 UART(UART—通用异步收发器)。主处理器在从端口上发送和接受消息。错误更正,重发等工作可由从处理器处理。

5.引脚分配及功能

5.1 封装示意图和引脚引出线(pinout)

见图 14.

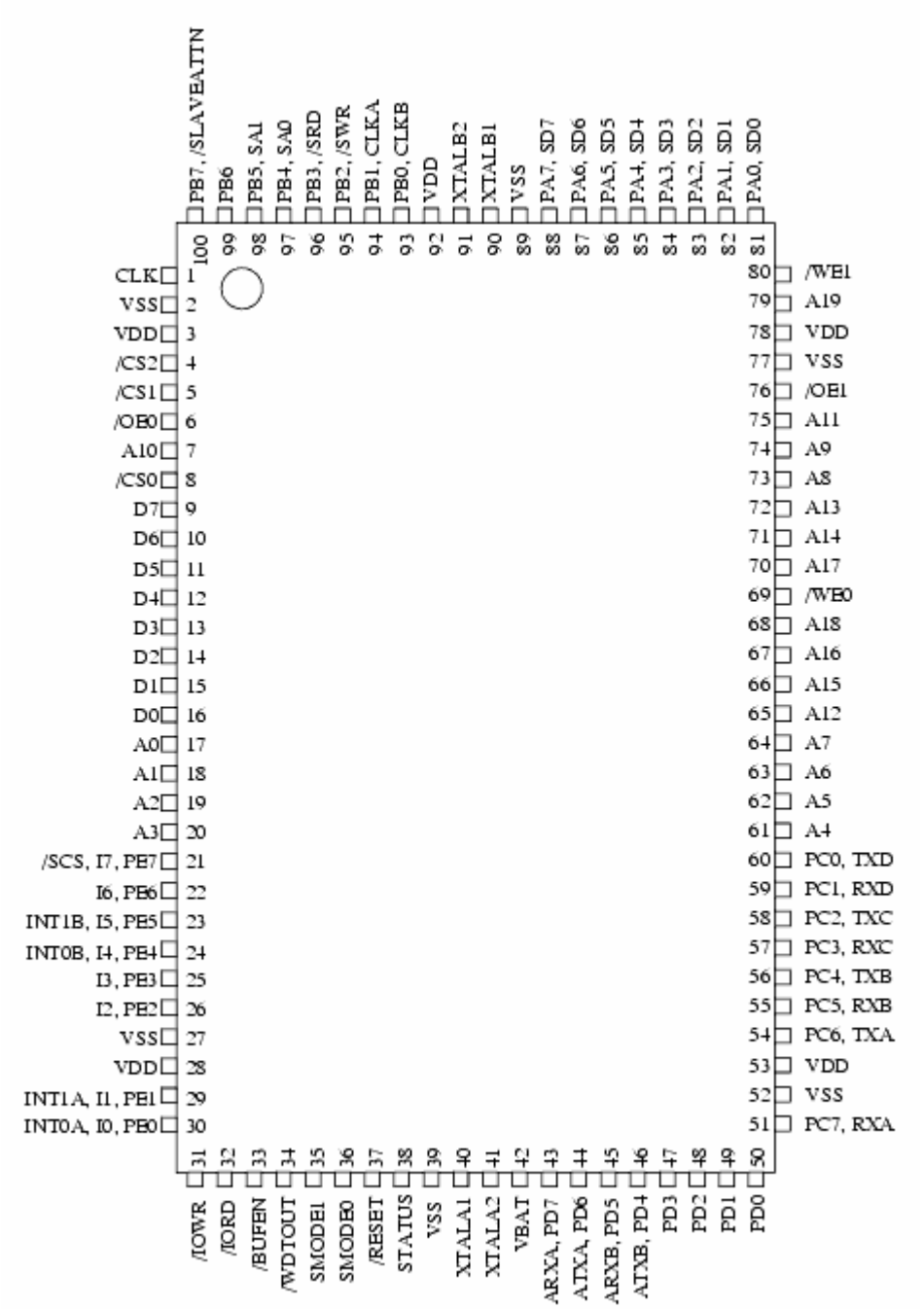
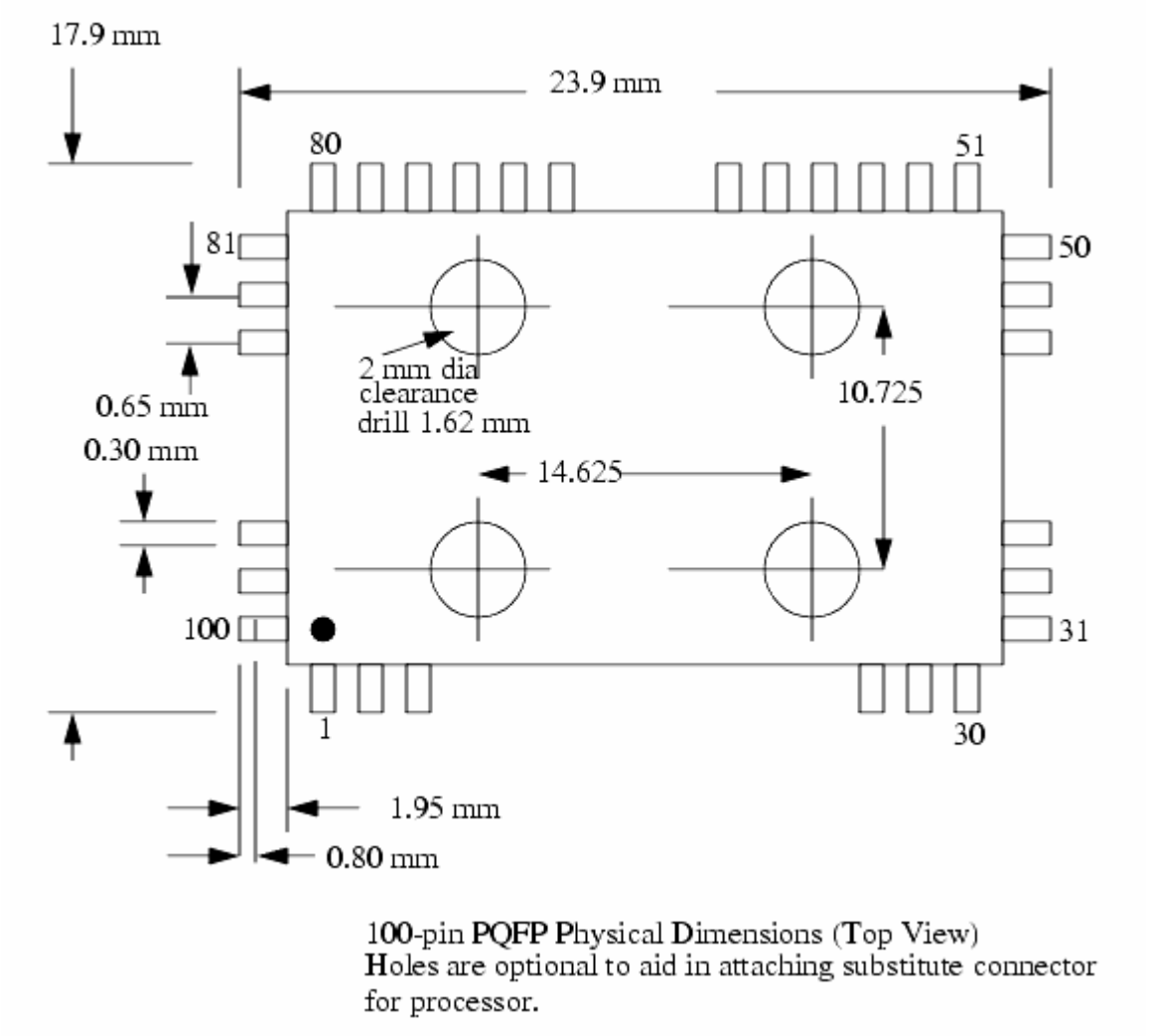


图 14 封装略图和引脚分配

5.2 封装的机械尺寸

图 15 表明 Rabbit 的 PQFP 封装的机械尺寸



(100-针 PQFP 物理尺寸(顶视)
孔是可选的,可辅助把处理器的代替连接器附加上)

图 15 Rabbit 的 PQFP 封装的机械尺寸

图 16 表示 PC 主板上的 100-针的 Rabbit 的 PQFP 的覆盖范围.

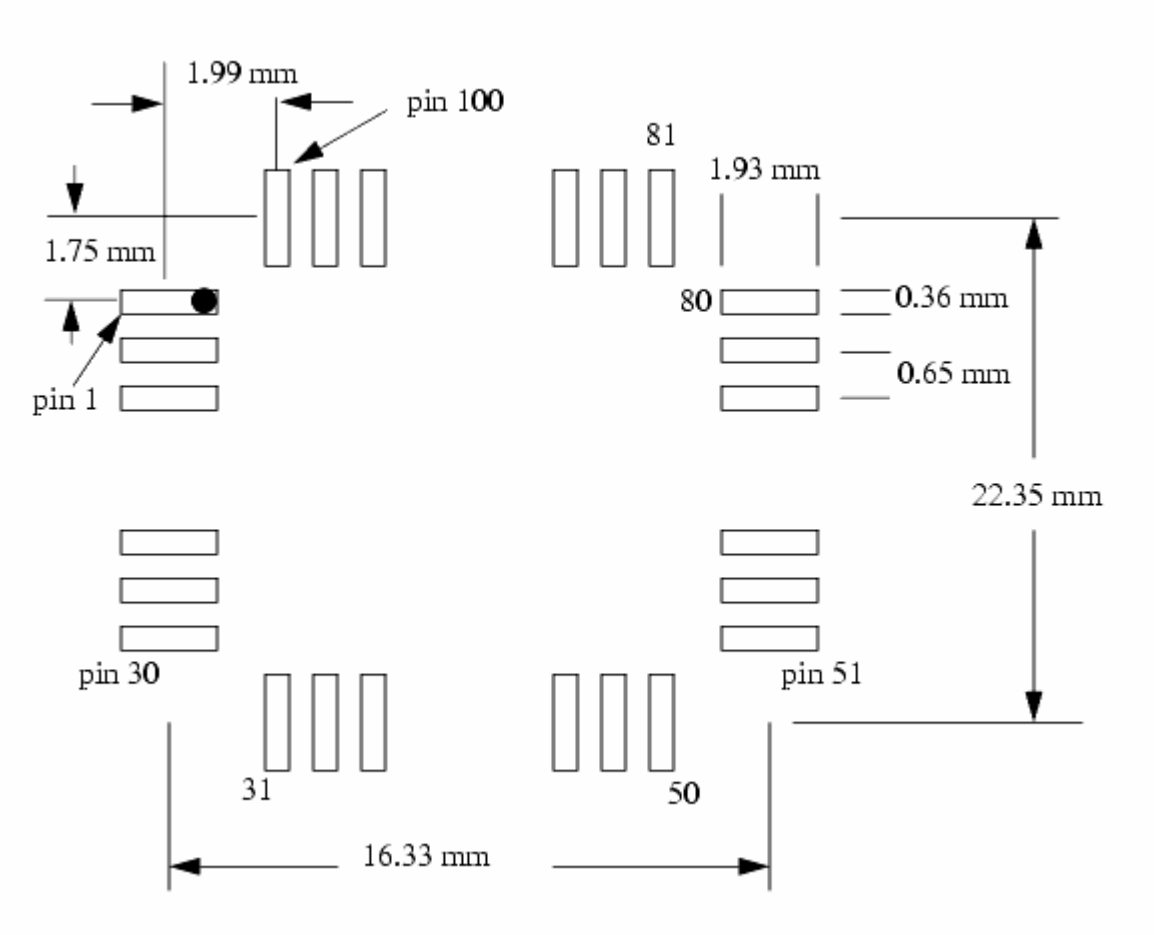


图 16 PC 主板上的 100-针的 Rabbit 的 PQFP 的覆盖面积

5.3 Rabbit 的引脚描述

表三列出器件的所有引脚，包括它们的方向、功能和引脚号。

表 3 Rabbit 引脚描述

引脚组	引脚名	方向	功　能	引脚号
硬件	CLK	输出	外部的时钟信号输出。这个信号来自内部的主系统振荡器即 perclk，可以编程内部电路进行 8 分频，或倍频，或者两者同时用。这个信号在复位之后有效。在程序控制下，这个引脚可以输出内部时钟频率，或 1/2 内部频率，还可以用作在软件控制下用于一般目的的输出引脚。 参看表 11“ 全局输出控制寄存器（GOCR=0Eh）”	1
	/RESET	输入	主处理器的复位	37
	XTALA1	输入	32KHz 时钟振荡器的石英晶体。连接到晶体的引线必须短，并有干扰保护。如果使用外部时钟信号，这个引脚应该由外部时钟驱动。	40
	XTALA2	输出	32KHz 晶振的石英晶体。如果使用外部时钟，不要连接这个引脚。	41
	XTALB1	输入	主频晶体振荡器。连接到晶体的引线必须短且有干扰保护。如果使用外部时钟，这个引脚由外部时钟驱动。	90
	XTALB2	输出	主频晶体振荡器。使用外部时钟则不用连接	
CPU 总线	A0-A19	输出	地址总线	7,17-20,61-68，70-75，79
	D0-D7	双向	数据总线	9-16
状态/控制	/WDTOUT	输出	WDT 超时——当内部看门狗（watchdog）超时，输出一个脉冲。也可以用于输出一个宽度 30 微秒的脉冲。	34
状态	STATUS	输出	对以下功能可编程： 1．在第一个操作码取出周期（fetch cycle）里被置为低电位（low） 2．在中断响应周期里被置为低电位 3．可作为一个通用用途的输出。 参看表 11“ 全局输出控制寄存器（GOCR=0Eh）”	38
状态	SMODE1 SMODE0	输入	启动模式选择（SMODE1 是 35 脚，SMODE0 是 36 脚），决定自引导过程。 （SMODE1=0，SMODE0=0）从地址 0 开始执行。 （0，1）由从端口冷启动。 （1，0）由时钟同步通讯的端口 A 冷启动。 （1，1）以 2400bps 的异步串行口 A 冷启动。 SMODE 引脚在冷启动结束之后可用作通用输出引脚。	35-36 （1：0）

（续上表）

芯片选择	/CS0	输出	内存芯片选择引脚 0——直接连接到静态存储器片选引脚。一般情况下，这个引脚用于选择保存程序的基片闪存（base flash memory)。	8
	/CS1	输出	内存芯片选择引脚 1—— 一般这个引脚直接连接到静态 RAM 的片选端。/CS1 可以在软件控制下被强制为连续的低电平。当电池供电的 RAM 的片选信号必须通过一个可能延迟时间长的控制器时，这个特性可帮助使用。	5
	/CS2	输出	内存芯片选择引脚 2——连接到静态存储器芯片。最后才使用这个片选引脚。	4
输出使能	/OE0	输出	内存输出使能引脚 0——直接连接到静态存储器芯片。	6
	/OE1	输出	内存输出使能引脚 1——备用的内存输出使能引脚，允许两个内存芯片共用片选	76
写使能（write enable)	/WE0	输出	内存写使能引脚 0——直接连接到静态存储器芯片。这个引脚可在软件控制下禁止，以对芯片写保护。	69
	/WE1	输出	内存写使能引脚 1——直接连接到静态存储器芯片。这个引脚可在软件控制下禁止，以对芯片写保护。	80
I/O 控制	/BUFEN	输出	I/O 缓冲区使能引脚——它的信号在外部 I/O 周期里被驱动为低电位，并可以用于控制总线缓冲区上的三态使能，目的是在每一个总线周期上不驱动 I/O 地址线或数据线以节能。	33
I/O 读选通	/IORD	输出	I/O 读选通。外部 I/O 读总线周期里驱动为低电位。也可以驱动 I/O 扩展有关的译码逻辑，或一个双向总线缓冲区上的方向引脚。参看端口 E 里的可编程选通引脚。	32
I/O 写选通	/IOWR	输出	I/O 写选通引脚。在外部 I/O 写周期里被驱动为低电平而作为写选通信号。由 I/O 体控制寄存器使能。参看端口 E 里的可编程选通引脚。	31
I/O 端口 A	PA0-PA7	输入/输出	这八个引脚作为通用目的的输入输出，或者作为从端口的数据端口。复位后这些引脚设置为输入，可浮空（float)。	81-88

（续上表）

I/O 端口 B	PB0-PB7	6 输入 2 输出	I/O 端口 B。作并行 I/O 使用时，PB7 和 PB6 只能作为输出。PB0-PB5 只能是输入。 PB0 和 PB1 如果被设置为定时串行端口的时钟信号端，则可以做输出。复位后，输出端置为 0。如果使用从端口，以下备用功能： PB7--/SLAVEATTN：从端口请求的注意信号 PB5，PB4—从寄存器的地址线（SA1，SA0） PB3—来自主处理器的从端口负读选通信号 PB2—来自主处理器的从端口负写选通信号 如果串口 A 设置为时钟同步，PB1 成为双向时钟线。如果串口 B 设置为时钟同步，PB0 成为双向时钟线。	93-100
I/O 端口 C	PC0-PC7	4 输入 4 输出	I/O 端口 C。当它作为一个并行端口使用时，1、3、5、7 是输入端，0、2、4、6 是输出端。0、2、4、6 可以交替选择成分别作为的串行端口 D、C、B、A 的串行数据输出。当这些输入正被串行口 UART 使用时，也可以从并行端口寄存器读这些输入的值。	51，54-60
I/O 端口 D	PD0-PD7	输入/输出/输出 open drain	I/O 端口 D。每一位都可被单独选择作为输入或输出。每个输出可选择为高/低驱动或漏极开路。输出可与定时器同步的寄存器进行缓冲以获得精确边沿控制。PD6 可编程为一个串行端口 A 的可选择的串行输出。PD4 可编程为一个串行端口 B 的可选择的串行输出。PD7 和 PD5 可由串行口 A 和 B 作为备用的串行输入使用，这时这些引脚必须编程为输入。	43-50

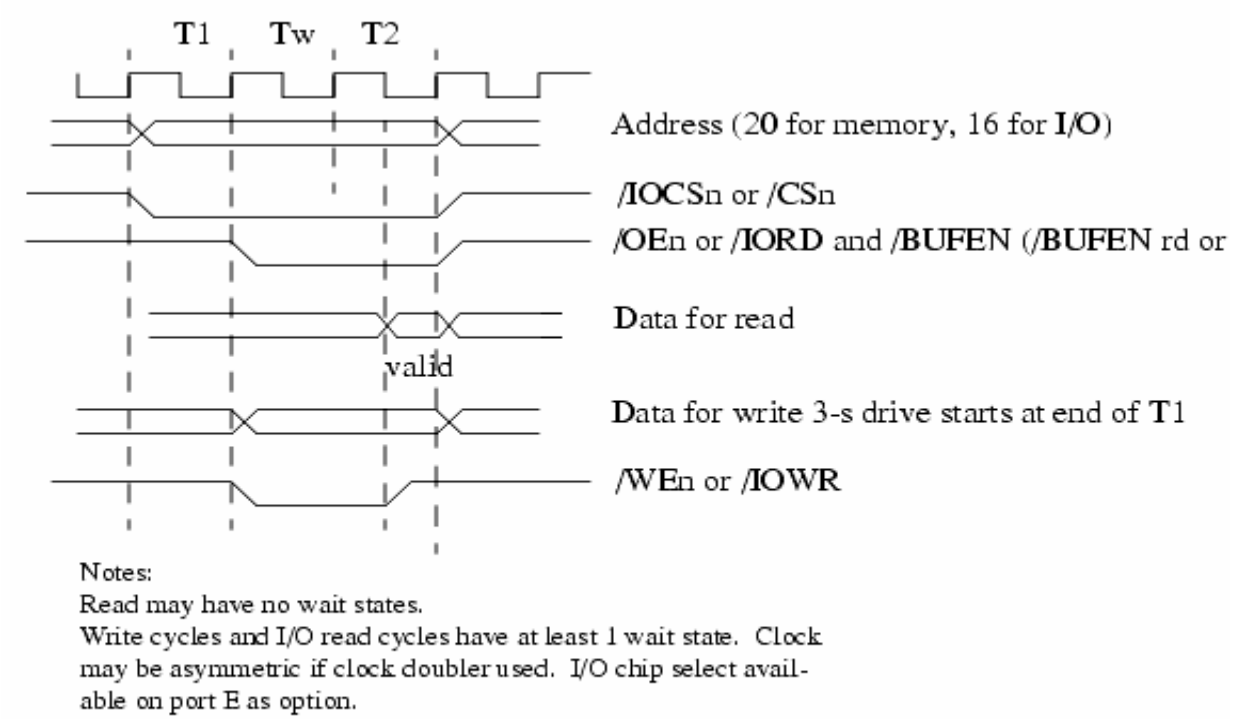
I/O 端口 E	PE7- PE0	输入/输出	I/O 端口 E。每一位可被单独选择作为输入或输出。输出可与定时器同步的寄存器缓冲以获得精确边沿控制。每根端口线都可被单独选择作为一个 I/O 控制信号线,而不是一根并行I/O 线。这 8 根可能的 I/O 控制信号线的任一根，都是一个在外部 I/O 周期时能够用作选通信号，选通 64K 外部 I/O 空间的 1/8。每个选通信号可编程为一个片选、一个写选通、一个读选通或一个复合读写选通。任何作为 I/O 控制选通使用的端口位必须编程为一个输出位。如果从端口被使能，PE7 作为从寄存器片选信号线（负极性有效）使用。PE7 应该编程为一个输入，以使从寄存器的片选功能工作。如果 PE7 编程为一个输出并设置为低电平，那么从寄存器片选引脚会一直被激活。PE0 和 PE4 作为外部中断 0 的备用输入使用。PE1 和 PE5 作为外部中断 1 的备用输入使用。如果 PE0 使能，那么 PE1 也应该使能，同样 PE4 和 PE5 也应该如此。中断可以设置为下降沿、上升沿或两个边沿触发。如果两个中断都允许，在对每个输入都进行边沿探测之后，对这两个中断进行“或”操作。端口位必须配置为输入以把它们作为中断请求输入使用。	21-26,29,30
电源	VBAT		+3.0V(有电池后备),+3.3V 或+5.0V	42
	VDD		+3.3V 或+5.0V	3,28,53 78,92
	VSS		地	2，27，39，52,77,89
串行端口	CLKA	输入/输出	工作于同步模式时串行端口 A 的时钟。PB1 的备用分配。	94
	CLKB	输入/输出	工作于同步模式时串行端口 B 的时钟。PB0 的备用分配。	93
	RXA,TXA RXB,TXB RXC,TXC RXD,TXD	RX-输入 TX-输出	串行端口 A-D 的串行输入和输出。并行端口 C 的备用引脚分配。	51，54-60
	ARXA,ATXA ARXB,ATXB	RX-输入 TX-输出	串行端口 A 和 B 的备用串行输入和输出。它们是并行端口 D 的 PD4-PD7 的备用引脚分配。	43-46

（续上表）

从端口	SD0-SD7	双向	从端口的数据总线。并行端口 A 的备用分配。	81-88
	/SLAVEATTN	输出	/SLAVEATTN—从处理器正在向主处理器请求注意。它是并行端口 B 的位 7 的备用引脚分配。	100
	/SRD	输入	选通端，用于读某一个从寄存器。它是并行端口 B 的位 3 的备用引脚分配。	96
	/SWR	输入	选通端，用于写从寄存器。并行端口 B 的位 2 的备用引脚分配。	95
	SA0，SA1	输入	寻址从寄存器的地址线。并行端口 B 位 4、5 位的备用引脚分配。	97，98
	/SCS	输入	从端口的片选引脚，低电位有效。并行端口 E 的位 7 的备用引脚分配。	21
I/O 选通	/I0，/I1 /I2，/I3 /I4，/I5 /I6，/I7	输出	I/O 选通引脚。每个选通脚使用 I/O 地址空间的 1/8 或者说 8K 地址。每个选通脚可编程为：芯片选择，读，写，复合读写。这些都是并行端口 E 的 0-7 位的备用引脚分配。每个引脚可单独重新分配，由并行端口转为选通功能。	21-26，29，30
外部中断 0	INT0A，INT0B	输入	采样这些引脚，检测到一个指定转变（pos,neg 任一）时锁存一个对外部中断号 0 的中断请求。每个引脚有单独的锁存器。当这个引脚配置为并行端口 E 的输入，可被使能。引脚的值可通过并行端口读取，使用并行端口的 0，1 位。如果并行端口设置为输出，并行端口的输出可以用于产生中断。	24，30
外部中断 1	INT1A，INT1B	输入	采样这些引脚，检测到一个指定转变（pos,neg 任一）时锁存一个对外部中断号 1 的中断请求。每个引脚有单独的锁存器。当这个引脚配置为并行端口 E 的输入，可被使能。引脚的值可通过并行端口读取，使用并行端口的 4，5 位。如果并行端口配置为输出，并行端口的输出可用于产生中断。	23，29

5.4 总线时序(Bus Timing)

外部总线实质上具有和内存周期或 I/O 周期相同的时序。一个内存周期开始于片选信号和地址信号有效。一个时钟周期过后，如果进行一个读操作输出允许信号有效。如果进行一个写操作则写允许信号有效。



（注意：读操作可以没有等待状态。

写周期和 I/O 读周期至少有一个等待状态。如果使用时钟倍频器，时钟可能是非对称的。

作为选择项，可在端口 E 进行上 I/O 片选。）

图 17 读和写总线时序

有些情况，图 17 中表示的时序可以由在第一个时钟脉冲（后面跟随图 17 中表明的访问次序）中的虚假内存访问作前缀。在这种情况下，地址以及很多时候的片选信号会在一个时钟脉冲之后改变值，而且假定内存的最终值实际被访问了。输出允许和写允许通常在最终的稳定地址和片选使能后，延迟一个时钟脉冲进行。一般，虚假内存访问试图开启另一个指令周期，这个指令周期在处理器认识到需要一个读数据或写数据总线周期之后的一个时钟脉冲里被中止。用户不要在没有把这些情况考虑在内时，试图做一种使用片选信号或内存地址信号作为时钟或状态改变信号的设计。

5．5 引脚描述（带备用功能）

见表四。

表 4 带备用功能的引脚

引脚名称	输出功能	输入功能	其他功能
STATUS (38)	1.第一次取操作码时为低电位 2.中断响应时为低电位		可编程的输出端口，由程序置高或低电平
SMODE1 (35)		(SMODE1,SMOD2) 启动时引导模式控制	引导完成后为 1-位输入
SMODE2 (36)		(SMODE1,SMODE2) 启动时引导模式控制	引导完成后为 1-位输入
CLK(1)	1.外设时钟 2.外设时钟的 1/2		可编程输出端口，由程序置高或低
/WDTOUT (34)	看门狗超时则输出 30.5 微秒的脉冲（处理器同时复位）		在程序控制下输出一个 30.5 微秒到 61 微秒的脉冲
PA7(88)	SD7	SD7	
PA6(87)	SD6	SD6	
PA5(86)	SD5	SD5	
PA4(85)	SD4	SD4	
PA3(84)	SD3	SD3	
PA2(83)	SD2	SD2	
PA1(82)	SD1	SD1	
PA0(81)	SD0	SD0	
PB7(100)	/SLAVEATTN(从处理器要求主处理器注意)		
PB5(98)		SA1(从地址)	
PB4(97)		SA0	
PB3(96)		/SRD(主处理器的读从寄存器的选通信号)	
PB2(95)		/SWR(主处理器写从寄存器的选通信号)	

(续上表)

PB1(94)	**CLKA(串行端口 A 的时钟同步的移位时钟，双向)**	**CLKA**	
PB0(93)	**CLKB(双向)**	**CLKB**	
PC7(51)		**RXA**	
PC6(54)	**TXA**		
PC5(55)		**RXB**	
PC4(56)	**TXB**		
PC3(57)		**RXC**	
PC2(58)	**TXC**		
PC159)		**RXD**	
PC0(60)	**TXD**		
PD7(43)		**ARXA**	
PD6(44)	**ATXA**		
PD5(45)		**ARXB**	
PD4(46)	**ATXB**		
PD3(47)			
PD2(48)			
PD1(49)			
PD0(50)			
PE7(21)	**/I7—可编程的 I/O 选通信号**	**/SCS(从芯片选择)**	
PE6(22)	**/I6**		
PE5(23)	**/I5**	**INT1(输入)**	
PE4(24)	**/I4**	**INT0(输入)**	
PE3(25)	**/I3**		
PE2(27)	**/I2**		
PE1(29)	**/I1**	**INT1(输入)**	
PE0(30)	**/I0**	**INT0(输入)**	

5.6 DC 特性(DC Characteristics)

5.6.1 5.0 伏

表 5 概要指出在建议的工作温度范围即 Ta=-40°C到 85°C上,V_{DD}=4.5V 到 5.5V 情况下，Rabbit 在 5.0V 上的 DC 特性。

表 5 5.0V DC 特性

标志	参数	测试条件	最小值	类型	最大值	单位
I_{IH}	输入损耗高	$V_{IN}=V_{DD}$,V_{DD}=5.5V			10	μA
I_{IL}	输入损耗低(无上拉电阻)	$V_{IN}=V_{SS}$,V_{DD}=5.5V	-10			μA
I_{OZ}	输出损耗(无上拉电阻)	$V_{IN}=V_{DD}$ 或 V_{SS}，V_{DD}=5.5V	-10		10	μA
V_{IL}	CMOS 输入低电压				0.3x V_{DD}	V
V_{IH}	CMOS 输入高电压		0.7x V_{DD}			
V_T	CMOS 开关阈值	V_{DD}=5.0V, 25°C		2.4		V
V_{OL}	CMOS 输出低电压	I_{OL}=参看表 7 (吸收) V_{DD}=4.5V		0.2	0.4	V
V_{OH}	CMOS 输出高电压	I_{OH}=参看表 7 (送出) V_{DD}=4.5V	0.7x V_{DD}	4.2		V

5．6 3.3V 特性

表 6 概要指出在建议的工作温度范围即 Ta=-40°C到 85°C上,V_{DD}=2.7V 到 3.6V 情况下，Rabbit 在 3.3V 上的 DC 特性。

表 6 3.3V DC 特性

标志	参数	测试条件	最小值	类型	最大值	单位
I_{IH}	输入损耗高	$V_{IN}=V_{DD}$,V_{DD}=3.6V			5	μA
I_{IL}	输入损耗低(无上拉电阻)	$V_{IN}=V_{SS}$,V_{DD}=3.6V	-5			μA
I_{OZ}	输出损耗(无上拉电阻)	$V_{IN}=V_{DD}$ 或 V_{SS}，V_{DD}=3.6V	-5		5	μA
V_{IL}	CMOS 输入低电压				0.3x V_{DD}	V
V_{IH}	CMOS 输入高电压		0.7x V_{DD}			
V_T	CMOS 开关阈值	V_{DD}=3.0V,25°C		1.5		V
V_{OL}	CMOS 输出低电压	I_{OL}=参看表 5 (吸收) V_{DD}=2.7V		0.11	0.4	V
V_{OH}	CMOS 输出高电压	I_{OH}=参看表 5 (送出) V_{DD}=2.7V	0.7x V_{DD}	2.3		V

5.7 I/O 缓冲区送出和吸收（sourcing and sinking）电流的限制

除非另外规定，Rabbit 的 I/O 缓冲区具有在全速 AC 切换时在每个引脚上送出和吸收 8mA 电流的能

力。全速 AC 切换假定 22.11Mhz 的 CPU 时钟和地址及数据线每个引脚低于 100pF 的电容性负载。地址引脚 A0 和数据引脚 D0 的额定值分别是 16mA。
表 7 表明并行 I/O 缓冲区的 AC 和 DC 输出驱动限制。

表 7 I/O 缓冲区送出和吸收电流的能力

引脚名称	输出驱动 送出[1]/吸收[2]限制（mA）	
输出端口名称	全速 AC 切换 SRC/SNK	最大[3]DC 输出驱动 SRC/SNK
PA[7:0]	8/8	12/12
PB[7,1,0]	8/8	12/12
PC[6,4,2,0]	8/8	12/12
PD[7:4]	8/8	12/12
PD[3:0][4]	16/16	25/25
PE[7:0]	8/8	12/12

[1] V_{DD} 引脚之间的用于 I/O 缓冲区的最大直流送出电流是 112mA。
[2] V_{SS} 引脚之间的用于 I/O 缓冲区的最大直流吸收电流是 150mA 。
[3] I/O 缓冲区上的最大直流输出驱动必须为考虑 AC 切换输出、切换输出时的电容性负载和切换电压所要求的电流需求而做相应的调整。起因于 AC 切换的电流，是当 AC 切换发生时流通的平均电流。它可以通过 I=CVf 计算，此处 f 是每秒的转换数，C 是切换的电容，V 是电压摆幅。举个例子，如果每秒发生 12，000，000 次转换，5V 的摆幅并驱动 100pF 的电容，那么每个引脚 I=6mA。起因于所有引脚与电源或地之间的电流（包括起因于切换电流或直流电流的电流），应该求一个和以测试这些界限值。
所有由开关和非开关 I/O 抽出的电流不允许超过注释 1 和 2 里给定的界限。
[4] 端口 D[7:0]的联合送出电流可能需要调整，以不要超出注释 1 里给定的 112mA 的送出界限要求

6．Rabbit 内部 I/O 寄存器

表 8　Rabbit 内部 I/O 寄存器

地址	复位值	功能性
GCSR=00h	1100 0000	全局控制状态寄存器。对时钟、周期性中断的控制和对看门狗的监视。参看表 9。
RTCCR=01h	0000 0000	实时时钟控制寄存器。 参看 7.5 小节
RTC0R=02h	xxxx xxxx	实时时钟字节 0 寄存器
RTC1R=03h	xxxx xxxx	实时时钟字节 1 寄存器
RTC2R=04h	xxxx xxxx	实时时钟字节 2 寄存器
RTC3R=05h	xxxx xxxx	实时时钟字节 3 寄存器
RTC4R=06h	xxxx xxxx	实时时钟字节 4 寄存器
RTC5R=07h	xxxx xxxx	实时时钟字节 5 寄存器
WDTCR=08h	0000 0000	看门狗定时器控制寄存器。参看 7.6 小节
WDTTR=09h	0000 0000	看门狗定时器测试寄存器
GOCR=0Eh	0000 0x00	全局输出控制控制寄存器。参看 7.4 小节
GCDR=0Fh	xxxx x000	全局时钟倍加器寄存器
MMIDR=10h	Xxx0 0000	内存管理 I 和 D 空间寄存器。控制 I&D 空间的使能和/CS1 的电池切换的支持
XPC	0000 0000	非 I/O 寄存器，但在复位时被初始化为 0。
STACKSEG=11h (Z180 CBR)	0000 0000	堆栈段的内存指针。指明堆栈段在物理内存中的位置。
DATASEG=12h (Z180 BBR)	0000 0000	数据段的内存指针。指明数据段在物理内存中的位置。

(续上表)

SEGSIZE=13h (Z180 CBAR)	1111 1111	指明 64K 内存空间里数据段的开始和堆栈段的开始
MB0CR=14h	0000 0000	存储体 0 控制寄存器。控制第一内存象限的 256K 到物理内存芯片的映射
MB1CR=15h	xxxx xxxx	存储体 1 控制寄存器。控制第二内存象限到物理内存芯片的映射。
MB2CR=16h	xxxx xxxx	存储体 2 控制寄存器。控制第三内存象限到物理内存芯片的映射。
MB3CR=17h	xxxx xxxx	存储体 3 控制寄存器。控制第四内存象限到物理内存芯片的映射。
SPD0R=20h	xxxx xxxx	从端口寄存器 0。使用于从端口通讯的用于读和写的独立寄存器
SPD1R=21h	xxxx xxxx	从端口寄存器 1
SPD2R=22h	xxxx xxxx	从端口寄存器 2
SPSR=023h	0000 0000	从端口状态寄存器
SPCR=24h	000x 0000	从端口控制寄存器
PADR=30h	xxxx xxxx	并行端口 A 数据寄存器，R/W。
PBDR=40h	00xx xxxx	并行端口 B 数据寄存器，R/W。
PCDR=50h	x0x0 x0x0	并行端口 C 数据寄存器。
PCFR=55h	x0x0 x0x0	端口 C 功能寄存器。
PDDR=60h	xxxx xxxx	并行端口 D 数据寄存器，R/W。
PDCR=64h	xx00 xx00	端口 D 控制寄存器
PDFR=65h	xxxx xxxx	端口 D 功能寄存器
PDDCR=66h	xxxx xxxx	端口 D 驱动控制寄存器
PDDDR=67h	0000 0000	端口 D 数据方向寄存器
PDB0R=68h	xxxx xxxx	端口 D 第 0 位的寄存器，W。
PDB1R=69h	xxxx xxxx	端口 D 第 1 位的寄存器
PDB2R=6Ah	xxxx xxxx	第 2 位。
PDB3R=6Bh	xxxx xxxx	第 3 位
PDB4R=6Ch	xxxx xxxx	第 4 位
PDB5R=6Dh	xxxx xxxx	第 5 位
PDB6R=6Eh	xxxx xxxx	第 6 位
PDB7R=6Fh	xxxx xxxx	第 7 位
PEDR=70h	xxxx xxxx	并行端口 E 数据寄存器。读/写
PECR=74h	xx00 xx00	端口 E 控制寄存器
PEFR=75h	xxxx xxxx	端口 E 功能寄存器
PEDDR=77h	0000 0000	端口 E 数据方向寄存器
PEB0R=78h	xxxx xxxx	端口 E 第 0 位寄存器，W

(续上表)

PEB1R=79h	**xxxx xxxx**	**第 1 位**
PEB2R=7Ah	**xxxx xxxx**	**第 2 位**
PEB3R=7Bh	**xxxx xxxx**	**第 3 位**
PEB4R=7Ch	**xxxx xxxx**	**第４位**
PEB5R=7Dh	**xxxx xxxx**	**第 5 位**
PEB6R=7Eh	**xxxx xxxx**	**第 6 位**
PEB7R=7Fh	**xxxx xxxx**	**第 7 位**
IB0CR=80h	**0000 0xxx**	**外部 I/O 控制体 0**
IB1CR=81h	**0000 0xxx**	**外部 I/O 控制体 1**
IB2CR=82h	**0000 0xxx**	**外部 I/O 控制体 2**
IB3CR=83h	**0000 0xxx**	**外部 I/O 控制体 3**
IB4CR=84h	**0000 0xxx**	**外部 I/O 控制体 4**
IB5CR=85h	**0000 0xxx**	**外部 I/O 控制体 5**
IB6CR=86h	**0000 0xxx**	**外部 I/O 控制体 6**
IB7CR=87h	**0000 0xxx**	**外部 I/O 控制体 7**
I0CR=98h	**xx00 0000**	**外部中断 0 控制寄存器**
I1CR=99h	**xx00 0000**	**外部中断 1 控制寄存器**
TACSR=0A0h	**0000 xx00**	**定时器 A 控制/状态寄存器**
TACR=0A4h	**xxxx xxxx**	**定时器 A 控制寄存器**
TAT1R=0A3h	**0000 xx00**	**定时器 A1 时间常数 1 寄存器**
TAT4R=0A9h	**xsxxx xxxx**	**定时器 A4 时间常数 4 寄存器**
TAT5R=0ABh	**xxxx xxxx**	**定时器 A5 时间常数 5 寄存器**
TAT6R=0ADh	**xxxx xxxx**	**定时器 A6 时间常数 6 寄存器**
TBCSR=0B0h	**xxxx xx00**	**定时器 B 控制/状态寄存器**
TBCR=0B1h	**xxxx 0000**	**定时器Ｂ控制寄存器**
TBM1R=0B2h	**xxxx xxxx**	**定时器 B 的 MSB（最高有效字节）1 寄存器**
TBL1R=0B3h	**xxxx xxxx**	**定时器 B 的 LSB（最低有效字节）1 寄存器**
TBM2R=0B4h	**xxxx xxxx**	**定时器 B 的 MSB 2 寄存器**
TBL2R=0B5h	**xxxx xxxx**	**定时器 B 的 LSB 2 寄存器**
TBCMR=0BEh	**xxxx xxxx**	**定时器 B 计数 MSB 寄存器**
TBCLR=0BFh	**xxxx xxxx**	**定时器 B 计数 LSB 寄存器**

（续上表）

SADR=0C0h	xxxx xxxx	串行端口 A 数据寄存器，接收/发送。
SAAR=0C1h	xxxx xxxx	串行端口 A 备用数据寄存器（传送第 9 位）
SASR=0C3h	0xx0 0000	串行端口 A 状态寄存器
SACR=0C4h	xx00 0000	串行端口 A 控制寄存器
SBDR=0D0h	Xxxx xxxx	串行端口 B 数据寄存器，接收/发送。
SBAR=0D1h	Xxxx xxxx	串行端口 B 备用数据寄存器（传送第 9 位）
SBSR=0D3h	0xx0 0000	串行端口 B 状态寄存器
SBCR=0D4h	xx00 0000	串行端口 B 控制寄存器
SCDR=0E0h	xxxx xxxx	串行端口 C 数据寄存器，接收/发送
SCAR=0E1h	xxxx xxxx	串行端口 C 备用数据寄存器（传送第 9 位）
SCSR=0E3h	0xx0 0000	串行端口 C 状态寄存器
SCCR=0E4h	xx00 x000	串行端口 C 状态寄存器
SDDR=0F0h	xxxx xxxx	串行端口 D 数据寄存器，接收/发送
SDAR=0F1h	xxxx xxxx	串行端口 D 备用数据寄存器（传送第 9 位）
SDSR=0F3h	0xx0 0000	串行端口 D 状态寄存器
SDCR=0F4h	xx00 x000	串行端口 D 控制寄存器

7．其他 I/O 功能

7．1 Rabbit 振荡器和时钟

Rabbit 处理器内部有两个晶体振荡器。主振荡器允许最高频率是 29.4912MHz 的晶体（第一个只能是谐波晶体（overtone crystal））。时钟振荡器需要一个 32.768KHz 的晶体，由 VBAR 供电，也可由电池供电。

通过把外部时钟连接到 XTALA1 或 XTALB1 并使其他晶体引脚（XTALA2 或 XTALB2）处于未连接状态，可以用一个外部振荡器或时钟代替任一晶体。如果用一个外部振荡器作为主时钟信号，当时钟需要从外部获得时，输出引脚 CLK（引脚 1）应该被使用。这个信号与内部时钟同步。作为对比，内部时钟比外部振荡器输入 XTALB1 滞后 10 纳秒。

主振荡器一般作为处理器和外设时钟信号源。32.768KHz 的振荡器一般用于为看门狗定时器、可用电池供电的时间/日期时钟和周期性中断提供时钟信号。主振荡器可以在一种特别的低能耗模式下切断，然后就使用 32.768KHz 的振荡器为通常由主振荡器提供时钟的部件提供时钟。这导致低能耗状态下（~200 微安）的较慢的执行速度。

芯片上的时钟布线法示于图 18。主振荡器的频率可加倍和/或八分频。如果同时使能倍频和分频，则

总结果是四分频。CPU 的时钟可选择性地二分频，然后选择性地驱动外部引脚 CLK。很多情况下不需要从外部引入时钟，此时 CLK 可以用作一个通用用途的输出引脚。这个二分频的选择项在时钟从芯片上分离的情况下，可以使电磁辐射最小化。

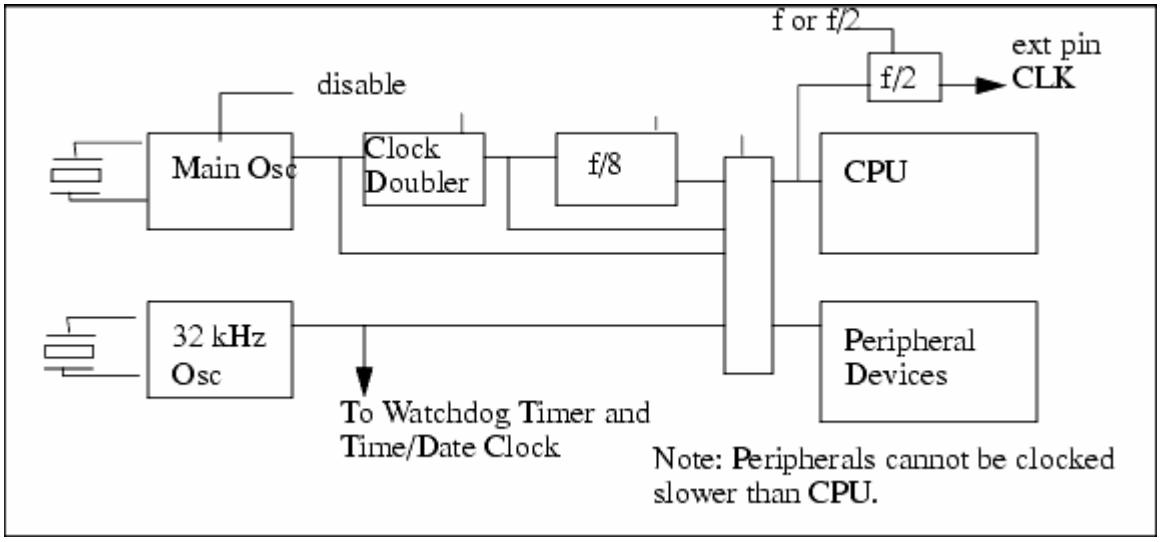


图 18　时钟分布

表 9 全局控制/状态寄存器（I/O 地址=00h）

位	值	描　　述
7:6	00	读过后，没有发生复位或看门狗定时器。
（只读）	01	看门狗定时器超时。读这个寄存器将这两位清零。
	10	不可能的位组合。
	11	复位发生。读这个寄存器清零将这两位清零。
5（只写）	0	读这个寄存器清除周期性的中断请求。这个位总是读作 0。
	1	强迫一个周期性中断
4:2（只写）	000	处理器时钟来自主振荡器，八分频。 外设时钟来自主振荡器，八分频。
	001	处理器时钟来自主振荡器，八分频。 外设时钟来自主振荡器，不分频。
	01x	处理器时钟来自主振荡器，不分频。 外设时钟来自主振荡器，不分频。
	1x0	处理器时钟来自 32KHz 振荡器，不分频。 外设时钟来自 32KHz 振荡器，不分频。
	1x1	处理器时钟来自 32KHz 振荡器，不分频。 外设时钟来自 32KHz 振荡器，不分频。 主振荡器关闭。

（续上表）

1:0(只写)	00	周期性中断被禁止。
	01	周期性中断使用中断优先级 1。
	10	周期性中断使用中断优先级 2。
	11	周期性中断使用中断优先级 3。

7．2 时钟倍频器（clock doubler）

使用时钟倍频器允许主振荡器使用较低频率的晶体，还可以提供一个附加的时钟频率调整范围。时钟倍频器使用一个芯片内的延迟电路，如果需要倍频，它必须由使用者在启动时进行编程。表 10 列出不同晶体频率的推荐延迟值。注意，“ 最佳”的振荡器频率已经反映了实际晶体频率的倍频。

表 10 全局时钟倍频器寄存器（GCDR，地址=0fh）

位	值	描述
7:3	xxxx	这几位被忽略。
2:0	000	时钟倍频电路被禁止。
	001	8 ns 的额定低电位时间（用于 30MHz 振荡器最佳）
	010	10ns 的额定低电位时间（用于 25MHz 振荡器最佳）
	011	12ns 的额定低电位时间（用于 20MHz 振荡器最佳）
	100	14ns 的额定低电位时间（用于 18MHz 振荡器最佳）
	101	16ns 的额定低电位时间（用于 16MHz 振荡器最佳）
	110	18ns 的额定低电位时间（用于 14MHz 振荡器最佳）
	111	20ns 的额定低电位时间(用于 12MHz 或更慢的振荡器最佳）

当使用了时钟倍频器并且随后也没有对时钟进行分频，输出时钟将会不对称，如图 19 所示。倍频时钟的低电位时间以宽（50%）偏差为条件，因为它依赖于处理器参数、温度和电压。上面给出的时间适用于 5V 的电源电压和 25℃温度条件。倍频时钟低电位时间在电压减小到 4V 时增加 20%，如果进一步减小到 3.3V，增加 40%。温度每增减 5℃，时间增减 1%。倍频时钟通过对延迟、反转时钟和它自身进行异或运算得到。如果原始时钟不具有 50-50 的占空比，那么备用时钟的长度将会有稍微不同。由于内置振荡器的占空比可以是非对称 的 55-45，时钟倍频器产生的时钟将在备用时钟上表现周期性的最高可达 10%的偏差。这不影响无等待状态的内存访问时间，因为它一般都使用邻近的两个时钟。然而，如果时钟脉冲由时钟倍频器提供，最高允许时钟速度必须减小 10%。唯一由时钟脉冲的下降沿同步的信号，是内存和 I/O 写脉冲，而且这些信号有非临界时序（noncritical timing)。这样，只要时钟的低电位时间没到过度地缩短时钟高电位时间长度的程度——这会使内存不能使用这样的过短的写脉冲，时钟的低电位时间就没到临界值。在实际的时钟速度和典型静态 RAM 存储器情况下，这不太可能发生。

功耗与时钟频率成比例，由于这个原因，在只有较少的计算活动时，可以通过降低时钟频率来降低功耗。时钟倍频器提供了一种作为能量管理方案的临时提高或降低时钟速度的方便的方法。

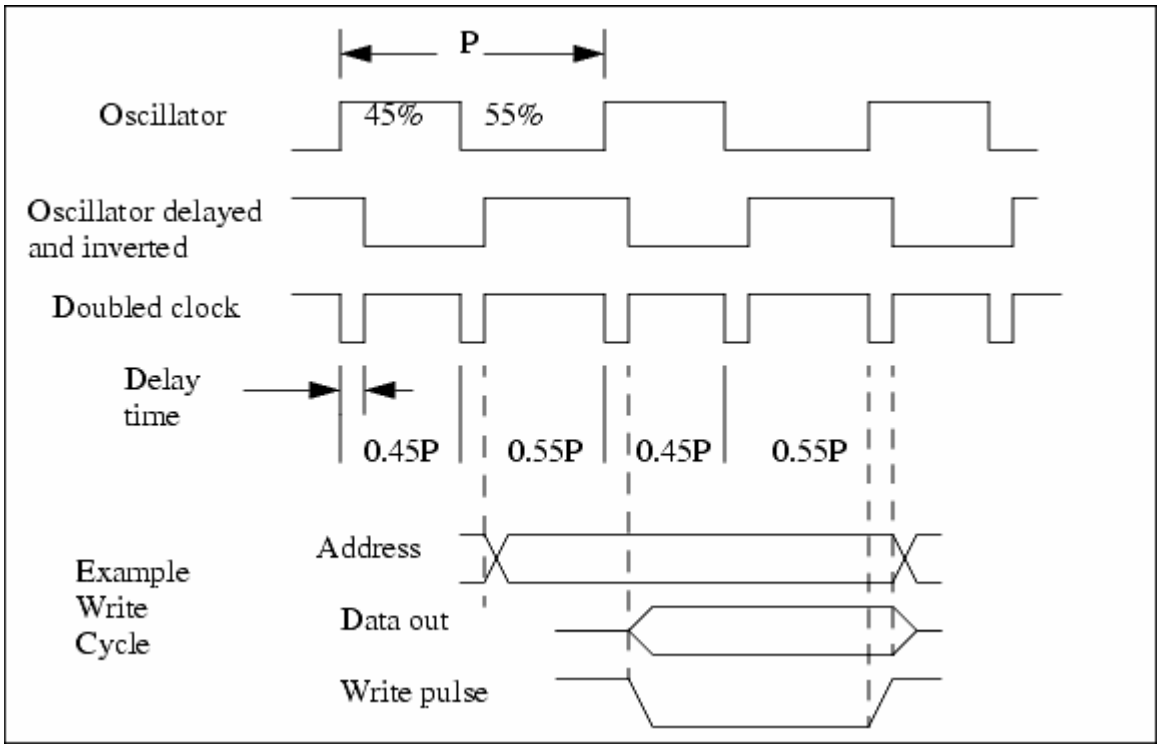


图 19 时钟倍频器的作用

7．3 控制功耗(Controling Power Consumption)

处理器的功耗可同速度进行代价交换，可通过减慢系统时钟、增加等待状态、使用低功耗指令等方法；而且，为了最大限度的节省功耗，可禁止主系统振荡器，而使用实时时钟振荡器提供时钟脉冲。可以使能以下节能特征。

- **给取指令操作附加内存等待状态。等待状态数可编程为 0,1,2 或 4。一般情况，两个等待状态应该用去零等待状态的一半能量。**
- **如果还没有使用时钟倍频器，把处理器和外设的时钟都四分频。如果没有什么部件（尤其是定时器和串行端口）依赖于外设时钟，这样做是允许的。**
- **如果使用了时钟倍频器，关闭它，把处理器和外设时钟都二分频。**
- **把处理器和/或外设时钟八分频**
- **在 RAM 里运行指令，而不要在闪存里运行。**
- **把处理器和外设时钟切换到 32.768kHz 振荡器。如果需要，禁止主振荡器。**
- **执行一个低能耗指令循环，这个循环大部分由不需要很多能耗的指令组成。最好的选择是连续的进行 0x0 乘法操作的 *mul* 指令。在第一条 *mul* 之后，不需要另外的干预指令来载入乘法项，因为所有相关寄存器都保持为 0。**

我们期望这些把电流消耗减少到 25 微安再加一些损耗的措施，在高温操作时有大用处。

7．4 输出引脚 CLK，STATUS，/WDTOUT，/BUFEN

某些特定的引脚可具有后备任务分配（alternate assignment），如表 11 指明的。

表 11 全局输出控制寄存器（GOCR=0Eh）

位	值	描述
7:6	00	CLK 引脚由外设时钟驱动。
	01	CLK 引脚由二分频的外设时钟驱动.
	10	CLK 引脚为低电位
	11	CLK 引脚为高电位
5:4	00	STATUS 引脚在一个操作码首字节取出期间有效（低）
	01	STATUS 引脚在中断响应期间有效（低）
	10	STATUS 引脚低
	11	STATUS 引脚高
3	1	WDTOUTB 引脚低电平（1 个周期最小，2 个周期最大，时钟 32kHz）。
	0	WDTOUTB 引脚从事看门狗功能。
2	x	这一位忽略。
1:0	00	/BUFEN 引脚在外部 I/O 周期内有效（低）。
	01	/BUFEN 引脚在数据内存访问期间有效（低）
	10	/BUFEN 低
	11	/BUFEN 高

7．5 时间/日期时钟（实时时钟）

时间/日期时钟（RTC）是一个 48 位（脉动）计数器，由 32.768kHz 振荡器驱动。RTC 是一个改进的脉动计数器，由六个分离的 8 位计数器构成。进位同时注入所有六个 8 位计数器，然后脉动 8 位。每一位的脉动时间为几纳秒，则 8 位的脉动总时间不会超过 200ns，即使在低电压下工作。

48 位的计数器在 32kHz 时钟频率下足够计数 272 年。按照惯例 1980 年 1 月 1 日上午 12 点作为时间零点。Z-World 软件忽略最高顺序位，使计数器的计时能力为 136 年。要读计数器的值，这个值首先应传输到一个 6 字节的保持寄存器，然后可以从保持寄存器读取单个字节。为执行这个传输，任何数据位都写到 RTC0R，即第一个保持寄存器。然后计数器可以在 RTC0R 到 RTC5R 上作为六个 8 位字节被读取。计数器和 32kHz 振荡器由独立的电源引脚供电，在芯片其他部分都断电之后，它们仍旧可以工作。这种设计可以使用电池供电。由于处理器以不同于 RTC 的时钟运行，进位正在进行时执行到保持寄存器的传输是可能的，这导致不正确的信息。为了防止这个错误发生，处理器应该执行两次读时钟操作，并确定两次读的值相同。

如果处理器本身运行在 32kHz 上，读时钟过程必须修改，因为低速时钟驱动的处理器读时钟期间会发生大量的时钟计数。一种合适的修改是忽略低字节，只读高顺序位的 5 个字节，它们每 256 个时钟周期或 1/128 秒计数一次。如果读操作在这个时候不能进行，可以忽略更多的低顺序位。

RTC 寄存器不能由一次写操作设置，但它们可以单独清零和计数。照这样，任何寄存器或整个 48 位的计数器都可以在 256 步之内设置为任意值。如果没有安装 32kHz 晶体而且其输入引脚接地，不会产生计数，这六个寄存器可以用作一个电池供电的小型内存。通常这样不会做，如果为 RTC 提供了电源切换的电路，也可以用于为常规的低功耗静态 RAM 提供电池供电。

表 12 实时时钟读寄存器

实时时钟 x 保持寄存器	（RTC0R）R/W	（地址=0000 0010）
	（RTC1R）	（地址=0000 0011）
	（RTC2R）	（地址=0000 0100）
	（RTC3R）	（地址=0000 0101）
	（RTC4R）	(地址=0000 0110)
	（RTC5R）	（地址=0000 0111）

表 13 实时时钟 RTCxR 数据寄存器

位	值	描述
7:0	读	返回 48 位 RTC 保持寄存器的当前值
	写	写到 RTC0R，把 RTC 的当前计数值传输到六个保持寄存器里，而同时 RTC 持续计数。

表 14 实时时钟控制寄存器（RTCCR 地址=01h）

位	值	描述
7:0	00h	对 RTC 计数器无作用，禁止字节递增功能，或取消 RTC 复位命令（除代码 80h 外）
	40h	用代码 80h 支持 RTC 一次复位，或用代码 0c0h 复位字节的递增功能
	80h	如果执行的是 40h 支持命令（arm command）,把 RTC 计数器的所有的六个字节复位到 0。
	c0h	把 RTC 计数器的所有六个字节复位到 0，并进入字节递增模式—用 40h 支持命令执行这条命令
7:6	01	这个位组合必须使用在每一个写到递增时钟寄存器的字节递增写操作上，这些写操作与位置“ 1 ”对应。例如：0100 1101 递增寄存器 0,2,3。字节递增模式必须使能。存储 00h 会取消字节递增模式。
5:0	0	对 RTC 计数器无作用
	1	RTC 计数器的相应字节递增。

7.6 看门狗定时器（Watchdog Timer）

看门狗定时器是一个 17 位计数器。普通工作时由 32kHz 时钟驱动。当看门狗定时器达到与 0.25 秒到 2 秒的延迟值相对应的几个值中的任意一个时，它“ 超时 ”。超时后，它从输出引脚发出一个 1 时钟周期长度的脉冲，通过内部电路复位处理器。为防止超时，程序必须在看门狗定时器超时之前把它“ 暂时断开（hit）”。断开的完成通过在 WDTCR 里存储一段代码执行。

表 15 看门狗定时器控制寄存器（WDTCR 地址=08h）

位	值	描述
7:0	5Ah	重启动（断开）看门狗定时器，超时值为 2 秒
	57h	重启动（断开）看门狗定时器，超时值 1 秒
	59h	重启动（断开）看门狗定时器，超时值 500ms。
	53h	重启动（断开）看门狗定时器，超时值 250ms。
	其他	对看门狗定时器无作用。

可以在 WDTTR 寄存器里存储一段特殊代码来禁止看门狗定时器。一般情况下不应该这么做，除非使用了一个外部看门狗器件。看门狗的功能是使处理器从软件崩溃或硬件扰动造成的死循环里跳转出来。

用写软件方法断开看门狗定时器（或关闭它）时，要非常小心。编程者在他的程序里不要到处都插入断开看门狗定时器的指令。如果程序崩溃并因此而禁止了由具有看门狗而具有的恢复能力（recovery ability），这些指令可能会变成某死循环的一部分。

下面是一种断开看门狗的建议方法。在 RAM 里建立了一个字节数组。每个字节都是一个虚拟看门狗。要断开一个虚拟看门狗，需要在某个字节里存储一个数值。每个虚拟看门狗递减计数，这个计数由一个周期性中断驱动的中断程序执行。这可能每 10ms 发生一次。如果所有虚拟看门狗都没有递减计数到 0，中断程序断开这个硬件看门狗。如果其中有任一个计到 0，中断程序禁止中断，然后进入一个死循环来等待复位。对虚拟看门狗的断开操作放在用户程序里的“必须执行”位置。

表 16 看门狗定时器测试寄存器（WDTTR 地址=09h）

位	值	描述
7:0	51h	用外设时钟为 WDT 定时器的最低有效字节提供时钟脉冲（仅限于芯片测试和代码 54h 以下）。
	52h	用外设时钟为 WDT 定时器的最高有效字节提供时钟脉冲（仅限于芯片测试和代码 54h 以下）。
	53h	用外设时钟并行地为 WDT 定时器的两个字节提供时钟脉冲（仅限于芯片测试和代码 54h 以下）。
	54h	禁止 WDT 定时器。这个值本身并不禁止定时器。当有一个两次写的操作序列时，即第一个写操作是 51h，52h 或 53h,而接着的写操作是 54h，这才实际地禁止 WDT 定时器。对这个寄存器的任何其他写操作都会重新使能 WDT 定时器。
	其他	WDT 的普通时钟情况（32kHz）。这是复位后的状态。

执行这个功能的代码也有可能以 0.25 秒周期断开看门狗，以加速复位。这些看门狗代码必须被写入，这样，软件崩溃把这些代码组合进来并在一个死循环里持续断开看门狗的可能性变的很小。下面的建议可提供一些帮助。

1．在看门狗断开程序入口点前放置一个跳转。这样就禁止了入口，避免直接调用或跳转到此程序。
2．调用这个程序之前，把一个数据字节设为一个特殊值，然后在程序里检验它，以保证调用请求来

自正确的调用者。如果不能保证，中断被禁止情况下就进入死循环。
3．保持数据破坏。如果这些出现错误，中断关闭下进入死循环。

7．7 系统复位

Rabbit 有一个主复位输入（/RESET），它初始化器件上的除 RTC 之外的所有东西。这个复位被延迟，直到正在进行的所有写周期完成，这样可以阻止任何潜在的内存破坏。如果没有正在进行的写周期，复位操作立即进行。

要等所有正在进行的写周期完成后才进行复位的目的是保护电池供电的内存变量避免在复位时被破坏。然而，如果负责电池切换的电源控制电路能阻塞 RAM 的片选信号，不管怎样的写周期都会被撤消。这实际并不一定很严重，因为可以使用软件方法保护电池供电内存里的关键变量。

即使复位时没有任何正在进行的写周期，复位过程最少需要 128 个主频时钟周期来完成。复位迫使处理器时钟和外设时钟处于八分频模式。注意，如果处理器由 32kHz 振荡器提供时钟，要使所有正在进行的写周期在复位过程之前完成和使时钟切换到八分频状态，128 个主频周期很可能不充分。

在复位时，所有内存控制信号保持无效。/RESET 信号成为无效（高）之后，处理器开始取指令，内存控制信号开始正常工作。注意，内存体控制寄存器的缺省值在每次访问时都选择四个等待状态，这样，初始的取程序的读内存操作长为 48 个时钟周期（8*（2+4））。

对内存控制信号的缺省选择由/CS0、/OE0 和/WE0 组成，而且允许写操作。可以软件编程改变这个选择使之与硬件配置相一致。一个典型初始化顺序是把时钟加到满速、选择合适数量的等待周期、选择合适的片选信号、输出允许信号和写允许信号。这时候，软件一般会检验系统状态，以判定刚才发生了什么类型的复位，并开始正常工作。

7．8 Rabbit 中断结构 (Rabbit Interrupt Structure)

中断执行一个调用，把 PC 推入堆栈，开始执行中断向量地址上的程序。中断向量地址对每个中断都具有固定的低字节。根据外部和内部中断的不同，高字节可通过设置寄存器 EIR 和 IIR 分别调整。并行端口 E 的某些特定引脚的电位跳变，能产生两个外部中断。

中断向量示于表 17。

这些中断与大多数 Z80 和 Z180 中断不同。在后者中，指向 EIR 和 IIR 的 256 字节的表，包含了开始中断程序的实际指令，而不是指向程序的 16 位指针。中断向量每 16 字节一组分离开，因此很小的中断程序的整个代码也适合这个表。

中断有 1、2、3 的优先级。处理器运行在优先级 0、1、2 或 3。如果申请了一个中断，它的优先级比处理器的高，中断将在下条指令后发生。这个中断自动把处理器的优先级提高到与它相同。原处理器优先级被推入包含了 4 位（4-position）优先级栈的 IP 寄存器。多个器件可能同时申请中断，请求中断的每个部件里的中断申请触发器被置位。如果这个中断被中断逻辑锁存之前那个触发器已被清除，中断请求丢失，不会有中断发生，这些示于表１8。表中表示的优先级仅适用于同优先级的中断，并且只在两个中断同时请求时有意义。大部分器件通过编程可选择优先级 1、2 或 3 上的中断。

表 17 外围设备地址和中断向量

芯片上的外设	I/O 地址范围	ISR 开始地址
系统管理（周期性中断）	0xh	{IIR,00h}
内存管理	1xh	无中断
从端口	2xh	{IIR,80h}
并行端口 A	3xh	无中断
并行端口 B	4xh	无中断
并行端口 C	5xh	无中断
并行端口 D	6xh	无中断
并行端口 E	7xh	无中断
外部 I/O 控制	8xh	无中断
定时器 A	Axh	{IIR,A0h}
定时器 B	Bxh	{IIR,B0h}
串行端口A	Cxh	{IIR,C0h}
串行端口B	Dxh	{IIR,D0h}
串行端口 C	Exh	{IIR,E0h}
串行端口 D	Fxh	{IIR,F0h}
RST 10 指令	n/a	{IIR,20h}
RST 18 指令	n/a	{IIR,30h}
RST 20 指令	n/a	{IIR,40h}
RST 28 指令	n/a	{IIR,50h}
RST 38 指令	n/a	{IIR,70h}

表 18 中断—优先级和清除请求的操作

优先级	*中断源*	*清除中断所需的操作*
最高	外部 1	中断应答时自动进行
	外部 0	中断应答时自动进行
	周期性（2kHz）	读 GCSR
	定时器 B	读 TBCSR[1]
	定时器A	读 TASR
	从端口	写 SPSR
	串行端口 A	RX：读 SADR 或 SAAR TX：写 SADR 或 SASR
	串行端口 B	RX：读 SBDR 或 SBAR TX：写 SBDR，SBAR 或 SBSR

（续上表）

	串行端口 C	RX：读 SCDR 或 SCAR TX：写 SCDR，SCAR 或 SCSR
最低	串行端口 D	RX：读 SDDR 或 SDAR TX：写 SDDR，SDAR 或 SDSR

[1] 如果在 ISR 里面没有进行比较寄存器（TBMxR 和 TBLxR）操作，中断只会触发一次。

外部中断时，清除中断请求的的唯一动作是中断响应，它会自动清除请求。必须在中断服务程序里为其他中断采取特别措施。

7．8．1 外部中断

有两个外部中断请求。由于 Rabbit 设计的问题，只有一个中断可用于通用用途。为了在有设计问题情况下工作，如技术注解 301“ Rabbit2000 微处理器中断问题 ”中所述，一般需要用一个 1 千欧姆的电阻把一个外部中断请求线连接到两个中断上，如图 20 示。这么做之后，两个中断（#1 和#0）在上升和下降沿都会产生中断，但中断#1 被忽略，中断#0 则进入中断服务程序。这样可以防止中断丢失或是虚假中断。为防止虚假中断，控制寄存器里的优先级设置为与竞争中断所使用的最高优先级相等，进入服务程序时降低优先级。

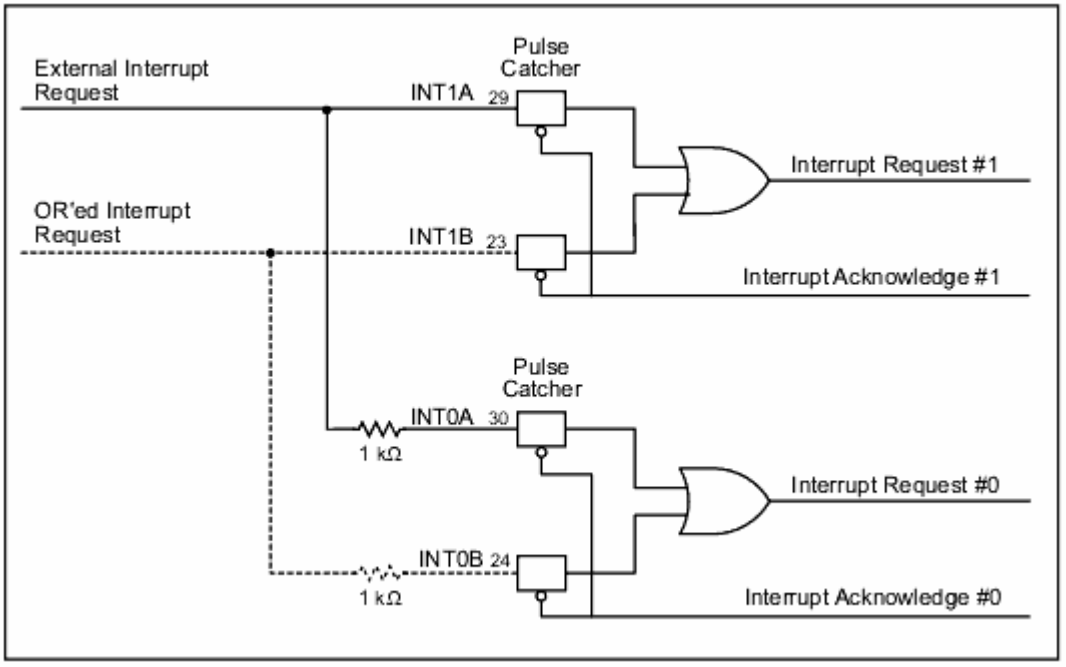


图 20 外部中断线逻辑图

外部中断在输入跳变时发生，输入跳变可编程为上升，下降或两者都可以。脉冲捕捉器是可独立编程的，用来探测输入端的上升，下降沿或任一沿。连接到相同中断的脉冲捕捉器对应该编程为对同一类型的脉冲沿的探测。每个中断引脚有自己的捕捉器件来捕获脉冲沿跳变，并发出中断请求。

中断响应时，与那个中断相关的两个脉冲捕捉器自动复位。因为触发脉冲沿几乎同时发生或者因为中断被处理器优先级所约束，相应的中断发生之前两种脉冲沿都检测到了，那么检测到的两个沿将只有

一个中断响应。如果跳变动作不是太快的话，中断服务程序可通过并行口 E 读取中断引脚，并判断哪些线有过跳变。还可以通过把端口 E 的匹配位配置为一个输出和触发位来产生中断。

表 19 外部中断的控制寄存器

寄存器名	寄存器地址	位 7,6	位 5,4	位 3,2	位 1,0
I0CR	1001 1000	xx	INT0B PE4	INT0A PE0	Enb INT0
I1CR	1001 1001	xx	INT1B PE5	INT1A PE1	Enb INT1
			脉冲沿触发 00-禁止 10-上升 01-下降 11-两者	脉冲沿触发 00-禁止 10-上升 01-下降 11-两者	中断 00-禁止 01-优先级 1 10-优先级 2 11-优先级 3

中断向量:INT0—EIR,00h/INT1—EIR,08h

当希望为附加的外设扩展中断数目时，使用者应该利用中断程序把中断分派给其他虚拟中断程序。每个附加中断设备都应该发信号给处理器，表明它在申请中断。每个设备需要一条独立的信号线，这样处理器可以判断是哪个设备在申请中断。

下面的代码表示了中断服务程序应该怎么写：

```
; External interrupt Routine #1
int1:
      ipres      ; 恢复中断优先级
      ret        ; 返回，忽略中断
;

; 外部中断程序#0（优先级可编程为 3）
int2:
      push ip   ; 保存中断优先级
      ipset 1   ; 设置所需中断优先级 (1, 2 等等)
; 中断程序体插入此处
;
      pop ip    ; 重新获得入口优先级
      ipres     ; 恢复被中断程序的级别
      ret       ; 从中断返回
```

7．9 引导操作（Bootstrap Operation）

设备提供三个引导源：来自从端口，来自时钟同步串行模式的串行端口 A，来自于异步模式的串行端口 A。这由复位后的 SMODE 引脚的状态来控制。如果（SMODE1，SMODE0）=（0，0），引导操作被禁止。

引导操作禁止从内存的普通取代码动作，而用一个小型启动 ROM 的输出代替取程序。这个引导程序

从选中的外设三个字节一组的读取。第一个字节 是一个 16 位地址的最高有效字节，接着是 16 位地址的最低有效字节，然后是一个数据字节。引导程序把数据字节写到下载的地址，并跳回引导程序的开头。用地址的最高有效位来判断数据字节的目的地。如果这位是 0，数据写到下载地址指示的内存位置。注意所有的内存控制信号在引导期间继续正常工作。

引导程序的执行自动等待来自选中外设的数据变为可用，每个传输的字节自动复位看门狗定时器。然而，看门狗定时器依然工作，所以字节传输必须足够频繁，避免看门狗定时器超时。

SMODE 引脚设为 0 时，引导操作终止。SMODE 就紧接于取引导程序的第一条指令之前被采样。如果它是 0，指令从常规内存取，开始地址是 0000h。从端口控制寄存器允许引导操作被远程终止。向这个寄存器的位 7 写“ 1 ”，可使引导终止。所以指令序列 80h,24h 和 80h 会终止引导。

引导并不限于仅在复位后立即进行，因为启动 ROM 的地址仅由地址的四位最低有效位决定。所以任何时候，如果地址以四个 0 结束，而且 SMODE 引脚非 0、SPCR 的位 7 是 0，引导程序就开始执行。这就允许了来自选中引导端口的内嵌下载。一旦引导完成（或者通过使 SMODE 返变 0，或者把 SPCR 位置位），将继续从响应引导断点处执行程序。

（SMODE1，SMODE0）=（0，1）时，选择从端口引导。这种情况下，并行端口 A 的引脚作为一条字节宽度的数据总线，并行端口 B 和 E 的选中引脚作为从端口控制信号。只有从端口数据寄存器 0 用作引导，对其他数据寄存器的写操作被忽略，并可通过屏蔽空写（Write Empty）信号来实际地干涉引导。

注意：参看 14.8.2 小节的使用引导模式的例子
SMODE=10 时，选择时钟同步的串行端口 A 引导。这时，并行端口 C 的位 7 作为串行数据口，并行端口 B 的位 1 作为串行时钟。注意引导的串行时钟必须由外部提供。这样排除了使用串行 EEPROM 用于引导。

SMODE=11 时，选择异步模式串行端口 A 引导。这时，并行端口的位 7 作为串行数据口，使用 32kHz 振荡器提供串行时钟。使用一个专用分频电路，使 32kHz 信号能为 2400bps 异步传输提供时间基准。只有 2400bps 支持引导，串行数据必须是 8 位，以便操作正确。

用串行端口 A 执行引导，不需要 TXA 信号，因为引导是单方向通信。复位结束且引导开始时，TXA 为低电位，表明被复位清零后它的功能是一个并行端口输出位。这有可能被一些串行通信器件看作断点信号（break signal）。通过发送三个一组的 80h,50h,40h ，TXA 可强加为高。此序列在并行端口 C 里存储 40h。一种备用方法是发送三个一组的 80h,55h,40h，这些指令通过写并行端口 C 的功能寄存器（55h），经由并行端口 C 的位 6 使能 TXA 输出。

引导时的传输率不能太快，要保证处理器能执行这些指令流。用从端口引导时，空写信号作为联锁，因为直到空写信号有效之前下一个字节都不准写到从端口。时钟同步串口引导和异步串口引导时，不存在这种联锁。这两种情况下，记住处理器始终以八分频模式开始工作并有四个等待状态，并因此而限制了传输率。2400bps 异步模式下，发送每个字符占用 4ms，所以，除非系统时钟非常低，不大可能出现故障。

8.Rabbit 内存映射和接口（Rabbit Memory Mapping and Interface）

参看 3.2 小节的 Rabbit 内存映射指导讨论。

图 21 表示 Rabbit 内存映射的概观。内存映射单元的任务是接收 16 位地址并把它转换成 20 位地址。内存接口单元接收 20 位地址并产生直接应用于内存芯片的控制信号。

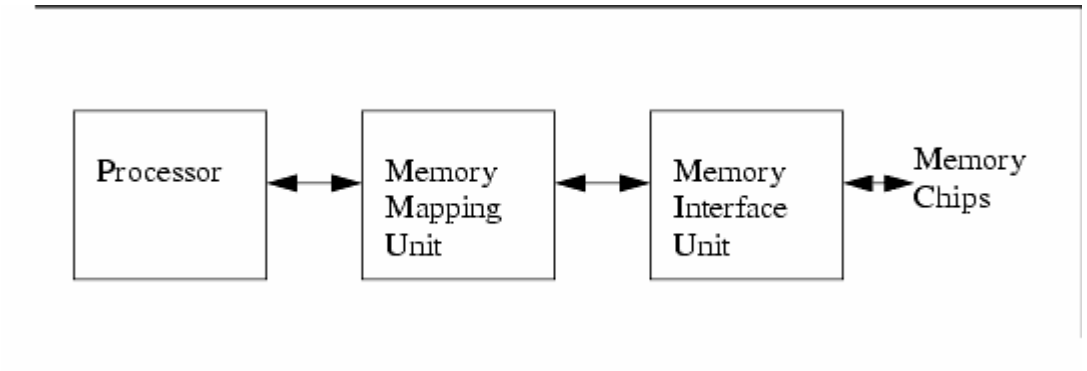


图 21 Rabbit 内存映射概观

８．１　内存映射单元（Memory-Mapping Unit）

处理器指令可访问的 64K 的 16 位地址空间被分段。每段长度是 4K 的倍数。除了扩展代码段，其他段大小可调整，有些段可减小到 0，因而从内存映像中消失。

在图 22 的实例中表示了四个段。段长度寄存器（SEGSIZE）决定图中标记的边界。扩展代码段一般占用地址 0E000h—0FFFFh。堆栈段从 SEGSIZE 寄存器的高 4 位指明的地址到 0DFFFh。例如，如果 SEGSIZE 的高 4 位是 0Dh，堆栈段占据 0D000h—0DFFFh，或者说 4K 空间。如果 SEGSIZE 的高 4 位大于或等于 0Eh，堆栈段消失。如果这 4 位都设为 0，在堆栈段下面的两个段将消失。

SEGSIZE 的低 4 位决定图中所示的低边界（数据段和根段之间）。如果它与高边界（数据段和堆栈段之间）相等或者大于 0Eh，数据段消失。如果数据段放在 0 位置，代码段（根段）消失。

内存管理单元接收一个来自处理器的 16 位地址，把它转换成一个 20 位地址。这个工作的流程如下。

1．检查地址的高 4 位，决定这个 16 位地址属于哪个段。所有地址必须属于可能的 4 个段中的一个。
2．每个段有一个 8 位段寄存器。这个 8 位寄存器被加到这个 16 位地址的高 4 位，创建一个 20 位地址。如果这种加操作导致一个不适合 20 位的地址，将产生回绕(wraparound)

表 20 段寄存器

段寄存器	功能
XPC	指出扩展代码段在内存里的位置。由处理器指令：ld a, xpc,ld xpc,a,lcall,lret,ljp 读写。
STACKSEG= 11h	指出堆栈段在内存里的位置
DATASEG= 12h	指出数据段在内存里的位置

表 21 段长度寄存器

	位 7..4	位 3..0
SEGSIZE=13h	堆栈段的边界地址	数据段的边界地址

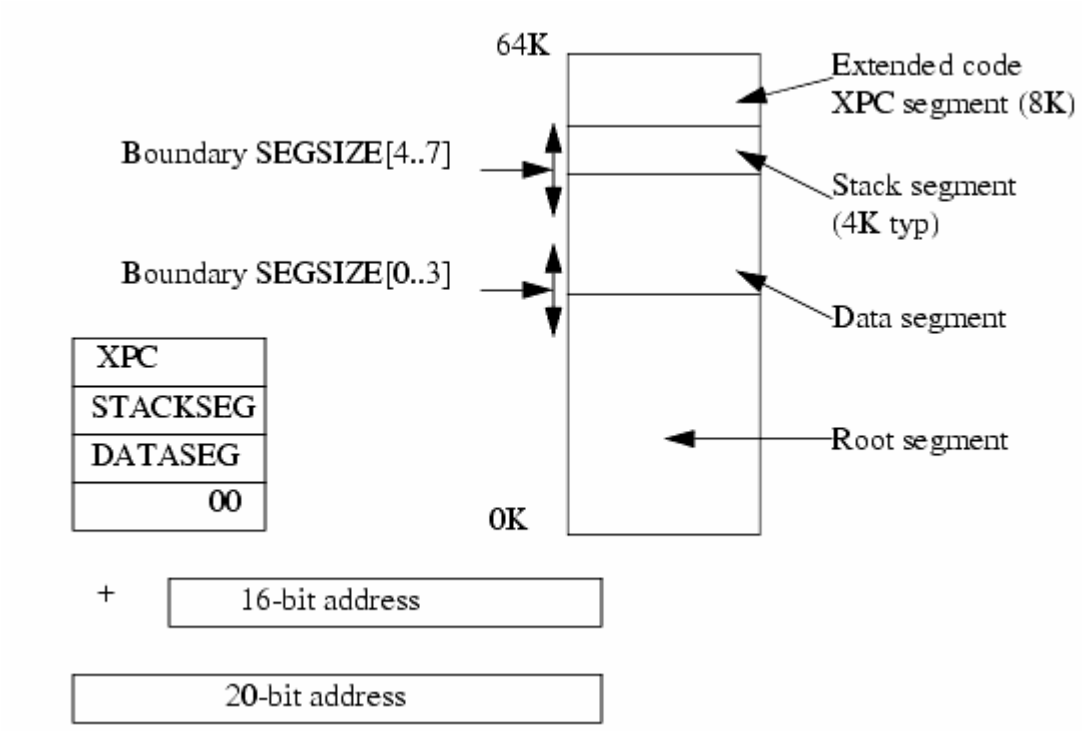


图 22 内存段

８．２　内存接口单元（Memory Interface Unit）

内存映射单元生成的 20 位内存地址装入内存接口单元。内存接口单元对 1M 物理内存的每 256K 象限都有一个独立的只写控制寄存器（参看表 22）。这个控制寄存器表明对那个象限的内存存取请求怎样分派到 Rabbit 所连接的内存芯片。有三个分离的片选输出线（/CS0，/CS1，/CS2）可用来选择三个不同内存芯片中的一个。控制寄存器里的一个域决定内存存取要选择哪个片选线。同样的一个片选线可以在多于一个象限里被访问。例如，如果有 512K 的 RAM 并选择了/CS1，用/CS1 访问第三和第四象限比较合适，从而把此 RAM 芯片映射到地址 80000h 到 0FFFFFh。

表 22 内存体控制寄存器 x（MBxCR=14h+x）

位 7,6	位 5	位 4	位 3	位 2	位 1,0
00—4 个等待状态 01—2 个等待状态 10—1 个等待状态 11—无等待状态	1—反转地址 A19	1—反转地址 A18	1—对这个象限实行内存写保护	0—使用/OE0，/WE0 1—使用/OE1，/WE1	00—使用/CS0 01—使用/CS1 1x—使用/CS2

８．３　内存体控制寄存器的功能

表 22 描述了四个内存体控制寄存器的操作。它们是只写的。每个寄存器控制 1M 地址空间里的一个象限（共四个象限）。

・位 7,6---访问此象限时使用的等待状态数目。无等待状态时，读操作需要 2 个时钟脉冲而写需要 3 个。等待状态附加到这些数目上。

・位 5,4---这几位允许高位地址线反转。取反发生在选择存储体寄存器的逻辑操作之后，所以设置这些线对使用哪个寄存器没有影响。这种取反可以用来在通常分配给 256K 芯片的地址空间里安装一个 1M 的内存芯片，大于 256K 的内存可以作为 4 个 256K 页之一来访问。这对内存体控制寄存器所控制的象限之外的空间无影响。

・位 3---禁止写脉冲访问这个象限里的内存。在保护闪存不受非有意的写脉冲影响时有用，虽然这个写脉冲并不能实际写闪存，因为闪存由锁定代码保护着，但它会暂时禁止闪存，而且如果这段闪存用于存放代码，它可能会使系统崩溃。

・位 2---选择驱动/OEx 和/WEx 哪种设置以访问这个象限的内存。

・位 1，0---决定驱动三根芯片选择线中的哪一根以访问这个象限的内存。

8．3．1 利用分段的可选择的 A16，A19 反转（/CS1 使能）

读/写 MMIDR 寄存器所控制的 A19 或 A16 的反转，在访问根段和数据段时，可用来重定向根段和数据段的映射。方式是反转某些特定位。当前还没有这个功能的规划使用。

可选择的/CS1 的使能，对那些正努力增加电池供电 RAM 使用时间的系统有价值。通过使能/CS1，可绕过断电时迫使/CS1 为高的开关延迟时间。这个特征增加了功耗，因为 RAM 一直使能，而且它的访问一般由/OE1 控制。

表 23 MMU 指令/数据寄存器（MMIDR=010h）

位 7，6，5	位 4	位 3	位 2	位 1	位 0
000	1—强制/CS1 一直使能	这几位必须是 0			

８．４　扩展代码和数据的地址分配（Allocation of Extended Code and Data）

Dynamic　C编译器把代码编译到根代码空间或扩展代码空间。根代码从低位内存开始向上编译。

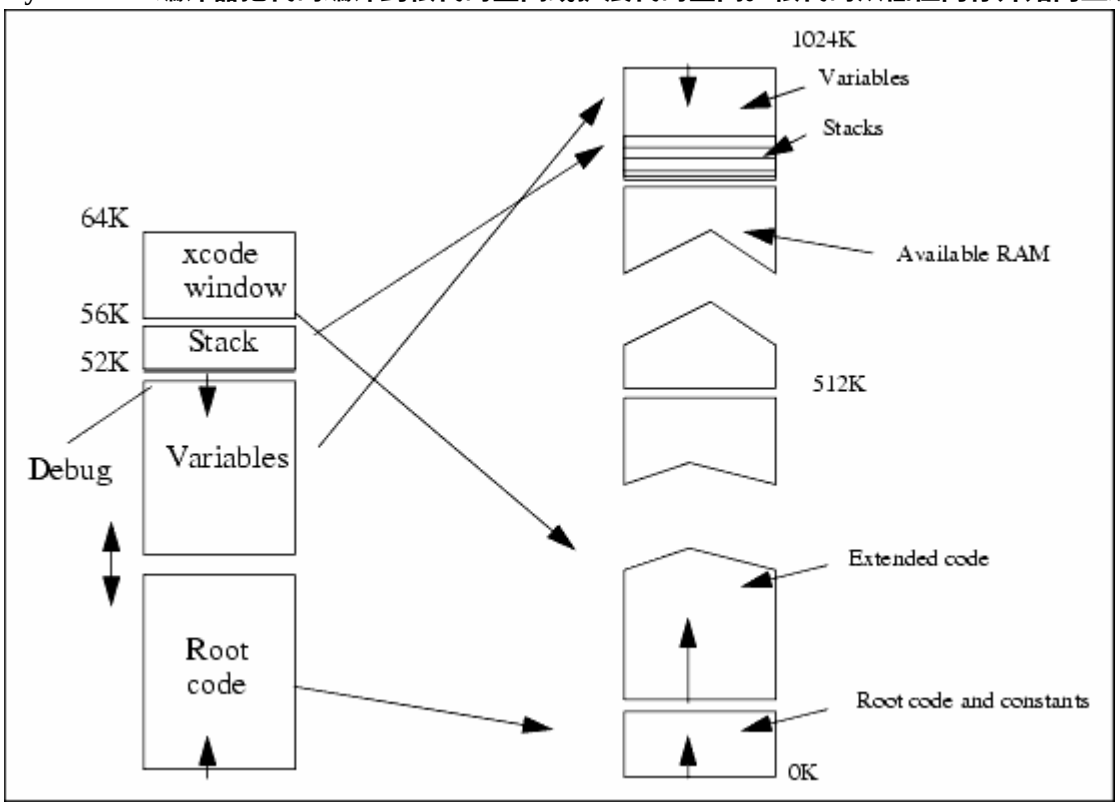


图 23　典型内存映射和内存使用

扩展代码从根代码和数据上方开始分配。分配一般持续到闪存的结束。

数据变量分配到内存中以向后方式工作的 RAM 里。分配通常在 64K D 空间里的 52K 上开始并继续下去。这 52K 空间从 0 向上分配，必须与根代码和数据共享。

Dynamic C 还支持扩展数据常量。这些常量与闪存里的扩展代码混合。

８．５　编译器如何进行到内存的编译（How the Compiler Compiles to Memory）

编译器为根代码、常量、扩展代码和扩展常量生成实际代码。它为数据变量分配空间，但并不产生要存储到内存里的数据位。

除了最小的程序，大部分程序的代码编译到扩展内存。这些代码在 8K（E000—FFFF）窗口里执行。这个 8K 窗口使用页访问方式。使用 16 位编址的指令可以在页内跳转，也可以跳出本页到 64K 空间的剩余部分。特殊指令，尤其长调用，长跳转和长返回，被用来访问 8K 窗口外的代码。当这些控制指令的某个转移执行时，地址和整个 8K 窗口或页面的景象都改变。这允许在 1M 空间里到任何指令

的转移。8 位的 XPC 寄存器控制 8K 窗口用 256 张 4K 页的哪一页定位。16 位 PC 控制指令的地址，通常的范围是 E000—FFFF。分页访问的优点是大部分指令继续使用 16 位编址。只在进行一个超出范围的转移控制时，才需要进行一个 20 位的转移控制。当一页是 8K 时，用 4K 的最小步幅做页定位的好处，是可以无间隔（间隔由页变化产生）的持续编译代码。当页移动 4K 时，只要在变动页定位基准之前已经移过了页的中点，前代码的结尾在窗口里就仍可见。

当编译器在扩展代码窗口里编译代码时，它适时的查看代码是否已经越过了窗口的中点，或说 F000。如果代码越过了 F000，编译器把窗口向下滑动 4K，这样 F000+x 的代码就会在 E000+x 上。这导致代码分成 4K 典型段，但每段也可以很短或长达 8K。在每个段内的转移控制用 16 位编址就可达到，段间则需要 20 位编制。

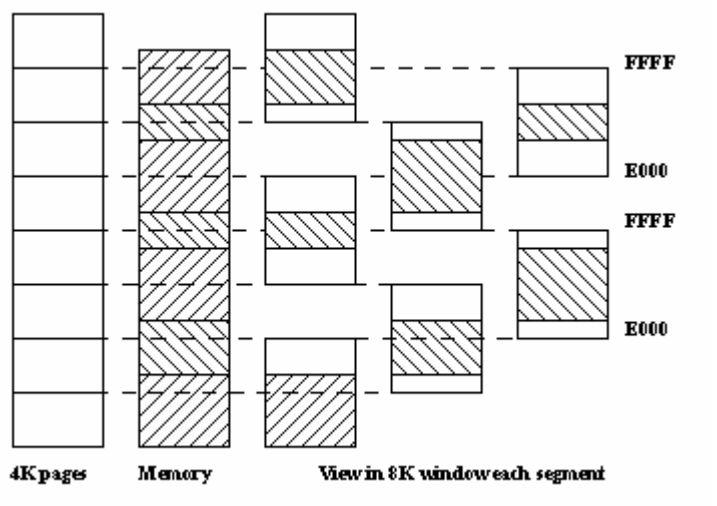


图 24　在扩展内存里编译代码段

9．并行端口（Parallel Ports）

Rabbit 有指定为 A、B、C、D 和 E 的五个八位并行端口。并行端口还与其他很多功能共享引脚，如表 4 所示。这些引脚的重要特性概括如下：

・端口 A—与从端口数据接口共享

・端口 B—与从端口的控制线和串行端口 A 及 B 的定时串行模式的时钟 I/O 共享

・端口 C—与串行端口的串行数据 I/O 共享

・端口 D—其中 4 位与串行端口 A 和 B 的备用 I/O 引脚共享；另 4 位不共享。端口 D 具有把它的输出配置成漏极输出的能力。端口 D 有输出预加载寄存器，它的内容可以在定时器控制下由时钟驱动，以生成脉冲。端口 D 的 0-3 位的电流驱动能力较强。

・端口 E—它的所有位可以配置为 I/O 选通信号。其中的 4 位可作为外部中断输入。端口 E 的有一位与从端口的片选脚共用。端口 E 有输出预加载寄存器，它的内容可以在定时器控制下由时钟驱动进入输出寄存器里，以生成脉冲。

9．1 并行端口 A

并行端口 A 有一个单独的读/写寄存器，如表 24 所示

表 24 并行端口 A 数据寄存器 PADR（地址=030h）

R/W 8位数据值

从端口使能时不可以使用这个寄存器。

从端口控制寄存器用来控制并行端口 A 是输出还是输入。要使端口为输入，在 SPCR（从端口控制寄存器）里存入 080h。要使端口为输出，在 SPCR 里存入 084h。并行端口 A 在复位时配置为输入。

读此端口后，所读值反映了引脚的电平，“1”高而“0”低。如果引脚被外部电压强制为一个不同的状态，所读值会与存放在输出寄存器里的值不同。

9．2 并行端口 B

表 25 表示并行端口 B，专用作并行端口时它有六个输入和两个输出。

表 25 并行端口 B 数据寄存器 PBDR（地址=040h）

	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
读	反射驱动 (echodrive)	反射驱动	PB5 in	PB4 in	PB3 in	PB2 in	PB1 in	PB0 in
写	PB7	PB6	x	x	x	x	x	x

从端口使能时，并行端口线 PB2-PB7 分配给各种从端口功能。然而，这时仍旧可能利用端口 B 数据寄存器来读 PB0-PB5。同时还可以读驱动 PB6 和 PB7 的信号（这个信号在来自从端口逻辑的信号线上）。

不管从端口是否使能，PB0 反映引脚的输入，除非串行端口 B 使它的内部时钟使能——这会导致这根线由串行口时钟驱动。PB1 反映引脚的输入，除非串行口 A 使能它的内部时钟。

复位时，输出的位 6 和位 7 被复位，输出到 PB6 和 PB7（封装引脚是 99，100）上的值将也是低。

9．3 并行端口 C

表 26 所示的并行端口 C，有 4 个输入和 4 个输出。偶数号端口，PC0、PC2、PC4 和 PC6 是输出。奇数号端口 PC1、PC3、PC5 和 PC7 是输入。当读它的数据寄存器时，位 1、3、5、7 返回的是引脚上的值，位 0、2、4、6 则返回输出缓冲器的驱动信号。输出缓冲器的驱动信号和输出引脚的值一般相同。或由端口 C 的数据寄存器驱动这些引脚，或由串行端口传送线（transmit lines）驱动它们。由 PCFR（并行端口 C 功能寄存器）里设置的位识别是数据寄存器还是串行端口传送线驱动着引脚。

并行端口 C 与四个串行端口共享其引脚。并行端口的输入脚也可以作为串行口输入使用（串行端口 A 和 B 可以交替使用位 7 和 5 在端口 D 里作为输入，而且这些串行端口的输入源决定于相应的串行口控制寄存器的配置）。当作为串行输入使用时，数据线仍可从并行口 C 的数据寄存器读取。通过在端口 C 功能寄存器（PCFR）相应位置的存入位值，可选择并行端口输出为串行端口输出。当一个并行端口的输出选择为串行端口输出，存储在数据寄存器里的数值被忽略。复位时，有效（偶数位）的功能寄存器位和数据寄存器的位都设成 0。这导致端口在四个输出位上都输出 0。

表 26 并行端口 C 数据寄存器和功能寄存器

	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
PCDR(r) adr=050h	PC7 in	反射驱动	PC5 In	反射驱动	PC3 In	反射驱动	PC1 in	反射驱动
PCDR(w) adr=050h	x	PC6	X	PC4	X	PC2	x	PC0
PCFR(w) adr=055h	x	驱动 TXA	X	驱动 TXB	X	驱动 TXC	x	驱动 TXD

9．4 并行端口 D

表 25 所示的并行端口 D 的 8 个引脚可独立的编程为输入和输出。编程为输出时，引脚可独立的被选择为漏极输出或标准输出。如果需要，端口 D 引脚可逐位编址。输出寄存器级联并由定时器控制，能够生成精确定时脉冲。此外，端口 D（和 E）输出具有较强的驱动能力。端口 D 位 4 和 5 可用作串行口 B 的备用引脚，位 6 和 7 可用作串行口 A 的备用引脚。备用串行端口位的分配，使同一串行端口能够使串行口在不同时间工作在不同通信线上。

复位时，数据方向寄存器置 0，使所有引脚作为输入。此外，控制寄存器的位（位 0、1、4、5）置 0，以保证装载时数据由时钟驱动进入了输出寄存器。所有其它与端口 D 相关的寄存器复位时不初始化。

下面的寄存器在表 27 和 28 里描述。

·PDDR—并行端口 D 数据寄存器。读/写。
·PDDDR—并行端口 D 数据方向寄存器。位值为“1”表明相应引脚是输出。只写。
·PDDCR—并行端口 D 驱动控制寄存器。位值为“1”表明相应引脚是漏极开路输出（如果这个引脚配置为输出）。只写。
·PDFT—并行端口 D 功能控制寄存器。这个端口可用来使端口的位置 4 和 6 为串行输出。只写。
·PDBxR—这 8 个寄存器可用于设置各自端口位置上的输出。
·PDCR—并行端口控制寄存器。这个寄存器用于控制端口的最终输出寄存器的高和低半字节的计时。复位时，位 0、1、4、5 清 0。

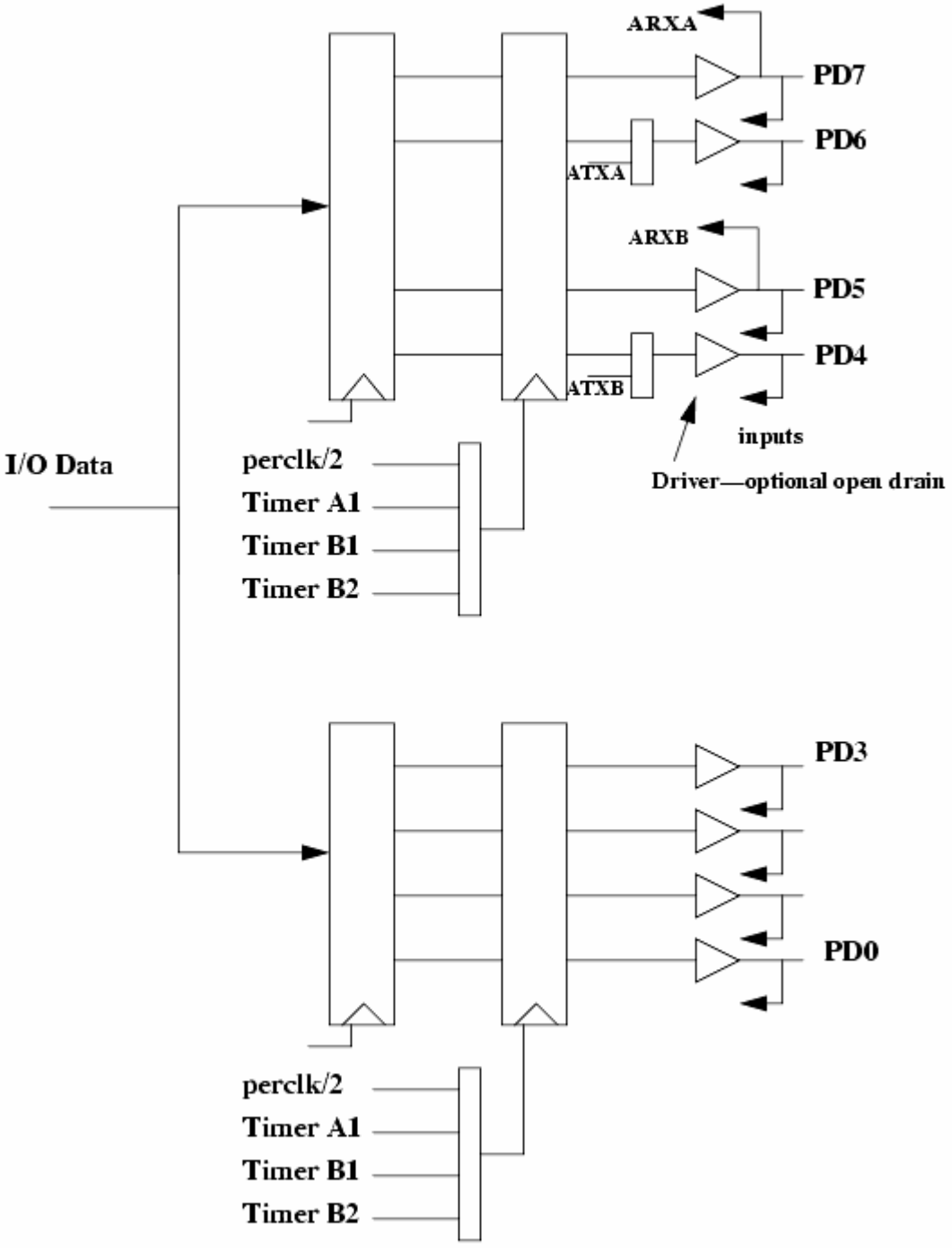

ARXA
PD7
PD6
ATXA
ARXB
PD5
PD4
ATXB
inputs
Driver—optional open drain
I/O Data
perclk/2
Timer A1
Timer B1
Timer B2
PD3
PD0
perclk/2
Timer A1
Timer B1
Timer B2

图 25 并行端口 D 框图

表 27 并行端口 D 的寄存器

	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
PDDR (R/W) adr= 060h	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
PDDCR (W) adr=066h	out= 漏极开路	out= 漏极开路	out= 漏极开路	out= 漏极开路	out= 漏极开路	out= 漏极开路	out= 漏极开路	out= 漏极开路
PDFR (W) adr=065h	x	alt(备用) TXA	x	alt(备用) TXB	x	x	x	x
PDDDR (W) adr=067h	dir (方向)= out	dir= out	dir= out	dir= out	dir= out	dir= out	dir= out	dir= out
PDB0R(W) adr=068h	x	x	x	X	x	x	x	PD0
PDB1R(W) adr=069h	x	x	x	X	x	x	PD1	x
PDB2R(W) adr=06Ah	x	x	x	x	x	PD2	x	x
PDB3R(W) adr=06Bh	x	x	x	x	PD3	x	x	x
PDB4R(W) adr=06Ch	x	x	x	PD4	x	x	x	x
PDB5R(W) adr=06Dh	x	x	PD5	x	x	x	x	x
PDB6R(W) adr=06Eh	x	PD6	x	x	x	x	x	x
PDB7R(W) adr=06Fh	PD7	x	x	x	x	x	x	x

表 28 并行端口 D 控制寄存器（adr=064h）

位 7,6	位 5,4	位 3,2	位 1,0
x	00—计时上半字节(时钟 clk/2) 01—计时上半字节（时钟为定时器 A1） 10—计时（定时器 B1） 11—计时（定时器 B2）	x	00—计时下半字节(时钟 clk/2) 01--计时下半字节（时钟为定时器 A1） 10—计时（定时器 B1） 11—计时（定时器 B2）

9．5 并行端口 E

图 26 所示的并行端口 E 有 8 个 I/O 引脚，都可独立编程为输入或输出。端口 E 比大部分其他端口有更强的驱动能力。从端口使能时，PE7 作为从端口片选信号。每个端口 E 的输出可配置为一个 I/O 选通信号。此外，端口 E 中的 4 根线可用作中断请求输入。输出寄存器级联，由定时器控制，使产生精确定时脉冲。

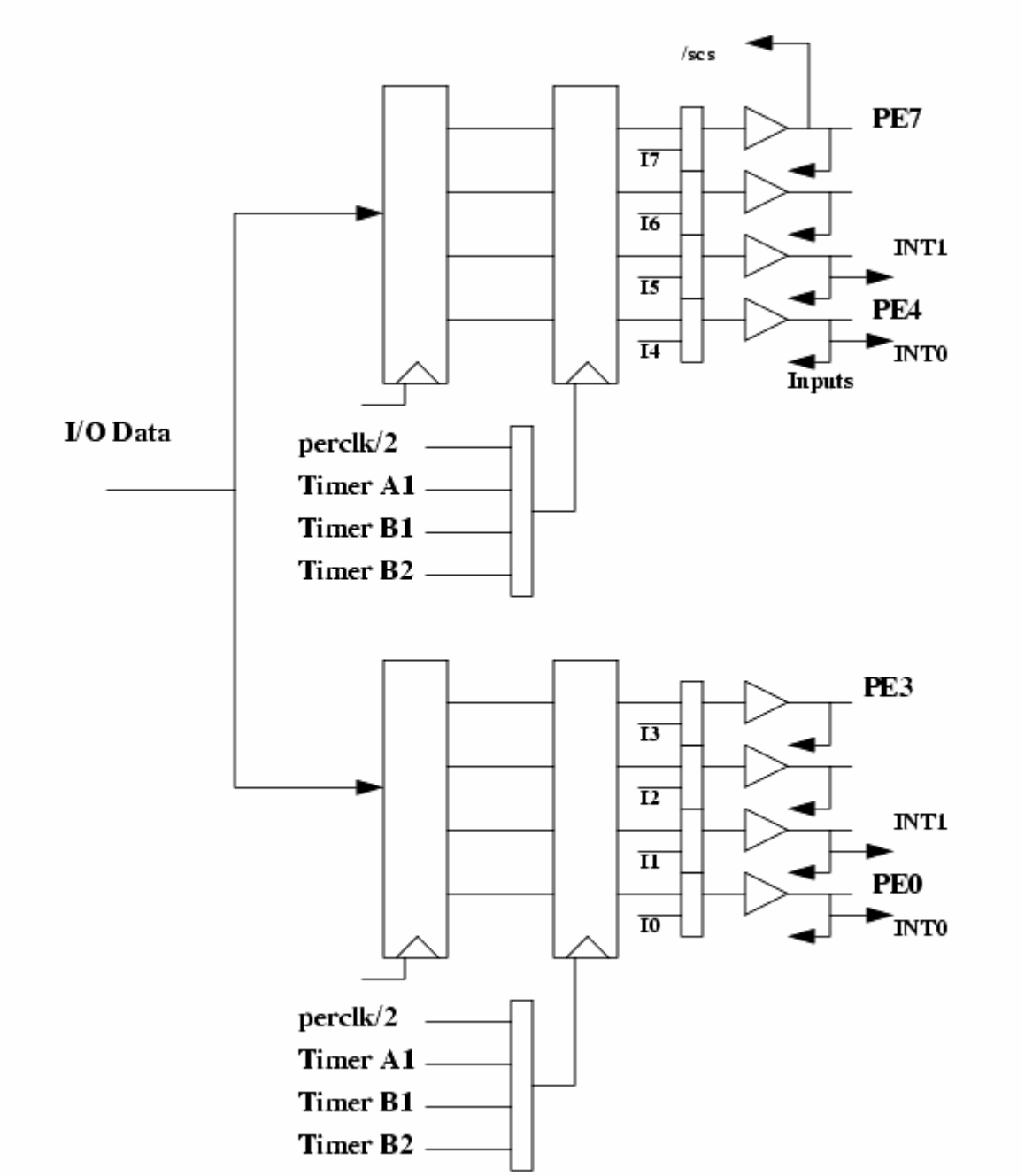


图 26 并行端口 E 框图

下面的寄存器在表 29 和 30 里描述。

· PEDR—端口 E 数据寄存器。读引脚的值。写端口 E 的预加载寄存器。

· PEDDR—端口 E 方向寄存器。值为“1”，表示相应引脚是输出。复位时此寄存器清 0。

·PEFR—端口 E 功能寄存器。值为“1”，表相应输出是一个 I/O 选通信号。I/O 体控制寄存器（IBxCR）控制着 I/O 选通信号。要是选通信号工作，数据方向必须设为输出。

· PEBxR—这些寄存器可设置单独输出位为开或关。

· PECR—端口 E 控制寄存器。这个寄存器用来控制端口的最终输出寄存器的高半和低半字节的同步计时。复位时，位 0、1、4、5 置 0。

表 29 并行端口 E 的寄存器

	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
PEDR(R/W) adr=070h	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
PEFR(W) adr=075h	alt /I7	alt /I6	alt /I5	alt /I4	alt /I3	alt /I2	alt /I1	alt /I0
PEDDR(W) adr=077h	dir= out	dir= out	dir= out	dir= out	dir= out	dir= out	dir= out	dir= out
PEB0R(W) adr=078h	x	x	x	X	x	x	x	PE0
PEB1R(W) adr=079h	x	x	x	X	x	x	PE1	x
PEB2R(W) adr=07Ah	x	x	x	X	x	PE2	x	x
PEB3R(W) adr=07Bh	x	x	x	X	PE3	x	x	x
PEB4R(W) adr=07Ch	x	x	x	PE4	x	x	x	x
PEB5R(W) adr=07Dh	x	x	PE5	X	x	x	x	x
PEB6R(W) adr=07Eh	x	PE6	x	X	x	x	x	x
PEB7R(W) adr=07Fh	PE7	x	x	X	x	x	x	X

表 30 并行端口 E 控制寄存器（adr=074h）

位 7,6	位 5,4	位 3,2	位 1,0
x	00—计时上半字节(时钟 clk/2) 02—计时上半字节（时钟为定时器 A1） 10—计时（定时器 B1） 11—计时（定时器 B2）	x	00—计时下半字节(时钟 clk/2) 03—01—计时下半字节（时钟为定时器 A1） 10—计时（定时器 B1） 11—计时（定时器 B2）

10．I/O 体控制寄存器（I/O Bank Control Registers）

端口 E 的引脚可独立设为 I/O 选通信号。8 个可用的 I/O 选通的都有一个控制寄存器，控制选通的性质和要插入到 I/O 总线周期里的等待状态的数目。可为任何选通信号取消写操作。图 27 示出选通的类型。8 个 I/O 选通中的每一个有效时，占用 64K 外部 I/O 地址空间的 1/8。

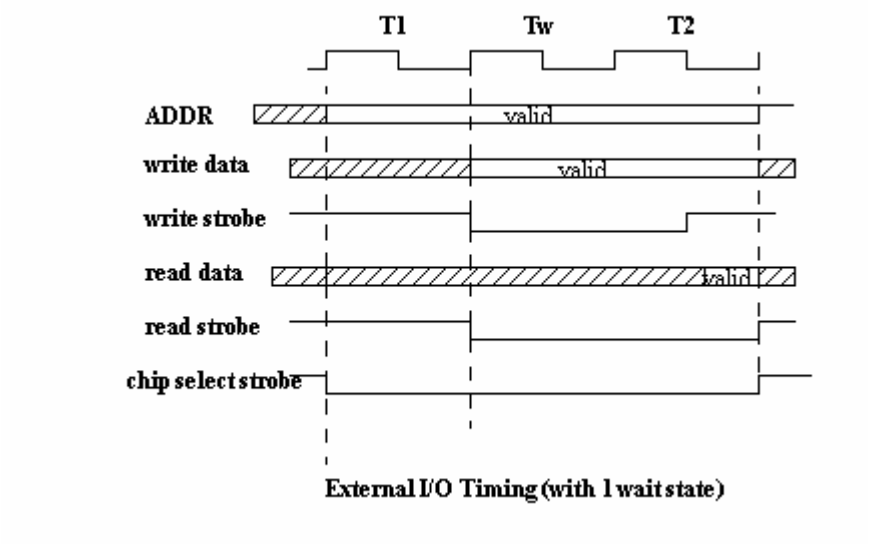


图 27 外部 I/O 总线周期

表 31 表明这 8 个 I/O 体控制寄存器如何组织。

表 31 I/O 体控制寄存器（adr IbxCR=08xh）

位 7,6	位 5,4	位 3	位 2-0
等待状态码 11-1 10-3 01-7 00-15	/IX 选通类型 00—片选 01—读选通 10—写选通 11—读和写选通的“或”	1—允许写 0—禁止写	忽略

这 8 个 I/O 体控制寄存器决定应用于外部 I/O 访问的 I/O 等待状态数目，在每个寄存器控制的范围内发生作用，即使相关选通没有使能。

产生等待状态的控制，独立于端口 E 里的相关选通是否使能。每个寄存器的高 2 位决定等待状态数目。有 1、3、7、15 个等待状态四种选择。至少有一个外部 I/O 等待状态，因此最小外部 I/O 读周期是 3 个时钟周期长。写禁止的功能适用于端口 E 写选通和/IOWR 信号。，

这些控制位对内部 I/O 空间无作用，它没有与读或写访问关联的等待状态。内部 I/O 读或写周期是两个时钟周期长。

I/O 选通极大的简化了内部器件的接口。复位时，每个寄存器的高 5 位清零。并行端口 E 不会输出这些信号，除非数据方向寄存器的位的设置与期望的输出位置相符。此外，端口 E 功能寄存器必须为每个相应位置设成“1”。

每个 I/O 体由 16-位 I/O 地址的三个最高有效位选择。表 32 表示 I/O 控制寄存器和它在 64K 地址空间

里相应空间的关系。

表 32　外部 I/O 寄存器地址范围和引脚映射

控制寄存器	端口 E 引脚	I/O 地址 A[15：13]	I/O 地址范围
IB0CR	PE0	000	0x0000—0x1FFF
IB1CR	PE1	001	0x2000—0x3FFF
IB2CR	PE2	010	0x4000—0x5FFF
IB3CR	PE3	011	0x6000—0x7FFF
IB4CR	PE4	100	0x8000—0x9FFF
IB5CR	PE5	101	0xA000—0xBFFF
IB6CR	PE6	110	0xC000—0xDFFF
IB7CR	PE7	111	0xE000—0xFFFF

注意：如果一条 I/O 指令（头标 IOI 或 IOE）后面跟随了使用 HL 作为索引寄存器的 12 种单字节操作码之一，参考 3.3.8 小节来修正 bug。

11．定时器（Timers）

有两个定时器，定时器 A 和定时器 B。定时器 A 主要用来产生串行端口的波特率、并行端口 D 和 E 的周期性时钟，或周期性中断。定时器 B 有同样的功能，但不能生成波特率时钟。定时器 B 使用时更灵活，因为程序可从一个持续运行的计数器读取时间，且事件可编程发生在指定的某未来时刻。

图 28 表明定时器 A 和 B 的时钟图。

11．1 定时器 A

定时器 A 由五个分离的递减计数定时器—A1 和 A4-A7 组成，如图 28 示。

定时器 A1 和 A4-A7 是 8 位递减计数寄存器，如图 29 示。重装载寄存器（reload register）可包含一个范围在 0-255 之间的任意数 n。计数器设成（n+1）分频模式。例如，如果重装载寄存器包含的数是 127，则在定时器右边输出一个脉冲之前，左边进入 128 个脉冲。如果它包含 0，则左边的每个脉冲导致右边输出一个脉冲，也就是说，为 1 分频模式。

定时器系统由外设时钟的二分频驱动。这个时钟总是与处理器时钟相同，或比后者快 1/8。输出脉冲一般为一个时钟脉冲长度。计数器的计时发生在脉冲的负边沿。当计数器到 0 时，重装载寄存器在下一输入脉冲上被载入，而不是执行一次计数。重装载寄存器可在任何时候重装载，因为外设时钟与处理器时钟同步。

定时器 A4、A5、A6 和 A7 一般为串行端口 A、B、C 和 D 分别提供波特率时钟。如果不是很低的波特率，不需要使用时钟 A1 为定时器 A4-A7 预分频时钟。例如，如果系统时钟是 11.0592MHz，同时定时器 A4 为 144 分频模式，可一步就获得异步波特率 2400bps。串行口的时钟必须是异步模式波特率的 16 倍或同步模式波特率的 8 倍。11.0592Mhz 的最大异步波特率是（11,059,200/(2*16)）=345,600。

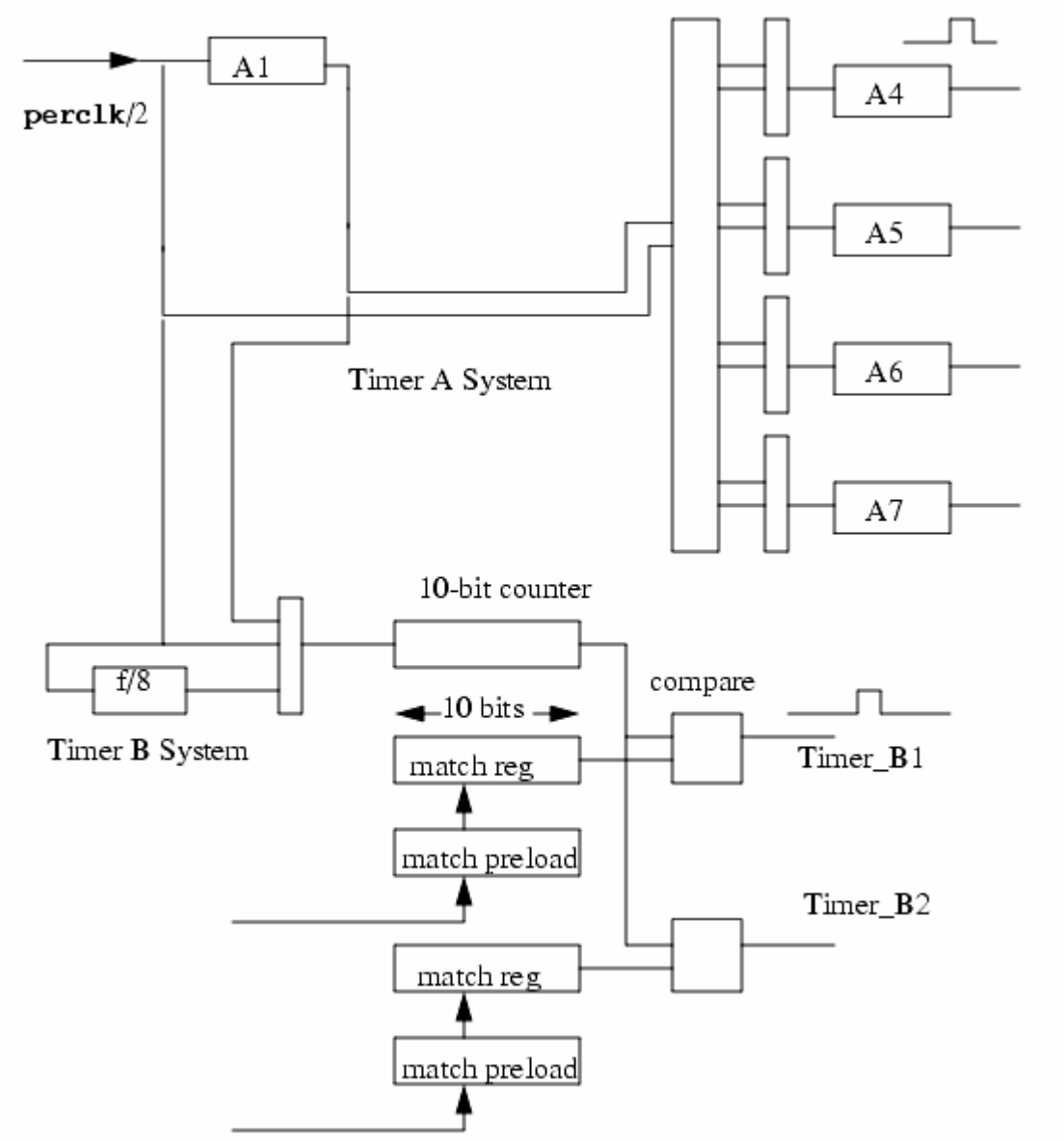


图 28 定时器 A 和 B 的框图

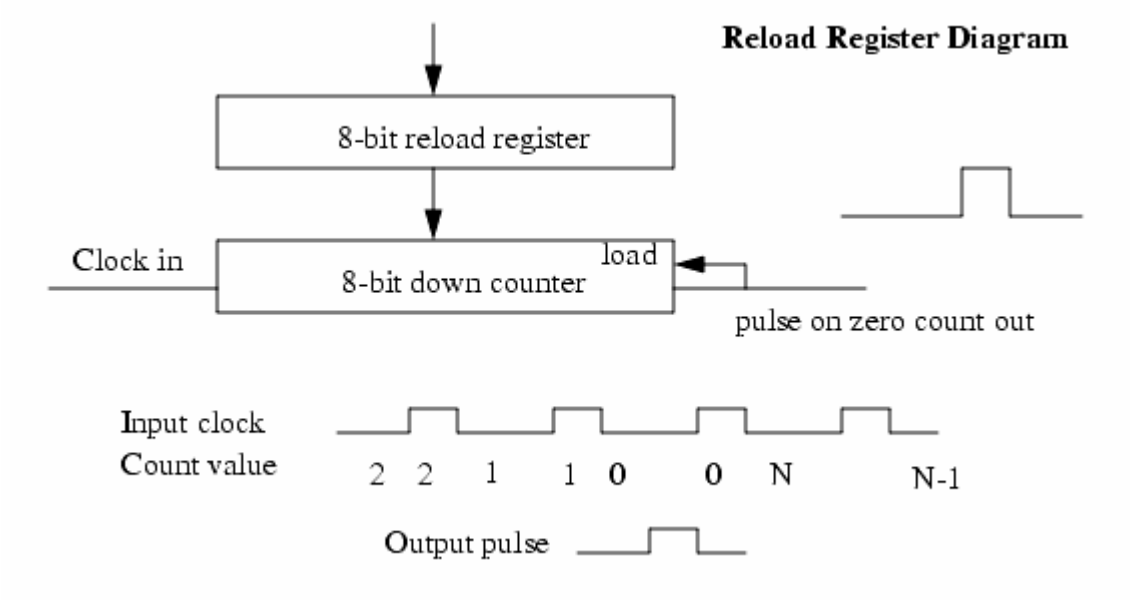


图 29　重装载寄存器的操作

定时器 A 的五个减计数，每个都可产生中断。定时器 A 有一个中断向量和普通中断优先级。普通状态寄存器（TACSR）中为每个定时器设有一个比特位，它表明从上次读状态寄存器以来是否有哪个定时器的输出脉冲发生。读一次状态寄存器，这些位清零，不会丢失任何位。这个位要么会被读状态寄存器的操作读取，要么在读状态寄存器动作完成后被置位。如果某位为“1”，且相应的中断使能，在优先级允许时就会发生中断。然而，就算中断使能也不能保证每个位都有各自的中断。如果在状态寄存器里读取了这个位，它被清零，也不会有相应于那个位的中断请求。还有可能某置位位将导致一个中断，然后一个或多个另外的位在读寄存器之前置位。然而这些位清零后，它们不能产生中断。如果有任何位置位，而且相应中断使能，只要优先级允许，中断发生。然而，如果这个位在中断被锁存之前清零，它不会导致中断。应该遵循的合适的原则，是让中断程序来处理它所看到的所有置位位。

11．1　定时器 A 的 I/O 寄存器

定时器 A 的 I/O 寄存器示于表 33。

表 33　定时器 A　I/O 寄存器

寄存器名	寄存器助记符	I/O 地址(hex)	R/W
定时器 A 控制/状态寄存器	TACSR	A0	R/W
定时器 A 控制寄存器	TACR	A4	W
定时器 A1 时间常数 1 寄存器	TAT1R	A3	W
定时器 A4 时间常数 4 寄存器	TAT4R	A9	W
定时器 A5 时间常数 5 寄存器	TAT5R	AB	W
定时器 A6 时间常数 6 寄存器	TAT6R	AD	W
定时器 A7 时间常数 7 寄存器	TAT7R	AF	W

定时器 A 的控制/状态寄存器（TACSR）在表 34 里单独列出。

表 34 定时器 A 控制和状态寄存器（地址=0A0h）

	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
读	A7 计数完	A6 计数完	A5 计数完	A4 计数完	0	0	A1 计数完	此位只写
写	A7 中断使能	A6 中断使能	A5 中断使能	A4 中断使能	x	x	A1 中断使能	1—使能定时器 A

位 1，4-7—读/写，在定时器 A1 和 A4-A7 上到达终点计数。读这个状态寄存器把任何置“1”位（位 1 和 4-7）清零。写这些位将使能相应定时器的中断。

位 0—只写，设为“1”时使能定时器 A 的时钟（perclk/2），设为“0”则禁止时钟（perclk/2，图 28）。

控制寄存器（TACR）在表 35 里列出。

表 35 定时器 A 控制寄存器（地址=0A4h）

位 7 A7	位 6 A6	位 5 A5	位 4 A4	位 3，2	位 1，0
A7 的时钟源 0—pclk/2 1—A1	A6 源 0-pclk/2 1-A1	A5 源 0-pclk/2 1-A1	A4 源 0-pclk/2 1-A1	不使用 忽略	00—禁止中断 01—使能优先级 1 中断 10—使能优先级 2 中断 11—使能优先级 3 中断

每个定时器的时间常数寄存器只是一个 8 位数据寄存器，容纳一个 0-255 之间的数。它是只写的。

11．1．2 定时器的实际应用

上电时定时器 A 被禁止（控制/状态寄存器里的位 0 指明）。定时器 A 一般在时钟被禁止时设置，但如果需要的话，可在定时器正运行时改变定时器的设置。没有使用的定时器应由 A1 的输出驱动，重装载寄存器应设为 255。这将使计数尽可能的慢，从而消耗最小电能。

定时器 A 有 5 个分离的子定时器单元，A1 和 A4-A5，也就是一般称作的定时器。

很多情况下，如果要使用一个串行端口并需要一个定时器为它提供波特时钟，使用的那个定时器配置为由系统时钟直接驱动，而且与此定时器相关的中断被禁止（串行端口中断由串行端口逻辑产生）。

重装载寄存器里的数值可在定时器运行时改变，以改变下一定时器循环的周期。重装载寄存器初始化后，减 1 计数器中的内容可能仍未知，例如，在上电初始化期间。如果中断开放，那么第一个中断会在一个不可知的时刻发生。与此相似，如果定时器输出用来驱动并行或串行端口的时钟，第一个时钟脉冲可能在一个随机时刻到来。如果需要的是周期性时钟信号，第一个时钟脉冲什么时候发生很可能并不重要，除非要求不同定时器之间有一个相位关系。

两个定时器间的相位关系可用几种方法得到。一种方法是把两个重装载寄存器都设为 0，从而等待两个定时器重装载要足够长的时间（最大 256 个时钟周期）。这样，在两个定时器开始计时之前或之后，

都可为重装载寄存器设置新值。

11．2 定时器 B

图 28 示出定时器 B 的框图。定时器 B 的计数器可直接用 perclk/2 、perclk 的 8 分频或定时器 A1 的输出驱动。定时器 B 有一个连续运行的 10 位计数器。这个计数器作为两个匹配寄存器的对比，即 B1 匹配寄存器和 B2 匹配寄存器。当此计数器转变成一个与匹配寄存器相同的值，产生一次 1 个时钟长度的内部脉冲。这个匹配脉冲可用来引起中断和/或给并行端口 D 和 E 的输出寄存器计时。匹配寄存器的内容从一个预装载寄存器里载入，可用一条 I/O 指令写入。当匹配脉冲输出时，数据位从预装载寄存器向前进入到匹配寄存器。

匹配寄存器从匹配预装载寄存器里载入，这可用一条 I/O 指令写入。当产生匹配脉冲时，匹配预装载寄存器里的数据位进入下一匹配寄存器。

任何时候出现匹配条件时，处理器设置一个内部位，来标记 TBLxR 中的匹配值无效。读 TBCSR 的动作将清除中断条件。TBLxR 必须重装载，以重新使能中断。任何时候都不需要重装载 TBMxR。

如果两个匹配寄存器都需要修改，最高有效字节必须首先改变。

定时器 B 的 I/O 寄存器列于表 36。

表 36　定时器 B 寄存器

寄存器名	寄存器助记符	I/O 地址（hex）	R/W	复位后的值
定时器 B 控制/状态寄存器	TBCSR	B0	R/W	xxxx x000
定时器 B 控制寄存器	TBCR	B1	W	xxxx xx00
定时器 B 的 MSB1 寄存器	TBM1R	B2		x
定时器 B 的 LSB1 寄存器	TBL1R	B3	W	x
定时器 B 的 MSB2 寄存器	TBM2R	B4	W	x
定时器 B 的 LSB2 寄存器	TBL2R	B5	W	x
定时器 B 计数 MSB 寄存器	TBCMR	BE	R	x
定时器 B 计数 LSB 寄存器	TBCLR	BF	R	x

定时器 B 的控制/状态寄存器（TBCSR）示于表 37。

表 37　定时器 B 控制和状态寄存器（TBCSR）（地址=0B0h）

位 7:3	位 2	位 1	位 0
未使用	1—检测到与匹配寄存器 2 的匹配状态。读这个寄存器后,此位清零；把此位置 1 使能中断。	1—检测到与匹配寄存器 1 的匹配状态。读这个寄存器之后，此位清零。把此位置 1 使能中断	1—使能这个定时器的主时钟。

定时器 B 的控制寄存器（TBCR）示于表 38。

表 38 定时器 B 控制寄存器（TBCR）

位 7:4	位 3,2	位 1:0
未使用	00—计数器时钟是 perclk/2 01—计数器时钟是定时器 A1 的输出 1x—定时器时钟是 perclk/2 的 8 分频	00—禁止中断 xx—使能优先级 xx 的中断

定时器 B 的 MXBx 寄存器（TBM1R/TBM2R）示于表 39。

表 39　定时器 B 的 MSBx 寄存器（TBM1R/TBM2R=0B2h/0B4h）

位 7:6	位 5:0
定时器匹配预装载寄存器两个最高有效位。	未使用

11．2．1 使用定时器 B

一般设置预定标器的目的是用一个值分频 perclk/2，以提供一个针对实际问题的合适的计数率。例如，如果时钟是 22.1184MHz，那么 perclk/2 是 11.0592MHz。时钟是 11.0592 MHz 的定时器 B 将在 92.6 微秒内完成一个 10 位时钟的完整循环。

一般情况下，定时器里的任一个补偿器产生一个脉冲时都会发生一个中断。中断程序必须侦测到是哪个补偿器导致了中断，并把这个中断发派给中断服务程序。服务程序配置下一个匹配值，它将成为下一个中断发生后的匹配值。如果时钟驱动的并行端口正使用，那么一般会向并行端口寄存器的某些位里装载一个值。这些位将成为下一个匹配脉冲时的输出（为并行端口保持一个影子寄存器很有必要，除非使用了端口 D 和 E 的按位编址特性）。

或者在由匹配脉冲引起的中断期间，或者在其他一些与匹配脉冲异步的中断程序里，如果我们希望从定时器 B 的计数器里读取时间，则需要使用一个特别程序段来读这个计数器，因为计数器的高 2 位在与低 8 位在不同的寄存器里。建议使用下面的方法。

1．读低 8 位
2．读高 2 位
3．再次读低 8 位
4．如果在第一次和第二次读低 8 位之间，位 7 从 1 变到 0，则已经产生到高 2 位的进位。这种情况下，再次读高 2 位，并对那 2 位进行递减操作，以获得正确的高 2 位值，并使用第一次读的低 8 位。

这个程序假定两次读之间的时间保证小于 256 次计数。这个条件在绝大多数系统里可通过禁止优先级 1 的中断得到保证，优先级 1 的中断在中断程序里一般都禁止。

如果由于高优先级的中断（级别 2 和 3）会妨碍目标的实现而禁止它们是不合适的的做法。

如果速度是关键问题，可以不检验进位就执行这三次读寄存器的操作。可以保留这三个寄存器值，进位检验也可由一个较低优先级的分析程序执行。由于高 2 位在地址为 0BEh 的寄存器 TBCMR 里，并且低 8 位在地址为 0BFh 的寄存器 TBCLR 里，两个寄存器都可以用一条 16 位 I/O 指令读取。下面的指令序列表明怎么读寄存器。

；有脉冲输出跳变时，从外部中断进入此程序

```
; 19 个时钟的等待时间和 10 个时钟的中断执行时间
push af   ; 7
push hl
ioi ld a,(TBCLR)       ; 11   获得计数器的低 8 位
ioi  ld  hl,(TBCMR)   ;13     获得 l=高 2 位, h=低 8 位
```

定时器 B 可用于多种用途。可以在某事件发生时读这个 10 位计数器，来记录发生时刻。如果此事件导致一个中断，可以在中断程序里读这个定时器。在里面减去中断程序的已知执行时间。但可变的中断等待时间是事件执行时间中的一个不确定因素。如果中断是最高优先级，等待可以小到 19 个时钟周期。如果系统时钟是 20MHz，计数器可以 10MHz 的速度计数。在 20MHz 的系统时钟里，脉冲宽度计量的不确定度几乎可低到 38 个时钟周期（2*19），或者说 2 微秒。

定时器 B 可用来在将来的某个特别的指定时刻改变某并行端口的输出寄存器。可以在一定限制下在任意时刻产生脉冲沿序列，此限制是两个相邻沿不能靠的太近，因为在每个边沿后都要提供一个中断服务而为下个边沿配置时间。这个限制使最小脉冲宽度大约 5 微秒，并依赖于时钟速度和中断级别。

12．Rabbit 的串行端口

Rabbit 有四个芯片内的串行端口 A、B、C 和 D。所有端口都可以在高波特率上执行异步串行通信。端口 A 和 B 具有可作为定时端口和可转换为备用 I/O 引脚的附加能力。端口 A 还有特殊的功能，它可用来执行微处理器系统的冷启动。

表 40 列出同步串行端口信号。

表 40 同步串行端口信号

Rabbit 信号名称	引脚功能
CLKA 或 CLKB	串行时钟
并行端口 C 上的 TxA 或 TxB[1] 并行端口 D 上的 ATxA 或 ATxB	数据传送
并行端口 C 上的 RxA 或 RxB 并行端口 D 上的 ARxA 或 ARxB	数据接收

[1] 串行端口 A 和 B 可在并行端口 C 和 D 之间多路复用。然而，只有串行端口 A 或 B 可配置为可在任何特定时刻在这些并行端口之一上工作。

图 30 示出串行端口的框图。

独立的串行端口能够在异步模式下以超过 500,000bps 的波特率工作，比并行模式快 8 倍。异步模式下可以发送或接收 7 或 8 个数据位，同时也支持所谓的“第 9 位”或地址位的工作模式。硬件不直接支持奇偶校验位和多停止位，但可以通过适当的编程技巧完成。

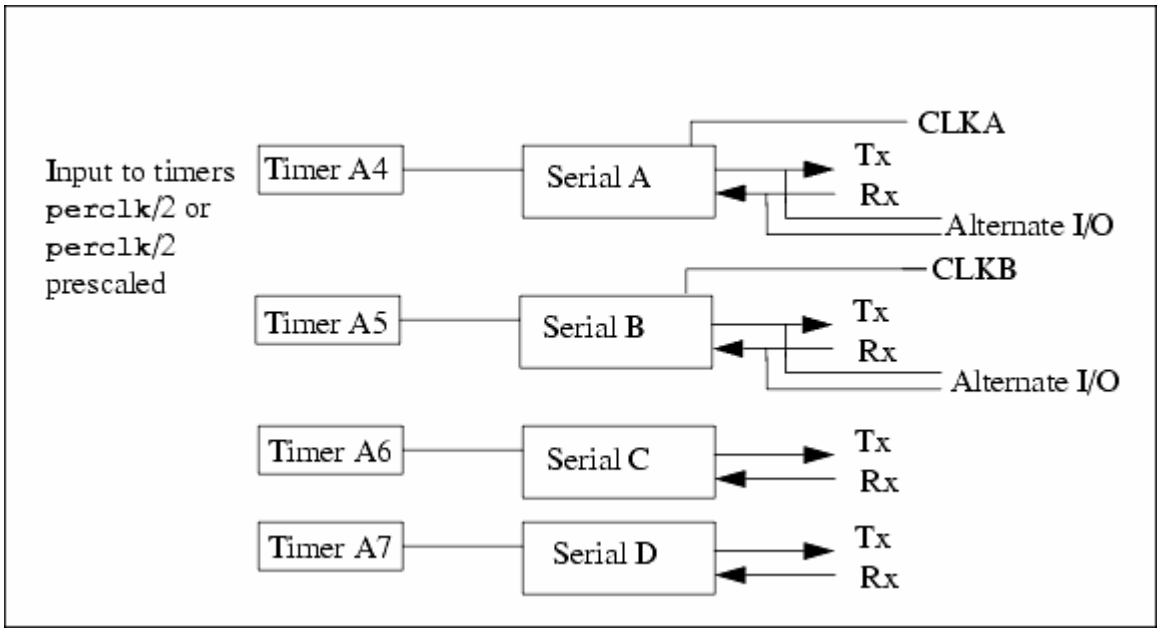


图 30 Rabbit 串行端口框图

12．1 串行端口的寄存器规划（Register Layout Serial Port）

表 31 示出一个串行端口的功能框图。每个串行端口有一个数据寄存器、一个控制寄存器和一个状态寄存器。写数据寄存器的动作开始一个传送操作。如果是对一个备用数据寄存器地址执行写操作，则发送额外地址位或第 9 位。数据接收到后，在数据寄存器处读取它们。用控制寄存器来设置发送和接收参数。可测试状态寄存器以检查串行端口的工作状态。

串行端口单元的时钟输入必须是异步模式波特率的 16 倍和使用内部时钟时的定时串行模式波特率的 2 倍。定时器 A4-A7 为串行端口 A-D 提供输入时钟。这些定时器可以用 1-256 之间的任意数进行分频。可以用定时器章节中（11 章）中提出的多种方法来选择定时器的输入频率。一种选择是系统时钟的 2 分频——使用这种选择并精心挑选一个主振荡器的晶体频率，可获得最常用的波特率，在最高 Rabbit 时钟频率下则可达大约 2400bps（参看附录 A.2 节）。

表 41 列出串行端口寄存器。

表 41　串行端口寄存器

寄存器	**地址 xx=00,01,10,11 (适用 A，B，C，D)**	**助记符 x=A，B，C，D**
数据寄存器	**11xx 0000**	**SxDR**
发送第 9（第 8）地址位的备用数据寄存器	**11xx 0001**	**SxAR**
状态寄存器（可读、写，写则清除发送 IRQ）	**11xx 0011**	**SxSR**
控制寄存器（只写）	**11xx 0100**	**SxCR**

串行端口中断向量示于表 17，第 7 章。

表 42 描述串行端口状态寄存器。

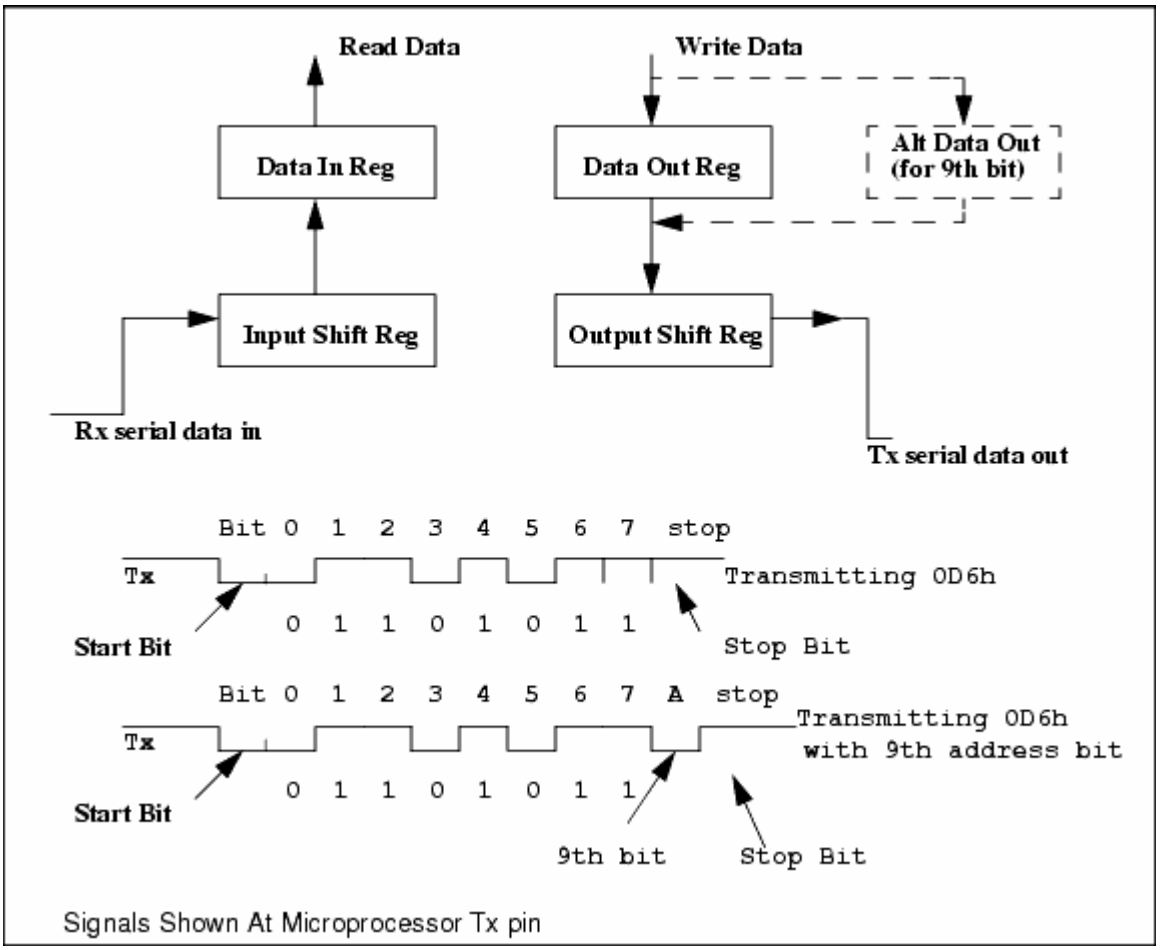


图 31 串行端口功能框图

表 42 串行端口状态寄存器（adr=11xx 0011,xx=A,B,C,D）

位 7	位 6	位 5	位 4	位 3	位 2	位 1，0
接收器就绪(接收数据寄存器里有一个字节)	接收到第 9 位	接收缓冲区超限	0	发送器数据寄存器满	发送器正发送一个字节	0,0

写状态寄存器的动作会清除发送中断请求 FF，但没有其他作用。

位 7——接收器就绪。当一个字节从接收器的移位寄存器传输到接收器数据寄存器时，这个位被置位。当对接收器数据寄存器进行读操作时，此位清零。从“0”到“1”的转换动作会置位接收器的中断请求触发器。

位 6——地址位或第 9（第 8）位。如果接收器数据寄存器里的字符有第 9（或第 8）位，这个位置位。这个位在清零后，在读数据寄存器之前还应该检查它，因为读数据寄存器之后可能马上会载入一个带有新地址位的新数据值。

位 5——接收器超限时置位(指接收器不能按照要求的速度接收数据)。这发生在移位寄存器和数据寄存器满并且又检测到一个开始位情况下。读接收器数据寄存器时，这个位清零。

位 3——发送器数据缓冲区满。发送数据寄存器满时置位，就是说，已经写了一个字节到串行端口数

据寄存器。传输一个字节到发送器移位寄存器，或执行了对串行端口状态寄存器写操作，它清零。中断使能时，这个位会在从 1 变化到 0 时请求一个中断。

位 2——发送器忙位（busy bit）。发送器移位寄存器发送数据正忙时，它置位。在开始位的下降沿置位，而且开始位的下降沿还把数据从发送器数据寄存器传输到发送器移位寄存器。发送器忙位在所发送字符的停止位的末端清零。最后一个字符发送完后（发送器数据寄存器里不再有数据），这个位从忙变到不忙时会产生中断。

位 0,1,4——一直是 0。

表 43 描述串行端口控制寄存器。

表 43　串行端口控制寄存器（地址=11xx 0100,xx=A,B,C,D）

位 7，6	位 5，4	位 3，2	位 1，0
00—空操作 01—接收 1 字节，时钟同步下（A，B） 10—发送 1 字节，定时模式下（A，B） 11—保留，将来使用	00—端口 C 用作串行输入 01—端口 D 用作串行输入 1x—禁止接收器的输入	00—异步模式，8 位 01—异步模式，7 位 10—定时模式，外部时钟（A，B） 11—始终控制模式，内部时钟（A，B）	00—无中断 01—级别 1 中断 10—级别 2 中断 11—级别 3 中断

位 7,6——异步模式下,这两位通常是 0。对于端口 A 和 B，如果定时串行模式使能，在这两位里存入代码来开始一个操作，或接收或发送。如果是内部时钟，用 8 个时钟脉冲的脉冲串驱动时钟线。外部时钟下，接收器或发送器等待外部提供的 8 时钟脉冲串。

位 5,4——这两位使能端口的标准或备用引脚。端口使能时，指定的 Tx 引脚的并行端口输出功能被禁止。并行端口 C 功能寄存器（PCFR）和并行端口 D 功能寄存器（PDFR）里的设置用来使能端口 C 和 D 的串行输出（参看 9.3 节“并行端口 C”和 9.4 节“并行端口 D”）。

位 3,2——这两位使能中断并设置中断级别。

12．2 串行端口中断

接收和发送中断使用一个普通中断向量。接收器和发送器各有一个独立的中断申请触发器。如果设置了任一个触发器，会产生一个串行口中断请求。触发器只由收发操作信号上升沿置位，由 I/O 读或写操作所产生的脉冲清零，如图 32 示。当请求中断，且优先级允许并且完成执行了一条指令，中断得到响应。如果请求触发器在中断发生之前清零，中断丢失。如果在中断程序里没有清除中断申请触发器，其它优先级低于该中断时会产生另一个中断。

接收时，接收中断请求触发器在采样到终止位时被置位，采样一般在终止位的 1/2 处进行。数据位在此时从移位寄存器传输到接收数据寄存器。

发送中断请求触发器在停止位的前沿（leading edge）和后沿（trailing edge）都可置位，前者要求数据寄存器空，后者要求移位寄存器空（发送器空闲）。除非数据寄存器在停止位的后沿时是空的，不然发送器不会空闲。而发送器仅在数据寄存器在终止位下降沿时空的情况下才会空闲。

表 17 示出串行端口中断向量（第 7 章）。

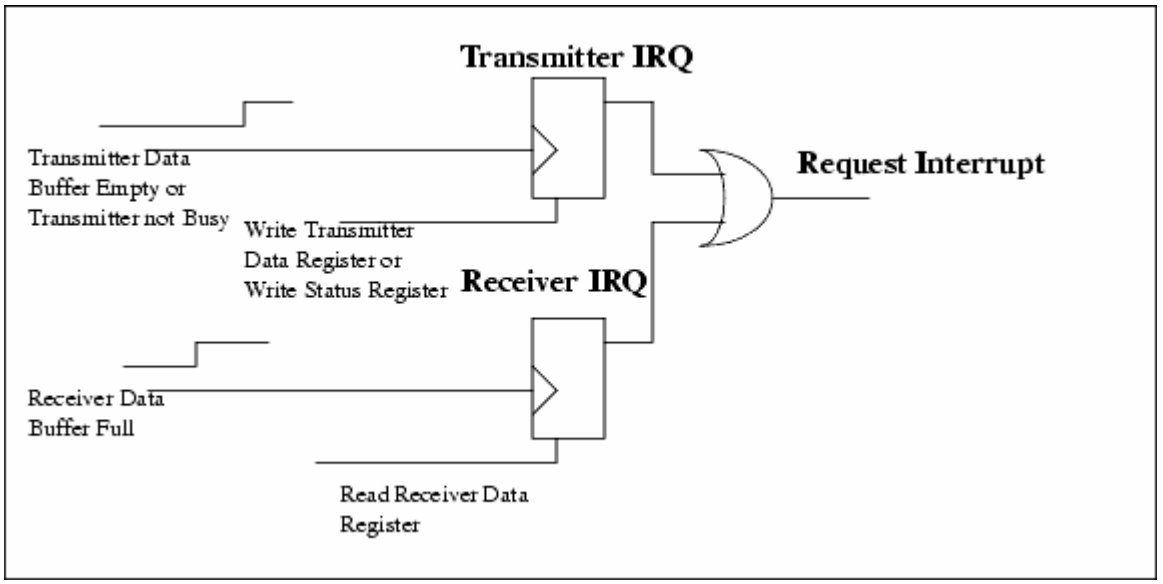


图 32 串行端口中断的产生

12．3 发送串行数据的时序（Transmit Serial Data Timing）

传送数据时，如果中断允许，当送寄存器变成空时，申请中断，此外，当移位寄存器和发送寄存器都是空，也就是发送器空闲时，产生一个中断。当发送数据寄存器装有数据并且移位寄存器完成了发送数据，数据位定时从发送寄存器进入移位寄存器，移位寄存器永不会空闲。或由写数据寄存器的动作，或由写状态寄存器的动作（但并不影响状态寄存器），来清除中断请求。每次移位寄存器完成发送数据而使数据寄存器空时，数据寄存器内容通常定时进入移位寄存器。这导致一个中断请求。中断程序一般在移位寄存器干转（run dry）（9 到 11 个波特时钟，依赖于操作模式）之前应答中断。中断程序把下一个数据项存储在数据寄存器里，并清除中断请求，提供接着要传送的数据位。当所有字符都传送完之后，一旦数据寄存器变空，中断服务程序应答中断。由于没有更多的数据了，它通过存储状态寄存器的操作清除中断请求。这时，中断程序应该检验移位寄存器是否为空，一般不为空。如果是空的，因为中断被滞后应答，中断程序应该做所有清除工作，并在清除挂起的中断后移位寄存器为空的情况下，向状态寄存器做存储操作。一般情况下，中断服务程序将返回，并还有一个可让中断程序禁止输出缓冲区的最后的中断，如 RS-485 传送的情况。

12．4 接收串行数据的时序（Receive Serial Data Timing）

接收器就绪时，一个下降沿表明检测到了起使位。下降沿在两个不同时钟脉冲检测到，它作为 Rx 输入也是不同的，时钟为 16 倍波特率。一旦检测到起始位，在每个数据位的中间采样数据位，并移入接收移位寄存器。接收了 7 或 8 个数据位之后，下一位或者是第 9（第 8）地址位，或者是停止位。如果 Rx 线是低电位，它是一个地址位，并且状态寄存器里接收到的数据位使能。检测到地址位后，接收器会试着采样停止位。如果 Rx 线是高电位，它是一个停止位，并在采样点之后开始扫描新的起始位。同时，数据位被传输进入接收数据寄存器，并请求一个中断（如果中断使能）。

接收数据时，如果接收器的数据寄存器有数据，将请求一个中断。这发生在数据位从接收移位寄存器传输到数据寄存器时。它还把状态寄存器的位 7 置位。读数据寄存器时，中断请求和位 7 被清除。

如果位 7 为高，请求一个中断。中断请求发生在发送器数据寄存器变成空或发送器移位寄存器变成空的脉冲沿处。通过写状态寄存器或数据寄存器来清除发送器的中断。

接收时，在检测到停止位时立即开始对下一个起使位的扫描。停止位一般在采样时钟脉冲上检测到，这个采样时钟理论上应发生在停止位的中间。如果存在第 9（第 8）地址位，停止位跟随在这个地址位之后。

12．5 时钟同步的串行端口（Clocked Serial Ports）

端口 A 和 B 可工作于时钟同步模式。图 33 示出数据线和时钟线的驱动方式。数据和时钟信号是脉冲串。发送移位寄存器在时钟脉冲的下降沿处进行前移。接收器在时钟的上升沿采样数据。串行端口可以产生时钟信号，也可由外部提供时钟。

为使能定时模式，必须在控制寄存器的位（3,2）里存有一个代码，以用内部或外部时钟来使能定时模式。外部和内部时钟之间的转换必须小心执行。一般在时钟线上需要一个上拉电阻，以防止任一方都没有对时钟驱动时的虚假时钟（spurious clock）。

定时串行模式下，异步通信时移位寄存器和数据寄存器工作于相同方式下。然而，为启动发送或接收，控制寄存器的位（7，6）里必须存储一个代码，且代码对任何接收或发送的字节相同。一种代码指明发送一个字节，另一种不同代码指明接收一个字节。这些代码的作用不同，依赖于是内部时钟还是外部时钟模式。

为了在内部时钟模式下发送数据，用户必须首先装载数据寄存器（它应该是空的），然后存入发送码。当移位寄存器完成发送当前字符，数据寄存器内容被装入移位寄存器，用一个 8-时钟脉冲串串送。一个字符正处于发送过程中时，另一个标记有发送码的字符可以在数据寄存器里等待。发送码有效地两倍缓冲。

为了在内部时钟模式下接收一个字符，接收移位寄存器应该空闲。然后用户向控制寄存器里存入接收码。然后生成一个 8 时钟脉冲串，发送方必须检测到这些时钟信号，然后在每个时钟的下降沿把输出数据移到数据线上。接收器在每个时钟的上升沿采样数据。接收模式在使用内部时钟时不能把字符两倍缓冲。移位寄存器在启动另一次字符接收之前必须处于空闲。然而，中断请求和字符就绪发生在最后一个时钟脉冲的上升沿。如果下一个接收码在下一个下降沿的自然位置之前存储，另一个接收动作不经暂停时钟就会启动。为了这么做，必须在 1/2 个时钟周期里为中断提供服务。

在外部时钟模式下发送每个字节，用户必须装载数据寄存器，然后存入发送码。当移位寄存器空闲和接收器提供时钟脉冲串时，数据位传输到移位寄存器并移出去。一旦传输进行到了移位寄存器上，可以向发送寄存器里装载一个新的字节，并存入新的发送码。

在外部时钟模式下接收字节，用户必须为第一个字节设置接收码，然后在每个字节从数据寄存器里移去之后为下一个字节存入接收码。由于接收码必须在发送器发送下一字节之前存入，接收器必须在 1/2 个波特时钟里服务中断，以保持全速发送。这一般都不实用，除非做了流量控制安排或发送器在时钟

脉冲串之间插入间隔。

为了进行高速通信，最佳布置方案通常是围绕接收器来提供时钟信号。当接收器提供时钟时，发送器应该一直有能力继续工作，因为它是两倍缓冲的且有充分的字符时间（character time）来应答发送器数据寄存器空的中断。接收器将应答在最后一个时钟上升沿上产生的中断。如果可以在 1/2 个时钟脉冲里服务中断，就不会有数据率的停顿。如果接收器需要更长的时间应答中断，会在字节之间产生间隔，间隔的长度决定于中断等待时间。举个例子，如果波特率是 400,000bps，每秒最多可发送 50,000 字节，或每个字节 20μs。如果接收器能够在 1/2 个时钟周期或 1.25μs 内应答其中断，不会发生减速。如果它能够在 1.5 个时钟或 2.75μs 内应答中断，数据传输率降到 44,444bps。如果可在 2.5 个时钟或 6.25μs 内应答，速率降到 40,000bps。如果可在 3.5 个时钟或 8.75μs 内应答，速率为 36,363bps。依此类推。

如果希望进行双向的半双工（half-duplex）通信，可倒转时钟方向，由接收器一直提供时钟。这稍微有些复杂，因为接收器不能启动一个消息。如果接收器试图接收一个字符而发送器没有发送，所发送的最后一位被接收作为所有八位的值。

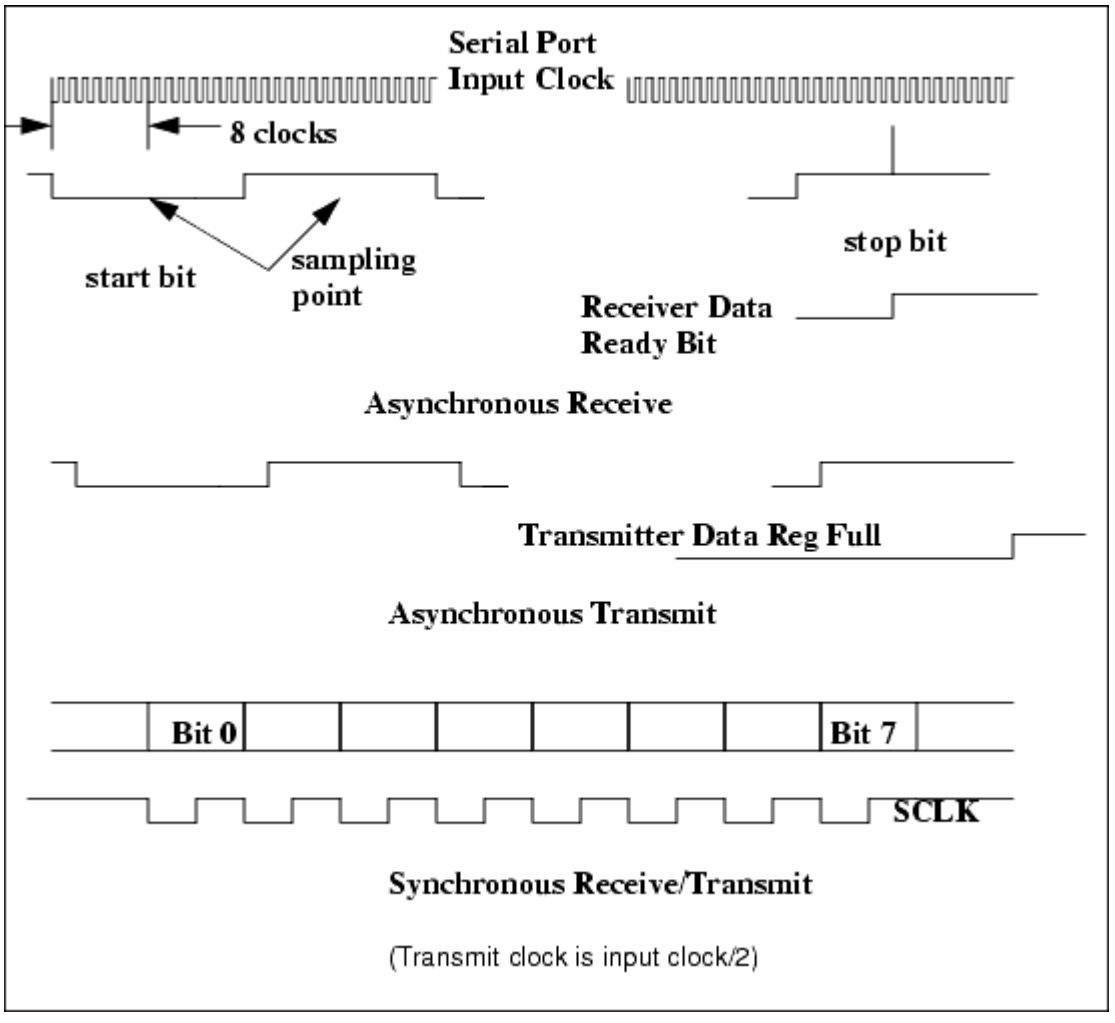


图 33　串行端口同步化

12．6 定时串行的时序（Clocked Serial Timing）

12．6．1 使用内部时钟的定时串行时序

为进行同步串行通信，串行时钟可由 Rabbit 或外部器件产生。图 34 中的同步框图适用于全双工和半双工定时串行通信，要求串行时钟由 Rabbit 内部产生。使用内部时钟，最大串行时钟速率是 perclk/4。

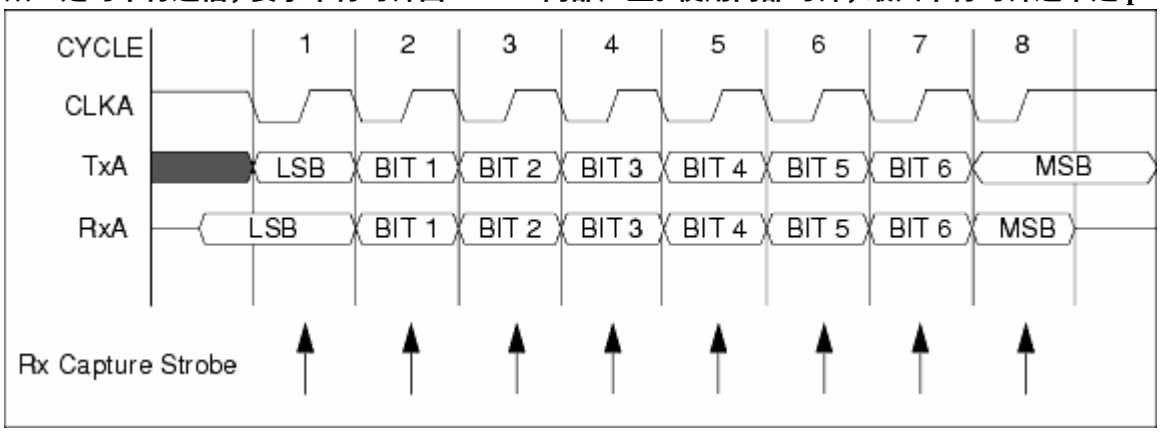


图 34 使用内部时钟的全双工定时串行时序框图

12．6．2 使用外部时钟的定时串行时序

在 Rabbit 的串行时钟由外部器件产生的系统里，Rabbit 能够发送或接收数据之前，串行时钟信号必须与内部的外设时钟（perclk）同步。根据外部串行时钟产生的时刻，传输任何数据之前都需要 2 到 3 个时钟周期才能使外部时钟与此内部时钟（perclk）同步。图 35 表明 perclk、外部串行时钟和数据传送之间的时序关系。

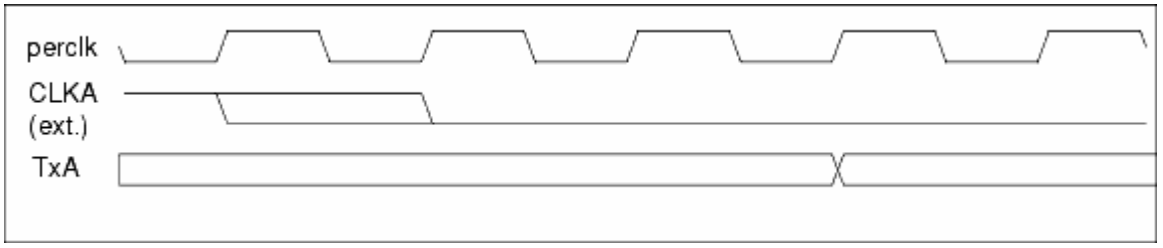


图 35 使用外部时钟的同步的串行数据发送同步方法

图 36 表明 perclk、外部串行时钟和数据接收之间的时序关系。注意 RxA 在 perclk 的上升沿采样。

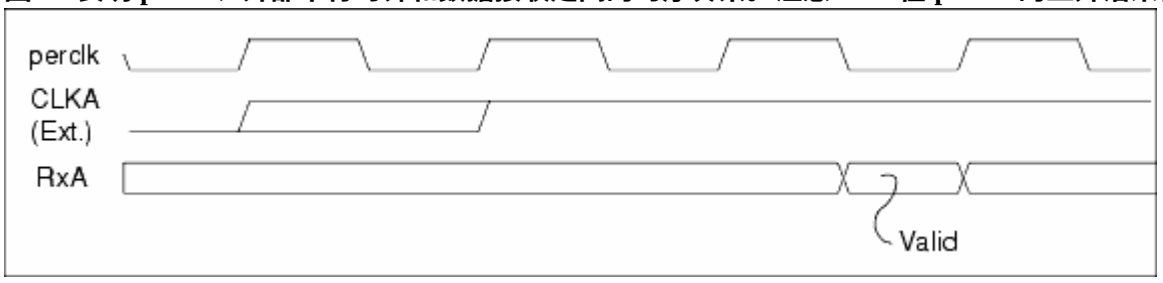


图 36 使用外部时钟的同步串行数据接收同步方法

Rabbit 的时钟由外部提供时，最大串行时钟频率受到使外部时钟与 Rabbit 的 perclk 同步所需时间的限制。如果把每次接收和发送时执行时钟同步所需的最大 perclk 周期数求和，则最快外部串行时钟频

率应限制在 perclk/6。

12．7 串行端口软件建议（Serial Port Software Suggestions）

接收器和发送器共享同一个中断向量，但通过分派中断给发送和接收中断程序中的一个，可以使发送和接收的中断服务程序（ISR）分离。这使 ISR 简单些，并减少中断时间，是很好的特性。这不会丢失任何中断，因为接收和发送有明显不同的触发器。中断分派器可以检验接收器的数据寄存器的满位（full bit）以决定怎么分派。如果这个位开启，中断分派给接收，否则用于发送。接收器接收首先需要考虑的事项，因为对它的服务必须很小心，不然数据会丢失。

分派器可为如下形式：

```
interrupt:

push  af              ; 10
ioi ld a,(SCSR)  ; 7   获得串口 C 的状态寄存器
or  a,a               ; 2   测试符号位
jp  m,receive       ; 7   为接收中断服务
jp  transmit         ; 7  (至此 41 个时钟)为发送中断服务
```

各个中断假设寄存器 AF 已经保存，且状态寄存器内容被装载入寄存器 A。

作为一种好的实际应用和为获得最佳执行效果，中断服务程序可以移去中断的导致原因，并只要可能就重新使能中断。这使中断等待时间保持较小，并允许在所有串行端口上有最快的传送速度。

所有串行端口一般产生优先级别 1 的中断。在特殊情况下，可以配置一个或多个串行端口来使用更高级别的中断。有一个例外需要注意，就是当串行端口必须以极高的速度工作时。在 PC 串行端口的最高速度 115,200bps 上，中断服务必须在 10 个波特时间或 86μs 内完成，目的是不丢失所接收字符。如果所有四个串行端口都工作于这个接收速度，必须在 21.5μs 内服务中断，以保证不丢失字符。此外，还要考虑其他相同或更高级别的中断占用的时间。一个接收服务程序可为下面的形式。其中 bufptr 上的字节用来指出缓冲区的地址，数据位就存储在这里。必须保存并递加这个字节，因为如果两个接收器中断很快的接连发生，字符处理可能变的无序。

```
receive:

push hl   ; 10   保存  hl
push  de   ; 10    保存 de
ld hl,struct ; 6
ld a,(hl) ; 5  获得内指针
ld e,a   ; 2   把内指针存在 e 中
inc hl   ; 2   指向外指针
cmp  a,(hl)   ; 5    内指针是否等于外指针（缓冲区满）
jr z,roverrun   ; 5   处理超速的接收器
inc a   ; 2   内指针递增
and a,mask ; 4   与掩码如 1111 0000
```

```
dec hl ; 2
ld (hl),a          ; 6  更新内指针
ioi ld a,(SCDR)  ; 11  获得端口 C 数据寄存器，清中断请求
ipres  ; 4  恢复优先级


; 至此 68 个时钟
; 中断发生前降低级别
; 现在可以发生更多中断了，
; 但接收器数据在寄存器里
; 现在处理接收器中断程序的剩余部分
ld hl,bufbase ; 6
ld d,0 ; 6
add hl,de  ; 2  存储数据的地方
ld (hl),a  ;  6  处理掉这个数据字节
pop de ;7
pop hl ; 7
pop af ; 7
ret     ; 8  从中断返回

; 至此 117 时钟
```

这个程序使中断在 68 个时钟或 3.5μs 内开启，当时钟速度是 20MHz 时。虽然可能无序地处理了两个字符，它对检验输入缓冲区状态的高级别的中断程序是不可见的，因为所有中断都会在高级别程序对缓冲区状态执行检验之前完成。

组织缓冲区的典型方法是设置一个内指针（in-pointer）和一个外指针（out-pointer），它们以循环方式通过数据缓冲区的地址递增。此中断程序操纵内指针，高级别中断程序操作外指针。如果内指针等于外指针，缓冲区被认为满。如果外指针加 1 等于内指针，缓冲区空。所有递增操作以循环方式进行，其中大部分可通过使缓冲区长度为 2 的幂来完成，接着在递增之后与掩码做“与”运算。实际的内存地址是指针加上缓冲区基地址。

12．7．1 控制 RS-485 驱动器和接收器（Controling an RS-485 Driver and Receiver）

RS-485 使用半双工通信方式。一个终端（station）使能它的驱动器并发送一条消息。消息发送完之后，这个终端禁止驱动器，并在线上侦听答复。驱动器必须在发送起使位之前使能，并在发送停止位之前不可被禁止。发送器空闲中断一般用来禁止 RS-485 驱动器，也可使能接收器。

12．7．2 传送哑字符（Transmitting Dummy Characters）

有时候可能需要操作串行发送器但不发送任何实际数据。可利用发送“哑”字符来跳过或量度时间。

通过操纵作为输出引脚使用的并行端口 C 和 D 的控制寄存器，可使发送器的输出与发送器的输出引脚断开。举个例子，如果串行端口 B 要暂时与它的输出引脚，即并行端口 C 的位 4 断开，可如下进行。

1．存储“1”到并行端口数据输出寄存器的位 4，提供驱动线的静止状态（quiescent state）。

2．把并行端口 C 功能寄存器的位 4 清零，这样输出不再来自串行端口。当然，发送器空闲之前不可以这么做。

如果串行端口被配置使用端口 D 上的备用输出引脚，可用一个相似的过程完成。只有串行端口 A 和 B 可使用并行端口 D 上的备用输出。

如果使用了 RS-485 驱动器，可用发送停止位后禁止驱动器的方法来传送哑字符。这是上面的过程的一种备用选择。

12．7．3　传送和检测断点（Transmitting and Detecting Break）

发送器的输出被驱动为低电位并用于一个扩展周期时，创建一个断点。如果接收到一个断点，它表现为一系列字符，这些字符填入 0 和为低位的第 9 位。如果实行第 9 位有效的协议的话，这只会与合法的消息混淆，因为在这些协议中一般不使用断点作为消息。

用很低的波特率发送一个 0 字节，可以发送一个断点。另一种也许更好的方法，是把发送器与其输出引脚断开，并在正发送哑字符以使断点超时时，使用相应并行端口位来设置此线为低电位。

应该避免使用断点作为发信号设备，因为它很慢，且不同类型硬件的支持也无规律。它产生的问题一般比它解决的多。

12．7．4　使用串行端口来产生周期性中断（Using A Serial Port to Generate a Periodic Interrupt）

通过不断的传送字符，可使用串行端口产生周期中断。由于经由并行端口 C 或 D 的 Tx 输出可被禁止，所传送的字符没有传送到任何地方。由于字符输出路径是双倍缓冲的，字符传送不会有间隔，而且中断也正好会是周期性的。中断可每 9，10 或 11 个波特时间发生一次，决定于传送的是 7 位还是 8 位和第 9（第 8）位是否被发送了。

12．7．5　额外的停止位，发送奇偶校验位，第 9 位通信方案

一些系统可能需要两个停止位。在某些情况下，还需要发送一个奇偶校验位。某些特定系统，如一些基于 8051 的多点通信系统，使用第 9 个数据位来标记一个消息帧的起使。Rabbit 2000 可以无误地接收奇偶位或包含第 9 位的消息格式。它还可以发送有奇偶位的消息或一直装有第 9 位的消息。对只用 7 个数据位的字节格式，这么做非常简单，此时的第 9 位或奇偶位实际上是第 8 位。对发送器软件情况会有点混乱，如果有 8 个数据位和一个第 9 奇偶校验位，或需要一个信号位时。发送一个第 9 低电位由硬件支持。发送一个第 9 高电位需要延迟下一个字符的发送一个波特时间，这有效地提供了第 9 位高位和一个停止位，这相当于两个停止位。

图 37 解释标准异步串行输出形式。

12．7．5．1 奇偶位，7 个数据位字符的额外停止位

如果只发送 7 个数据位，则发送一个附加奇偶位或信号位的问题，可通过发送 8 个位并总是把字节的位 7（第 8 位）设置成“1”或“0”，而很容易的解决；设为“1”或“0”取决于需要。即使要接收两个停止位，也不需要任何特殊预防措施。如果接收了 7 个数据位之后还接收了奇偶位，把此数据看作 8 位，奇偶位在字节的最高位。

12．7．5．2 奇偶位，8 个数据位字符的额外停止位

为了接收 8 个数据位之外的奇偶位，要对每个字符进行第 9 位为低的检验。如果串行端口状态寄存器的位 6 在字符接收后设为“1”，第 9 位（或说奇偶位）值为低。如果第 9 位不是 0，串行端口把它当作额外的停止位。所以，如果没有设置第 9 位为低标志，应该假定奇偶位是“1”。

不需要为接收额外停止位做预防措施，也不需要为多于 1 位的停止位进行串行端口检验。如果第 1 停止位丢失，把它作第 9（第 8）位（低）看待，并作为一个 9 位（8 位）字符接收。

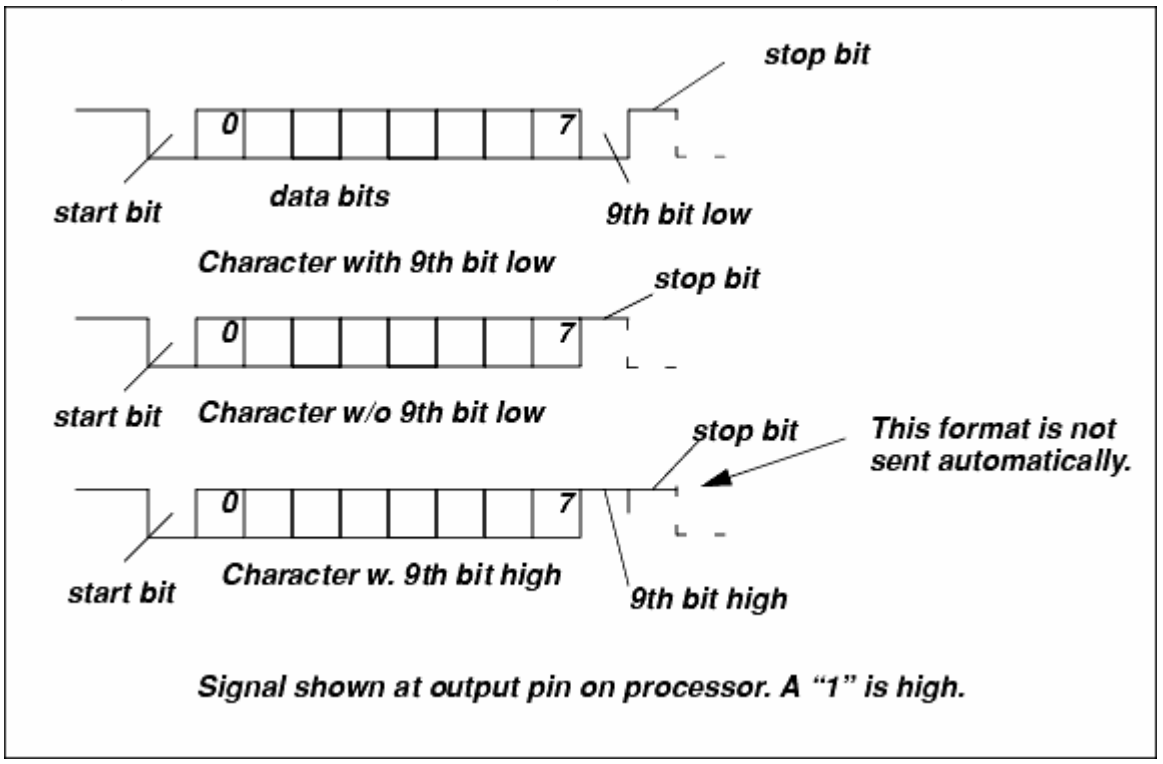


图 37　异步串行输出模式

传送额外的值为“1”的停止位或奇偶校验位有一些困难，因为没有适用于所有情况的解决方法，虽然每种情况都有一种解决方法。为发送额外的值“1”的停止位或校验位，必须延迟发送下一个字符，这样停止位将延伸为至少两个波特时间长。为了用一个附加波特时间来延迟下一个字符，程序必须等待发送器空闲的中断，这个中断发生在数据寄存器空的中断之后。数据寄存器就绪的中断请求通过写状态寄存器来终止。发送器空闲中断（发生在停止位的后沿）之后不允许中断程序在另一个波特时间装载下一个字符，例如，波特率 115,200bps 时 8.6μs 或 9,600bps 时 104μs。最高波特率时，在中断程序里使用一个忙等循环于下一个字符载入数据寄存器之前使一个波特步（baute step）超时。这个忙等循环可以很简短，因为延迟可由下面的时间做部分补偿：用于保存中断入口的寄存器的时间，用于从传送缓冲区取出下一个要发送的字符的时间。当然，忙等循环工作在处理器时钟上，它可调高或调低，因此循环计数必须与当前处理器速度协调。

在较低波特率下仍旧可以使用忙等循环，但这时会对中断等待时间产生有害影响，除非在中断程序里重新使能中断。这肯定可以做到，只要接收器中断和发送器中断被适当地分派了独立的程序，因为它们共享同一个中断向量。此外，当中断在中断程序里重新使能时，必须与实时核心或操作系统（如果有的话）进行协调。这种协调通常涉及中断程序的嵌套计数，此计数由每个重新使能中断的中断程序在返回之前进行大的调整。如果使用了忙等循环，字符发送时它占用大概 10% 的处理器计算时间，因

为它在发送每个字符的 11 个波特时间中忙等一个。使用发送器空闲中断请求下一字符，将导致字符间的间隔，间隔在最坏情况下长达中断等待时间。大部分应用不受字符间的间隔影响，但一些特定应用实例如 Modebus 要求控制字符间的间隔。因此，我们不建议高数据速率时使用有奇偶校验的 Modbus。

其他添加 1 个波特时间延迟的方法列于下。

- 使用另一个串行端口作为定时器。禁止发送端口上的中断，并在装载数据寄存器的同时，在另一个串行端口里装载一个哑字符和一个第 9 位。辅助端口里的中断在 11 个而不是 10 个波特时间后发生，这样保证停止位有充足的时间。
- 发送一个满哑字符，以创建一个很长的停止位。如果要避免长停止位，波特定时器可以在哑字符被发送以减少额外停止位的长度时被加速。定时器 A4-A7 的同步性质允许分频比根据意愿增加或减少，而不会产生不规则时钟脉冲。
- 使用一个定时器中断来产生字符间的额外的 1 波特时间的延迟。可以为同一个定时器使能中断，此定时器用来产生波特时钟，且时钟可以减速，这样一个周期与所需的延迟长度相等。
- 使用有同步能力的串行端口 A 和 B 以同步模式（输出 Tx 被禁止）发送一个字符。同步字符以比异步波特率大 8 倍的速度发送，产生一个附加的波特时间。为使这个能工作，用作输出同步时钟的引脚（端口 B 的位 0 或 1）必须未连接或连接到一个能承受 8 个时钟的脉冲串的设备。

12．7．6　支持第 9 位通信的协议

这个部分描述第 9 位通信的协议怎么工作。第 9 位通信协议有处理器如 8051 和 Z180 支持，支持公司包括 Cimentrics Technology 等。数据字节有额外的第 9 位，它附加于奇偶校验位通常放置的地方。从网络主机到它的一个从设备的请求组成了字节帧——第一个字节的第 9 位被设为“1”（在处理器的 Tx 引脚上观测到该信号时设置），接下来的字节的第 9 位设为“0”。第一个字节被识别为地址字节，它指明消息指向的从处理器。这样就使能了一个从处理器来寻找消息的开端，也就是第 9 位被置位的第一个字节，同时还使从处理器判断此消息是否是指向它的。如果消息指向一个特定的从处理器，从处理器就读取消息里其余的字节；不然从处理器将继续扫描寻找装有它的地址的消息开端。

一般只有网络主机所发送请求的第一个字节的第 9 位置“1”。随后的字节和从处理器答复的第 9 位为“0”。由于通信量大多数都把第 9 位置为低，故只需要为第一个字节或地址字节增加停止位。这可以在地址字节后发送一个哑字符（发送器断开）做到，也不会牺牲任何性能。

一些微处理器的串行端口有一个“唤醒”模式的操作。这种模式下，第 9 位非置“1”的字符被忽略，也不会产生中断。当检测到帧的开端时，在那个字节上发生一个中断。如果这个字节包含某从处理器的地址，“唤醒”模式关掉，这样帧里剩下的字符可以由此从处理器读取。这种方式减少了与指向其他从处理器的消息相关的开销，但它对最坏情况下的装载没有确实帮助。大多数时候，最坏情况下的计算载入是嵌入式系统的关键因素。此外，中断驱动器忽略那些不是指向这个系统的字符也很容易。由于这些原因，Rabbit 并没有实现“唤醒”模式。

第 9 位协议有一个严重的问题，就是 IBM-PC 的通用异步收发器只在使用专门驱动器时才能支持第 9 位。

12．7．7　Rabbit 专用主/从协议（Rabbit-Only Master/Slave Protocol）

如果只连接了 Rabbit 微处理器，可以在地址字节上设置第 9 位低位，其余的字节可以用通常的 8 位模式传送。这比其他第 9 位协议更有效，因为只有第一个字节需要 11 个波特时间，其余的字节的发送只要 10 个波特时间。

12．7．8 数据组帧/Modbus

一些协议，例如 Modbus，依靠数据帧的间隔来检测下一个帧的开始。第 9 位协议是检测数据帧开始的另一种方法。

Modbus 协议要求数据帧以一个最小的 3.5 字节的安静时间（quiet time）开始。接收器使用这个 3.5 字符的间隔来检测帧的开始。为了让中断服务程序检测这个间隔，建议传送哑字符，以帮助检测这个间隔。这可用下面的方式做到。当接收到第一个字符时，发送器开始传送哑字符。每次都有一个中断，或者是接收器数据寄存器满，或者是发送器数据寄存器空。如果发送器数据寄存器是空的，传送一个哑字符。虽然接收器和发送器以近似相同的波特率工作，它们的波特率仍有最多达 5%的偏差。这样，接收器满和发送器空的中断会变的有相位差，假定远程站以字符间无间隔的方式传送的话。如果这个计数器保持为（n），表明在帧里已经检测到了间隔；间隔的长度是（n-1）到（n）个字符。帧的开端可用（n）到 3 标记，表明间隔至少两个字符长。

13．Rabbit 从端口（Rabbit Slave Port）

当某个 Rabbit 微处理器配置成从处理器使用时，并行端口 A 和固定的某些其它数据线作为从处理器和主处理器之间的通信线使用。从处理器是一个配置为从处理器的 Rabbit。主处理器可以是 Rabbit 或任何其他处理器。作为从处理器的 Rabbit 自身又可以带从处理器。

主从处理器之间通过从端口互相通信。从端口是一个物理器件，它包括数据寄存器，数据总线和各种握手联络线。从端口是从 Rabbit 的一个部分，但逻辑上它是用来在两个处理器之间通信的独立器件。从端口的框图见图 38。

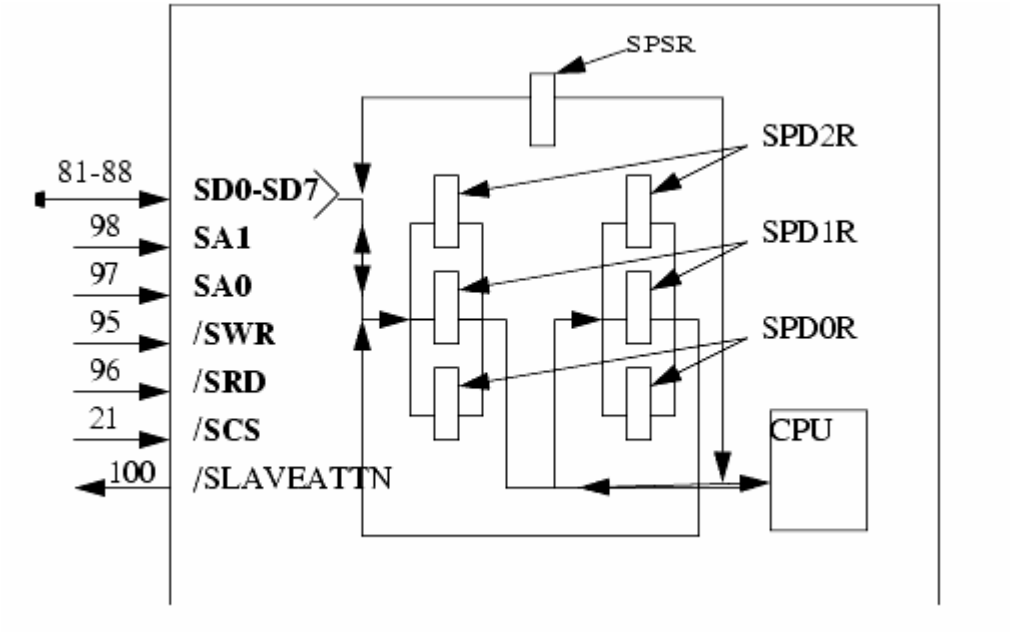


图 38 Rabbit 从端口

从端口为每个方向的通信各备有三个数据寄存器。这三个数据寄存器，分别称为 SPD0R，SPD1R，

SPD2R，可由主处理器写和由从处理器读。另三个不同的寄存器，也命名为 SPD0R，SPD1R，SPD2R，可由主处理器读和由从处理器写。不同的寄存器可以使用相同的名字，是因为通常可以从上下文中知道所指的是哪个寄存器。如果必须区分同名的寄存器，我们可以称 SPD0R 对从处理器可写或 SPD0R 对主处理器可读，同一寄存器可有不同的描述法。

状态寄存器可由从和主处理器读。状态寄存器中有为那六个寄存器服务的满/空位。数据寄存器在有能力写它的任一端写它后被认为“满”。如果同一个寄存器被任一端读后，认为它“空”。相应的寄存器的标志位在它被写后设为“1”，被读后设为“0”。

这些寄存器对从处理器表现为内部 I/O 寄存器。对主处理器，至少对 Rabbit 主处理器来说，它们表现为外部 I/O 寄存器。图 39 示出主处理器读/写从端口寄存器时的事件顺序。

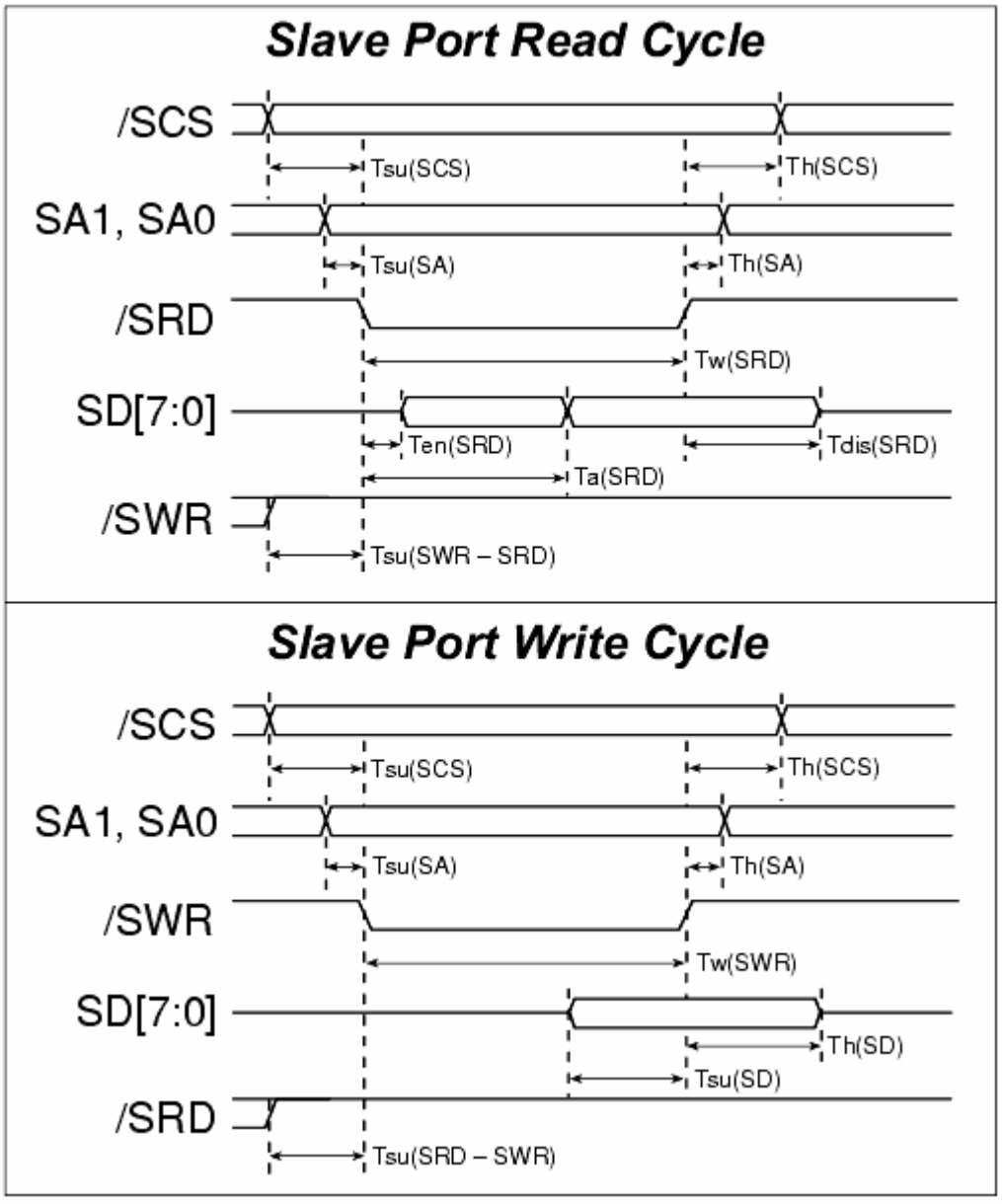


图 39　从端口 R/W 顺序

下面的表解释图 39 中使用的参数。

符号	参数	最小值	最大值
Tsu(SCS)	/SCS 建立时间	10	--
Th(SCS)	/SCS 持续时间	10	--
Tsu(SA)	SA 建立时间	10	--
Tsu(SA)	SA 建立时间	10	--
Tw(SRD)	/SRD 为低的脉冲宽度	120	--
Ten(SRD)	从/SRD 到 SRD 的使能时间	0	--
Ta(SRD)	从/SRD 到 SRD 的存取时间	--	90
Tdis(SRD)	从/SRD 到 SRD 的禁止时间	--	20
Tsu(SRW-SRD)	/SWR 为高到/SRD 变低的建立时间	120	--
Tw(SWR)	/SWR 为低的脉冲宽度	120	--
Tsu(SD)	SD 建立时间	20	--
Th(SD)	SD 持续时间	10	--
Tsu(SRD - SWR)	从/SRD 高到/SWR 低的建立时间	120	--

两个 SPD0R 寄存器具有其他数据寄存器没有的特殊功能。如果主处理器写 SPD0R，一个的中断触发器被置位。如果从端口的中断开放，从处理器将产生从端口中断。如果从处理器写另一个 SPD0R 寄存器，从注意信号线（/SLAVEATTN，100 号引脚）有效（低电平），由从处理器声明使用权。这根线可用来在主处理器里产生一个中断。被中断的任一方都可以通过写从端口的状态寄存器来清除中断请求信号。这样数据被忽略，中断源的触发器也被清除。图 40 示出这个功能的一个逻辑示意图。

图 41 示出两个从 Rabbit 连接到一个主 Rabbit 的例子。主处理器用口线为两个从处理器复位，由主处理器的时钟为从处理器提供时钟。虽然主和从处理器不一定共用同一时钟，但这样做也使从处理器没有连接晶振必要。然而，从处理器不需要非易失性内存，因为主处理器可以通过从端口冷启动它们并下载程序给它们。为了这么做，SMODE0 和 SMODE1 引脚必须按图 41 正确地进行配置，目的是在从处理器复位后开始一个冷启动过程。

从端口线示于图 38。它们的功能描述如下。

- SD0-SD7——双向数据线，一般连接到主处理器的数据总线。多个从处理器可以连接到这个数据总线上。从处理器只在/SCS 和/SRD 信号同时拉低时占用数据线。
- SA1，SA0——地址线，用来选择从处理器接口的四个数据寄存器中的一个。一般连接到主处理器的低位地址线上。主处理器始终控制这些信号。
- /SCS——输入。从芯片的选择。如果片选信号无效，从处理器忽略读或写请求。如果主处理器是 Rabbit，这根线可以连接到主处理器的可编程的片选线/I0-/I7 中一根。
- /SRD——输入。如果/SCS 为低，这根线被拉低使地址线所选择的寄存器的内容被放置到总线上。如果主处理器是 Rabbit，这根线一般连接到通用 I/O 读选通引脚/IORD。
- /SWR——输入。如果/SCS 为低，这根线能够将数据总线上的数据锁存进入地址线所选择的寄存器中；寄存器的选择在/SWR 和/SCS 中之前就有效。如果主处理器是 Rabbit，这根线一般连接到通用 I/O 写选通引脚/IOWR。

- **/SLAVEATTN——如果从处理器写 SPD0R 寄存器，这根线置为低电平，声明从处理器中有可用的数据。如果主处理器写从状态寄存器，这根线变为高电平。一般，连接这根线的目的是在它变低时向主处理器申请中断。**

从端口的数据线与从处理器并行端口 A 共用，两者使用相同的封装引脚。选中从端口，则并行端口被禁止，这可通过向从端口控制寄存器（SCR）里写入一个合适的代码实现。处理器复位后，口 A 被置为并行端口输入，除非（SMODE0，SMODE1）设置为（0，1），这种情况下复位后口 A 被设置为从端口，而且从处理器用从端口开始冷启动过程。

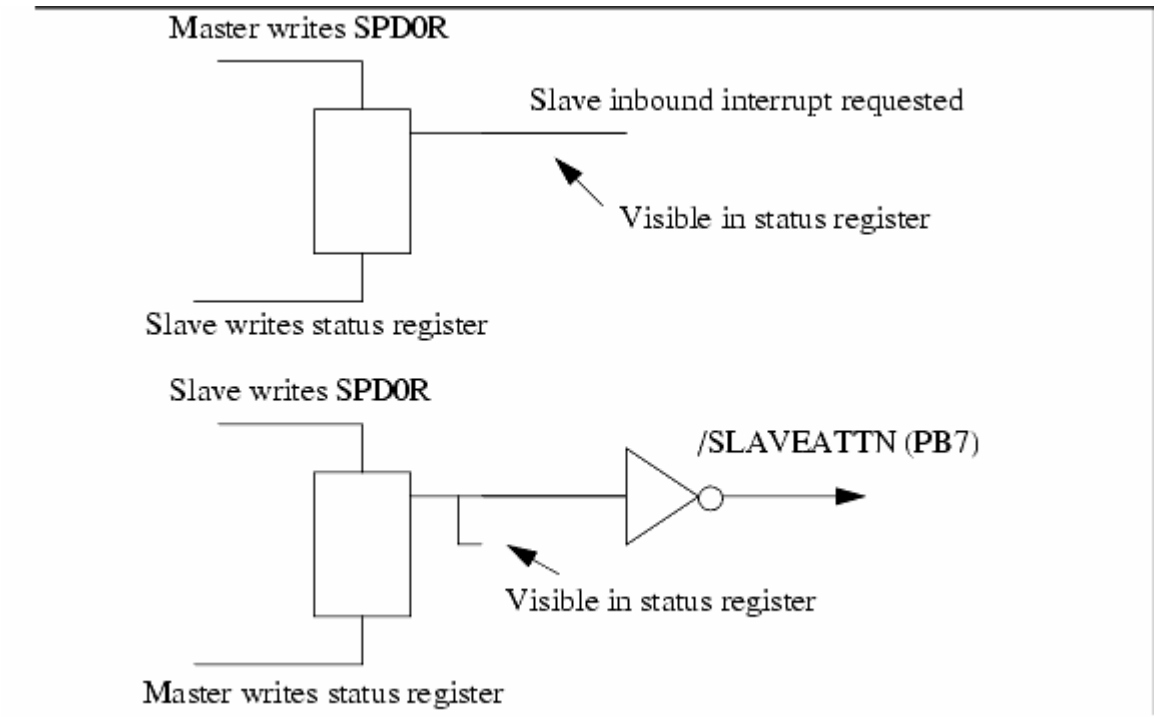


图 40 从端口的握手和中断

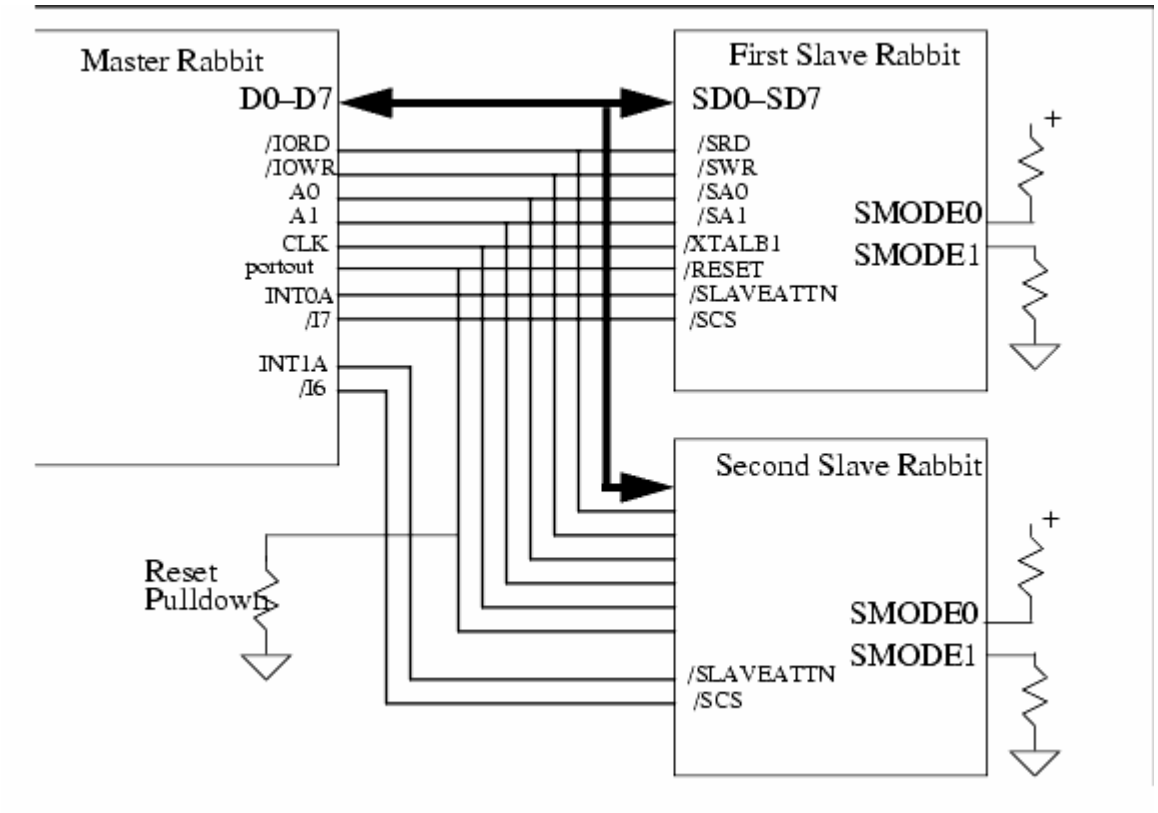


图 41 从 Rabbit 与主 Rabbit 的典型连接

13．1 从处理器互连的硬件设计（Hardware Design of Slave Interconnection）

图 41 表示了连接两个从 Rabbit 到一个主 Rabbit 的典型电路图。设计者能够冷启动从处理器和每次冷启动时下载程序到 RAM。另一个设计是同时用 RAM 和闪存配置从处理器。这种情况下，从处理器将只能下载程序以维护或升级。通常，每次启动时不应写闪存，因为闪存写操作有次数的限制。图 41 所示的从处理器的复位是在主处理器的程序控制下完成的。如果主处理器复位，从处理器也将被复位，因为对主处理器的口线在复位时处于三态，由下拉电阻把从处理器的复位信号拉低。如果主处理器崩溃且有看门狗超时的话，它会迫使从处理器崩溃，这可能是用户不希望出现的情况。

13．2 从端口寄存器

从端口寄存器列于表 44。实际上每个寄存器是两个分离的寄存器，一个用于写，一个用于读。它们可由从处理器在表中所示的 I/O 地址访问，可由主处理器在所示的外部地址访问，这个地址指明主处理器读或写寄存器时，输入到从处理器的从地址（SA0，SA1）的值。从处理器可写的寄存器只能由主处理器读，反之亦然。如果有一方在同一时刻试图读另一方正在写入寄存器，读的结果会很混乱。然而，协议和通信中的握手信号的将避免发生这种事情。

表 44 从端口寄存器

寄存器	助记符	内部地址	外部地址
从端口数据 x 寄存器	SPD0R	20h	0
	SPD1R	21h	1
	SPD2R	22h	2
从端口状态寄存器	SPSR	23h	3
从端口控制寄存器	SPCR	24h	N.A.

如果由于某些原因用户想放弃所建议的协议，查询并等待另一端写这个寄存器时，用户应该清楚所有位不会在准确的结果改变时间改变，并从一些位改变成新值而另一些没有改变的寄存器处读入一个过渡值。过渡值只能存在于一次读的时间内，各位仅是在某时间点上使其旧值改变到新值，而不会持续摇摆不定。为了避免读到过渡值，用户可以读两次寄存器，在接受这个值之前确定两个值相同，或者用户可以只检验某个位看是否改变。过渡值存在的情况很稀少，且有产生 bug 的潜在危险，这个 bug 发生时足够严重，因它很不常见，所以诊断也很困难。因此，警告用户请避免这种情况产生。

图 45 描述从端口控制寄存器。

图 45 从端口控制寄存器（SPCR）（adr=024h）

位 7 w/o	位 6,5 R/O	位 4	位 3,2 w/o	位 1,0 w/o
0--遵循 SMODE 引脚 1--忽略 SMODE 引脚	读 SMODE 引脚 smode1,smode0	x	00—禁止从端口，端口 A 是一个字节宽的输入端口 01—禁止从端口，端口 A 是一个字节宽的输出端口 1x—使能从端口	00—无从端口中断 pp—使能从端口中断，级别 1-3

各位的功能如下:

位 7——如果设为“ 0 ”,冷启动特性被使能。冷启动完成后它一般设为“ 1 ”。如果（SMODE1，SMODE0）引脚在复位结束后设为（0，1），从端口的冷启动被自动使能。这个特性禁止了处理器的普通操作，并致使通过从端口寄存器 SPD0R 接受命令。这些命令导致数据存储在内存或 I/O 空间里。当管理冷启动的主处理器已经完成了内存和 I/O 空间的设置，（SMODE1，SMODE0）引脚改变为（0，0），这导致从零地址开始执行程序。典型情况下，这将开始一个辅助启动程序的执行。在某个点上，位 7 被设为 “ 1 ”，这样 SMODEx 引脚可以用作普通输入引脚。

位 6，5——可用来读输入引脚 SMODE1，SMODE0。

位 3，2——被设为 “ 1 ” 时，位 3 使能从端口，并禁止并行端口 A 和各种其他端口线。如果通过从端口完成了一次冷启动，位 3 自动设为“ 1 ”。如果位 3 是“ 0 ”，则位 2 控制并行端口 A 是输入（位 2=0）还是输出（位 2=1）。

位 1，0—这个两位域设置从端口中断的优先级。值（0，0）禁止中断。

表 46 描述从端口状态寄存器。如果某特定寄存器满，状态寄存器有六个位被置位。这意味着寄存器已经被有能力写它的处理器写过，但还没有被有能力读它的处理器读。寄存器中相应于 SPD0R 的位用来控制从端口中断和握手信号，如图 40 所示。

表 46 从端口状态寄存器（SPSR）(adr=023h)

位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
1--主处理器写 SPD0R 时置位。从处理器写 SPSR 时清零	1—主处理器写 SPD2R 时置位。从处理器读寄存器时清零	1—主处理器写 SPD1R 时置位。从处理器读寄存器时清零。	1—主处理器写 SPD0R 时置位。从处理器读寄存器时清零。	1—从处理器写 SPD0R 时置位。主处理器读寄存器时清零。	1—从处理器写 SPD2R 时置位。主处理器读寄存器时清零。	1—从处理器写 SPD1R 时置位。主处理器读寄存器时清零。	1—从处理器写 SPD0R 时置位。主处理器读寄存器时清零。

13．3 从处理器的应用程序和通信协议

从端口使用的通信协议决定于应用程序。从处理器的使用可能有多种原因。一些可能的应用程序列于下。

记住 Rabbit 也可以通过串行端口作从处理器使用，且一些协议通过串行通信连接可以很好的工作。使用串行连接时，协议变的更复杂，如果需要考虑传送中的错误的话。如果可以控制物理链接而使传送错误不发生——如果控制了互连环境这在实际中是可能的，串行协议就算把错误连接考虑进来也更简单和迅速。

13．3．1 从应用程序（Slave Application）

- 动作控制器——许多种类的动作控制要求快速响应，它也可能是计算强化（compute-intensive）的，或者两者都有。传统的伺服系统解决方案太昂贵，或由于系统的非线行而不能很好的工作。动作控制器的基本通信模型是为主处理器发送短信息即定位指令给从处理器。从处理器确认这些命令并执行，同时报告异常情况。
- 通信协议处理器——通信协议可能会很复杂，要求快速响应，也可能是计算强化的。
- 图形控制器——Rabbit 可执行诸如画几何体和创建字符等操作。
- 数字信号处理——虽然 Rabbit 不是专业数字信号处理器，但它有足够的运算速度来处理一些可能要求专业处理器的工作。从处理器可以处理数据以执行模式识别，或从数据流中析取具体参数。

13．3．2 主-从消息传递协议（Master-Slave Messaging Protocol）

此协议中主处理器发送消息给从处理器，并接收一个确认消息。此协议可由查询或中断驱动。一般情况下，主处理器发送带有消息类型码的消息，可能是字节计数（byte count），也可能是文本消息。从处理器以一条类似的消息作为答复。除非主处理器发送消息，不会发生任何事情。不允许从处理器启动一条消息，但从处理器可以发信号给主处理器，方法是用并行端口线（而不是/SLAVEATN），或把数据放置在主处理器能不妨碍消息协议就可读取的寄存器里。

主处理器通过把消息字节存入 SPD0R 而发送一条消息。从处理器注意到 SPD0R 满，然后读取这个字节。当主处理器发现由于从处理器读 SPD0R 而使它变空时，把下一个字节存入 SPD0R。任何一方可以通过读状态寄存器来判断哪些寄存器是满或空。当从处理器以一条回答消息确认了消息，处理操作反向。若要用中断来执行协议，可在每次从处理器接收到一个字符时产生一个从端口中断。从处理器通过读 SPD0R 来给主处理器一个确认，只要主处理器正查询从处理器对每个字符的响应。如果从处理器以中断主处理器方式来确认每个字符，从处理器可以在 SPD0R 里存入一个哑字符来产生一个主处理器中断，这时假定/SLAVEATTN 线被连接以中断主处理器。确认消息以相似的方式工作，除了主

处理器写一个哑字符来中断处理器并告诉从处理器它有字符。

如果每个发送的字符有双重中断，一些问题会出现。其中一种是消息传送率将在由中断执行时间和处理器运算速度所限制的速度范围内不确定。这会占用一个或全部两个处理器的很高的计算机资源，使其他进程尤其是中断程序缺乏时间。如果这已经成为一个问题，可以用一个定时中断在某端驱动处理器，这样就限制了数据传送率。

另一种解决方案，可能比限制传送率更好。就是仅在消息的第一个字节上在从处理器端使用中断，然后降低中断级别，并把剩下的事务作为查询事务处理。在主处理器端，整个事务都作为查询事务。这种情况下，整个事务在从处理器上在中断程序里发生，但其他中断并没有被禁止，因为降低了中断级别。

典型的从系统由一个 Rabbit 微处理器和连接到它的 RAM 内存组成。时钟的提供，可连接晶振，或外部时钟提供，后者还可作为主处理器时钟。复位线一般由主处理器驱动。系统启动时间里，主处理器复位从处理器，并通过从端口冷启动它（必须适当配置 SMODE 引脚）。一旦软件载入从处理器，从处理器就可以执行它的功能了。

作为一个简单例子，假设从处理器用作一个四端口 UART。它有在它的四个串行端口上发送或接收字符的能力。不考虑参数配置问题如波特率，我们可以如下定义一个协议。

- 主处理器可读 SPD0R 是一个状态寄存器，它的位表明这四个接收器和四个发送器里哪个就绪，也就是说，已经接收了一个字符或准备发送一个字符。
- 主处理器可写 SPD0R 是一个控制寄存器，用来给从处理器发送命令。
- SPD1R 用来发送或接收数据字符或控制命令。
- /SLAVEATTN 线连接到主处理器的外部中断请求处，这样从处理器写 SPD0R 时，主处理器被中断。典型情况下，当串行端口中的任一个的状态发生变化时，从处理器都会写 SPD0R。

从处理器可以在任何时候通过存 SPD0R 中断主处理器。每次一个使能的发送器准备接受字符和每次一个使能的接收器接收字符，它都这么做。当它存 SPD0R 时，所存入的代码表明中断的原因，也就是说，接收或发送以及通道号。如果原因是接收，所接收的字符还会被放置在从处理器可写的 SPD1R 里。主处理器由于任何原因被中断时，主处理器会通过读 SPSR 暗中进行一次对 SPD0R 的取数。如果中断由接收一个字符引起，它会从 SPD1R 里移走此字符，读 SPD0R 来与从处理器进行握手联络。

如果主处理器由于发送器就绪而被中断（这通过暗中取数来判断），它把要传出的字符放置在 SPD1R 里，向 SPD0R 写一个表明传送和通道号的代码。这将导致从处理器被中断，且从处理器将采用这个字符并读 SPD0R 而与主处理器握手。这个握手信号不中断主处理器。

14．Rabbit 硬件设计和发展（Rabbit Hardware Design and Development）

可围绕 Rabbit 2000 微处理器进行核心设计。一个核心设计包括内存，微处理器，晶振，Rabbit 标准编程端口和一些情况下的电源控制器及电源。虽然现代设计通常使用至少四层的印刷电路板，两端电路板对于 Rabbit 也是可行的选择，尤其如果时钟不高且此设计打算在 2.5V 或 3.3V 上工作时，因为这些因素减少了脉冲沿的速度和电磁辐射。

解释 Rabbit 微处理器使用的原理图可在 Rabbit 半导体公司的 Web 站点上获得。

14．1　RS-485 通信接口
将来增加。

14．2　RS-232 通信接口
将来增加。

14．4　数字-模拟转换器
将来增加。

14．5　高电压驱动器
将来增加。

14．6 时钟
Rabbit 有两个内置振荡器。32.768kHz 时钟振荡器在电池供电的时钟、看门狗定时器和冷启动功能处需要。主振荡器为微处理器提供运行时钟。

上电后 32.768kHz 振荡器开始振荡的过程较慢。由于这个原因，BIOS 里有一个等待循环，它等待直到振荡器规则振荡了，才继续下面的启动过程。如果时钟是电池供电的，没有启动延迟，因为振荡器早已正常振荡了。启动延迟可达 5 秒。具有低串联阻抗（R<35k Ω）的晶体的启动要快些。要求的振荡器电路示于图 42。如果实时时钟的电流消耗很重要，下图所示的调节器电路会在使用 3V 锂电池时大大减少电流消耗量。使用这个电路，电池供电时钟需要的电流小于 25μA。如果这 3V 被充分使用，电流消耗大约是 70μA。

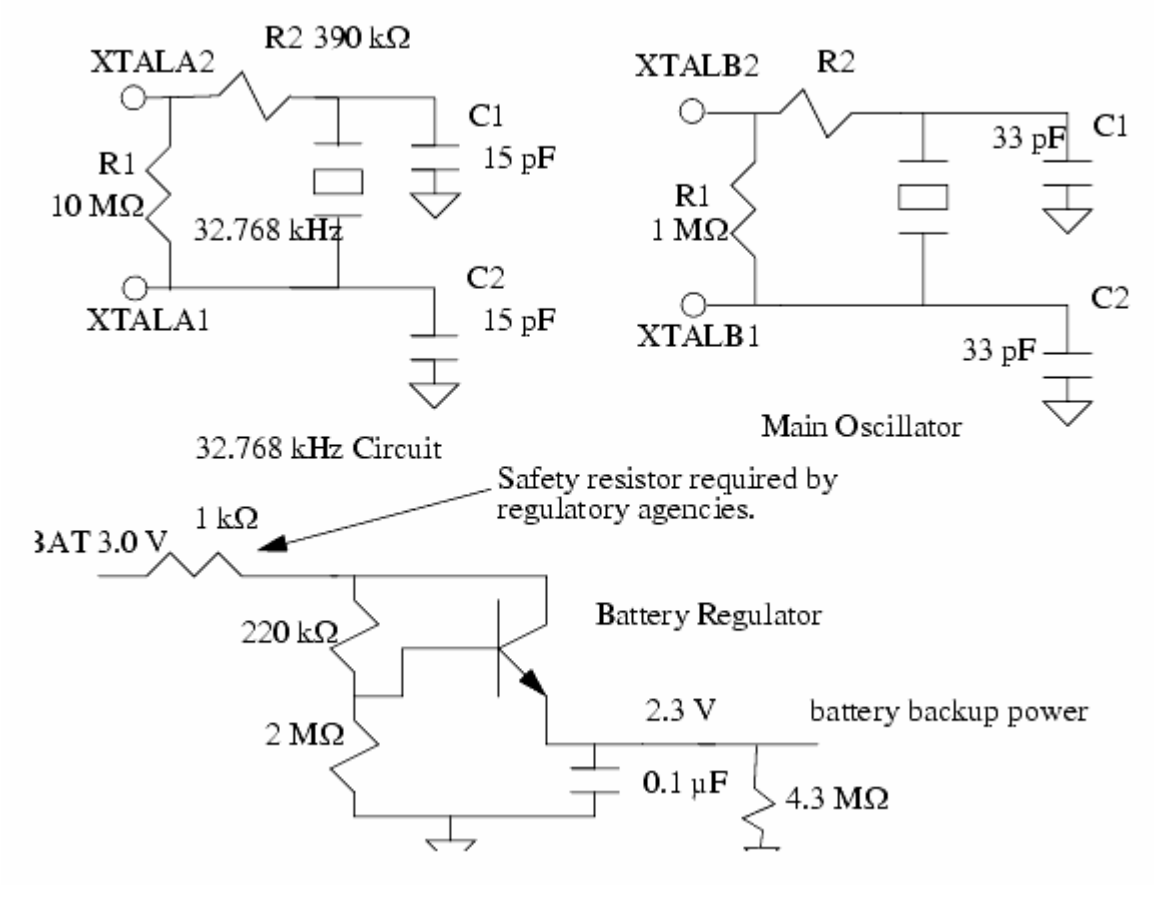


图 42　振荡器电路

14．7 低功率设计（Low-Power Design）

功耗与时钟频率和工作电压的平方成正比。这样，工作于 3.3V 而不是 5V，其功耗只是后者的 10.9/25 或 43%。低工作电压时，时钟速度与电压成正比例的减少。因此，3.3V 上的时钟速度是 5V 上的大约 2/3。工作电流与工作电压成比例的减少。

Rabbit 没有其他一些微处理器具有的“待机”状态，而是具有把它的时钟切换到 32.768kHz 振荡器的性能。这被称为睡眠状态。如果这么做了，功耗极大的减少。电流消耗在这个时钟速度上通常减少到 100µA 的范围。在这么低的时钟速度下，Rabbit 每微秒执行 6 条指令。一般来说，速度减小到这个程度时，Rabbit 处于一个严格轮询循环，等待一个事件把它唤醒。增加时钟速度以唤醒 Rabbit。

14．8 基本内存设计（Basic Memory Design）

一般，/CS0、/OE0 和 WE0 应该连接到保存着从零地址开始执行的启动代码的闪存。当处理器存在复位且（SMODE0，SMODE1）设为（0，0），它将试图从连接到/CS0、/OE0 和/WE0 的内存的开端部分开始执行指令。

根据惯例，基本 RAM 内存应该连接到/CS1、/OE1 和/WE1。/CS1 有一个特殊性质，使它成为电池供电的 RAM 的优选片选线。可以把 MMIDR 寄存器里的一个位置位，促使/CS1 保持使能（低）（参看 8.3.1 节的表 23）。可用这种能力来解决当片选线穿过一个器件表面时遇到的问题，此器件用来在电源切换到电池供电时通过抬升/CS1 使内存芯片处于待机状态。电池切换设备通常有传播延迟，大概 20ns 或更长。这足够满足某些情况下为访问 RAM 插入等待状态的要求。通过迫使/CS1 为低，传播延迟不会成为一个重要因素，因为 RAM 一直被选中，并由/OE1 和/WE1 控制。这么做 RAM 比处于电池供

电时消耗更多电能，如果它之前运行时采用动态片选和等待状态。如果使用这个特殊性能来加速电池供电的 RAM 的访问时间，不可以有其他内存芯片连接到 OE1 和 WE1。

14．8．1 内存访问时间（Memory Access Time）

要求的内存访问时间决定于时钟速度及地址和数据线的电容性负载。可编程以指定等待状态，以在给定的时钟速度上适应低速内存。保存程序的内存应该避免等待状态，因为这样执行速度会极大的降低。等待状态在指令内存里的重要性要比在数据内存里大的多，因为绝大部分内存访问都是取指令。从 0 个到 1 个等待状态，相当于使时钟速度减少了 30%。从 0 个到 2 个等待状态，相当于减少时钟速度 45%。不同时钟速度要求的内存访问时间表示于第 15 章的表 48。

14．8．2 为未编程的闪存的预防措施（Precautions for Unprogrammed Memory）

如果不是处于某种冷启动模式，当一个基于 Rabbit 的系统上电并脱离复位，处理器试图开始执行启动，方式是从连接到/CS0、/OE0 和/WE0 的内存的 0 地址进行读操作。如果这个内存是一个未编程或不正确编程的闪存，内存会有烧毁的危险，如果闪存的写安全特性被禁止。闪存有一个写安全特性，它禁止开始写周期，除非首先存入内存一个特殊代码。举个例子，Atmel 闪存使用字节 AAh，55h 和 A0h 以特定序列存入地址 AAAAh 或 5555h。任何没有这个序列做前缀的写操作将被忽略。如果内存禁止了写保护，且开始了一个操作，有可能建立一个包括写内存操作的无限循环。由于闪存在几百次写操作之后会损坏，内存会在一个很短时间内由于这个循环而被破坏。不幸的是，闪存从工厂里运过来时，是让写保护特性禁止的，为的是适应已过时的编程器。

解决这个问题的方法，是订购内存时要求写保护使能，或者用闪存编程系统使能它。这样，内存结合在 Rabbit 系统里才会安全。如果这么做不方便，可使用检验软件在编程连接串行链路上发送一个字节序列使内存安全。

下面的例子示出一个程序可使 Atmel AT29010a 128K*8 闪存安全，它可通过冷启动协议下载。这时，连接到/CS1 的 RAM 用来保存一个从零地址开始的程序。闪存被映射入从 1000h 地址开始的数据段，为的是访问闪存的开始部分。

```
; 存储这段程序之前 RAM 被映射到地址空间第一象限
; 此段程序驻留在 RAM 的 0 地址上
; 注意：此段程序没有经过测试
ld a,0e1h ; 3e e1  段尺寸寄存器
ioi ld (13h),a  ; d3 32 13 00  数据段从 1000h 开始
ld a,3fh ; 3e 3f  数据段寄存器
ioi ld(12h),a ; d3 32 12 00  设置闪存数据段基址为 1000h
ld a,0  ; MB1CR 即 CS0 上闪存的存储体寄存器的值是 3e 00
ld (15h),a ; 32 15 00  存储体 1 读从 256K 上开始的闪存
ld a,0aah ; 3e aa
ld (5555h+1000h),a ; 32 55 65  解锁码的第 1 字节
ld a,55h  ; 3e 55
ld (2AAAh+1000h),a  ; 32 aa 3a  解锁码的第 2 字节
ld a,0a0h ; 3e a0
ld (5555h+1000h),a ; 32 55 65  解锁码的第 3 字节
ld hl,1000h  ;21 00 10  指向闪存的开始
```

```
ld (hl),0c3h ; 36 c3  跳到操作码
inc hl ; 23ld (hl),00h  ; 36 00  零
inc hl ; 23
ld (hl),00h ; 36 00  零
jr *  ; 18  无限循环的结束
```

下面的代码可从串行端口以三个一组的方式发送。

```
80 14 01    ; I/O 向 0000 写入 01， MB0CR 选择 CS1- 把 RAM 映射到第一象限
00 00 3e  ;  写内存地址 0
00 01 e1
00 02 d3
00 03 32
00 04 12
00 05 00
;  接以上代码
00 2b 18 ;最后一条指令
00 2c fe  ;  最后一个字节
80 24 80  ;  从 0 地址开始执行程序
```

整个程序大约执行 10ms。

14．9　PC 主板规划和内存线排列（PC Board Layout and Memory Line Permutation）

为了有效使用“不动产”PC 主板，建议连接到内存的地址和数据线改变排列，以最小限度的使用 PC 主板资源。通过重新排列这些线，减少了 PC 板上的线交叉覆盖的必要，节省了通路和空间。

对静态 RAM，地址和数据线可以任意布置，意思是来自处理器的地址线可以用任何顺序连接到 RAM 的地址线。这同样适用于数据线。例如，如果 RAM 有 15 根地址线和 8 根数据线，如果处理器的 A15 连接到 RAM 的 A8，不会产生什么不同。反之亦然。类似地，处理器的 D8 可连接到 RAM 的 D3。唯一的限制是，处理器的所有 8 根数据线必须连接到 RAM 的 8 根数据线。当同一 PC 主板区域能适应几种不同类型的 RAM 时，由于安装了一个更小的 RAM 而未使用的高位地址线必须保持有序。例如，如果同一区域可以接受一个 128K*8 RAM（17 地址线），或一个 512K*8 RAM（19 地址线），那么 A18 和 A19 可以彼此互换，但不可以同 A0-A17 互换。

同样，布线对闪存没有什么不同。如果内存要插入插槽，且计划在板外编程内存，保持数据和地址线的自然顺序很可能是最好的。然而，由于闪存可以使用 Rabbit 编程端口在电路里编程，我们希望大部分设计者把闪存以未编程状态焊接入主板。这时，数据和地址线的相互交叉因素必须考虑进来，因为闪存要求使用特殊解锁码来移去写保护。解锁操作涉及特殊序列的读和写——访问特殊地址和写解锁码。

另一个必须考虑的因素，是闪存可能被分成扇区。为了改变内存某地址的内容，至少必须写一个整的扇区。在分成小扇区的内存里，内存分成 1024 个扇区。如果在某特定设计里可用的闪存是 512K，则最大的扇区是 512 字节。如果可用的最小内存是 128K，则最小扇区是 128 字节。为使所有可能类型的内存的扇区相连，低 7 位地址线（A0-A6）必须作为一个组排列。A7 和 A8 根本不要排列，如果可

以保持更大的扇区相连的话。高 10 位地址线可作为一个分离的组排列。特殊的内存芯片地址 05555h 和 0AAAAh 必须作为解锁序列的一部分来访问。这些地址只使用前面的 16 根地址线，且对奇数和偶数号线相同方式使用。解锁码使用数字 55h，AAh 或 A0h。

在实际系统里应该避免排列闪存的数据和地址线。

15．AC 时序规范（AC Timing Specification）

Rabbit2000 处理器可以工作在 2.5V 和 5.5V 之间，温度范围-40℃到 80℃，也有可能在范围-55℃到 120℃上工作。大多数使用者在 5.0V 或 3.3V 上使用 Rabbit。每瓦特的最大计算量在大约 3.3V 上得到。最高的实际速度一般在 5V 上得到。

有一种型号的 Rabbit—R30，它有最高时钟速度 29.4MHz 和工业温度范围-40℃至+85℃。R30 在 3.3V ±10%上的最高时钟速度是 18.9MHz。2.5V 上的最高时钟速度是 11.5MHz。

如果使用一个半速晶体和时钟倍频器以得到期望的时钟速度，最高时钟速度必须减小 40%来允许振荡器产生的波形的最高 4%的非对称度。这是因为时钟倍频器使用脉冲沿中点产生双倍频率。如果使用时钟倍频器来使 14.7456MHz 加倍到 29.4912MHz，工作温度必须限制到 70℃。

为了优化功耗，一般的策略是使用供电电压在 3V 到 3.5V 之间，并且把时钟速度尽可能可行地向下调。这将使每瓦特功率有最大计算量。

表 47　Rabbit 基本的最坏情况下的时序

	2.50V min. -40℃-+85℃	3.3V±10% -40℃-+85℃	3.3V±5% -40℃-+70℃	5.0V±10% -40℃-+85℃	5.0V±10% -40℃-+70℃
最高时钟速度	11.5MHz	17.5MHz	19.25MHz	29.5MHz	31.5MHz
使用时钟倍频器时产生的最高时钟速度	11.06MHz	16.75MHz	18.5MHz	28.5MHz	30.0MHz
带 20pF 地址线负载时的寻址输出延迟时间	15ns	11 ns	10 ns	8 ns	7 ns
带 70pF 地址线负载时的寻址输出延迟时间	27 ns	21 ns	19 ns	15 ns	14 ns
建立时间（T setup）	4 ns	4 ns	3 ns	3 ns	2 ns
					2000.08.01

表 47 里的工业时钟速度值（在最高温度 85℃上），比 70℃（这里扩展到了-40℃）时的商业额定值提高了 7%。温度单独作用时，每升高 5℃，时钟速度大约减少 1.2%。最高时钟速度大约直接正比于工作电压。

如果要以标准波特率使用串行通信，则必须使用特定的时钟速度。这些时钟速度一般是 1.8432MHz 的倍数，以保证可获得 57,600bps，19,200bps 和更低的波特率。3.6862MHz 的倍数时钟可保证获得 115,200bps，38,400bps 和更低的波特率。1.2288MHz 的倍数可保证获得 38,400bps 和更低的波特率。标准 Rabbit BIOS 能接受 0.6144MHz 的倍数的时钟速度。

图 43 和图 44 表明了最高时钟速度。在这个速度上面，当电压和温度变动时，对典型的 Rabbit 2000

没有检测到故障。正式的设计规格指定了一个较低的最高频率，以允许处理器的变化因素。

印模（die）在较高时钟速度时，受自身发热的影响很大。零空气流量时，印模对周围的热阻是 44℃/W。在 5V 和消耗电流 65mA 上，这会导致 15℃的自发热，并使最高时钟速度减少大约 3%。这个减少量包括在表 48 中，它提供了内存访问时间的要求。

当对内存器件接口（interfacing）时，可直接接口的内存要求的内存访问时间由下式:给出：

（1）访问时间=（时钟周期）*（2+等待状态）-T_{setup}-T_{adr}

此处 T_{adr} 是 T_1 的上升沿到地址变有效之间的延迟，T_{setup} 是与时钟相关的数据建立时间。T_{setup} 和 T_{adr} 示于表示内存读/写和 I/O 读/写周期的图 45 和图 47。大多数 5V 内存是 TTL 兼容的，这样它们可从 0.8V 切换到 2.0V。T_{setup} 由 VDD 电压电平的 30%/70%指定。

所测量的 T_{adr} 是信号从高电平降落到 0.8V 所需的时间。T_{adr} 决定于总线负载——地址线 A0 有一个更强有力的驱动器并可以处理双倍的电容却具有不变的的延迟时间。T_{adr} 也适用于内存片选线。

这些数值也适用于内存输出使能线，除了它们比片选和地址线迟一个时钟被使能的情况之外。

方程（1）中的公式在使用了时钟倍频器时仍旧有效，除了访问时间必须每个时钟周期减少 4%（如果有奇数个等待状态）。

表 48 示出内存访问时间的要求。

一般来说，最高工作速度与电源电压成正比。工作电流与电压成正比，因此工作能耗正比于电压平方。工作能耗还正比于时钟速度。更高的温度以 1%/5℃减少最高工作速度。此外，更高的工作速度增加印模温度，原因是产生的热量和因此而稍微混合的更高温度的副作用。

图 45，图 46 和图 47 解释内存读和写周期。Rabbit 工作的每个总线周期为 2 个时钟再加任意的等待状态，这些都会稍后指明。

注意 T_{hold} 根据数据被读或写而有不同的值。读数据的 T_{hold} 指定当时钟周期重复时，数据必须在 T1 的上升沿之后保持有效多长时间。写数据的 T_{hold} 指定一旦/WEx 或 IOWR 变高数据可保持有效多长时间，必须至少 0.5 个 CPU 时钟周期。

I/O 总线周期有一个自动等待状态，因此每个总线周期需要三个时钟时间加上指定的任意数目的额外等待状态。

表 49 列出这些图中表示的参数，提供最小值或测量值。

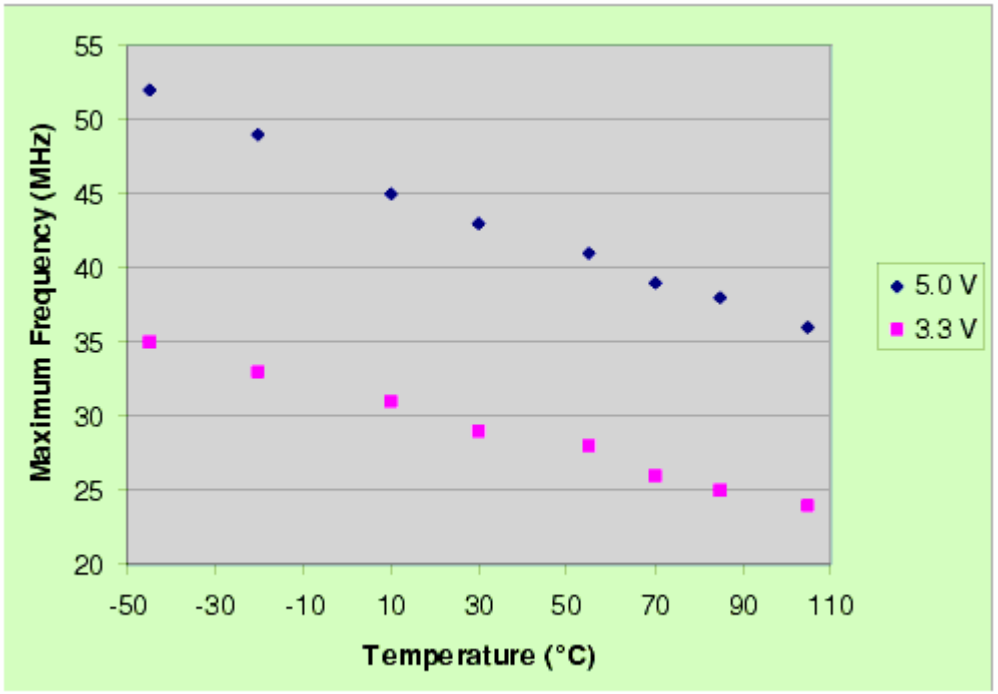


图 43 Rabbit 2000 典型最高工作频率
对比 5V 和 3.3V 上的温度

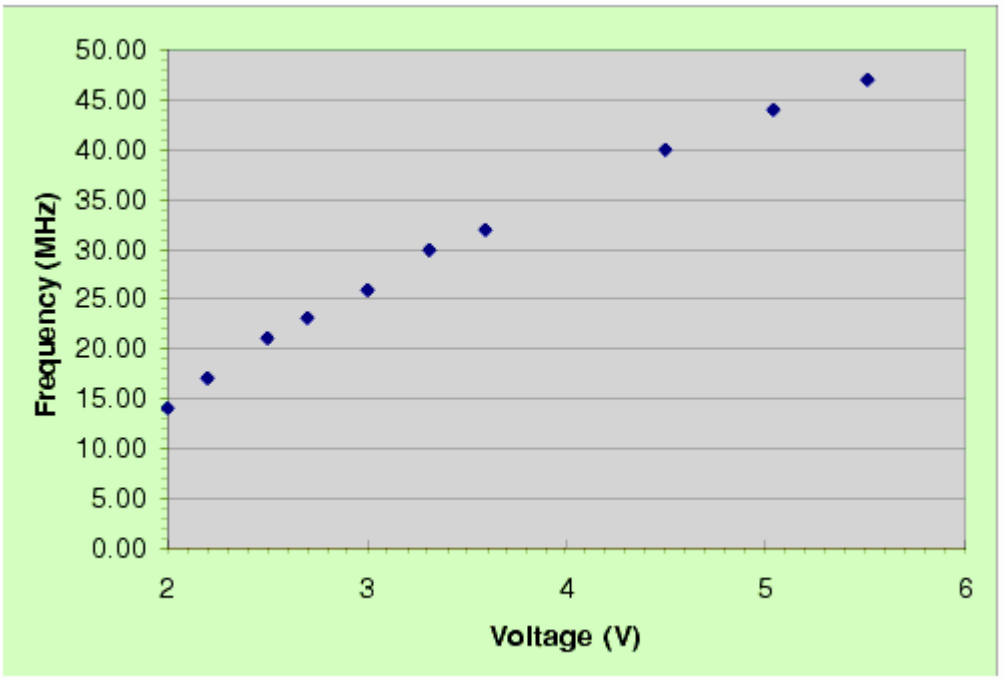


图 44 Rabbit 2000 典型最高工作频率
对比 25°C时的电压

表 48 内存访问时间要求（V±5%，T -40℃到+70℃）

时钟速度（MHz）	周期（ns）	等待状态	内存访问时间 5V 20pF 负载（ns）	内存访问时间 5V 70pF 负载（ns）	最高 PC-兼容的波特率（bps）
29.4912	34	0	59	52	921,600
27.6840	36.2	0	64	57	57,600
25.8048	38.7	0	69	62	115,200
25.8048	38.7	1	108	101	115,200
25.8048	38.7	2	147	140	115,200
24.576	40.7	0	73	66	38,400
23.9616	41.7	0	75	68	57,600
22.1184	45.2	0	82	75	230,400
22.1184	45.2	1	127	120	230,400
22.1184	45.2	2	173	165	230,400
20.2752	49.3	0	90	83	57,600
18.432	54.2	0	100 @5V 96 @3.3V	93 @5V 87 @3.3V	115,200
14.7456	67.8	0	127 @5V 123 @3.3V	120 @5V/ 114 @3.3V	460,800
14.7456	67.8	1	197 @5V/ 193 @ 3.3V	190 @5V/ 184 @3.3V	460,800
11.0592	90.5	0	172 @5V 168 @3.3V 162@2.5V(min)	165 @5V/ 159 @3.3V 150@2.5V(min)	115,200
7.3728	135.6	0	263 @5V/ 259 @3.3V 253 @2.5V(min)	256 @5V/ 250 @3.3V/ 241@2.5V(min)	230,400

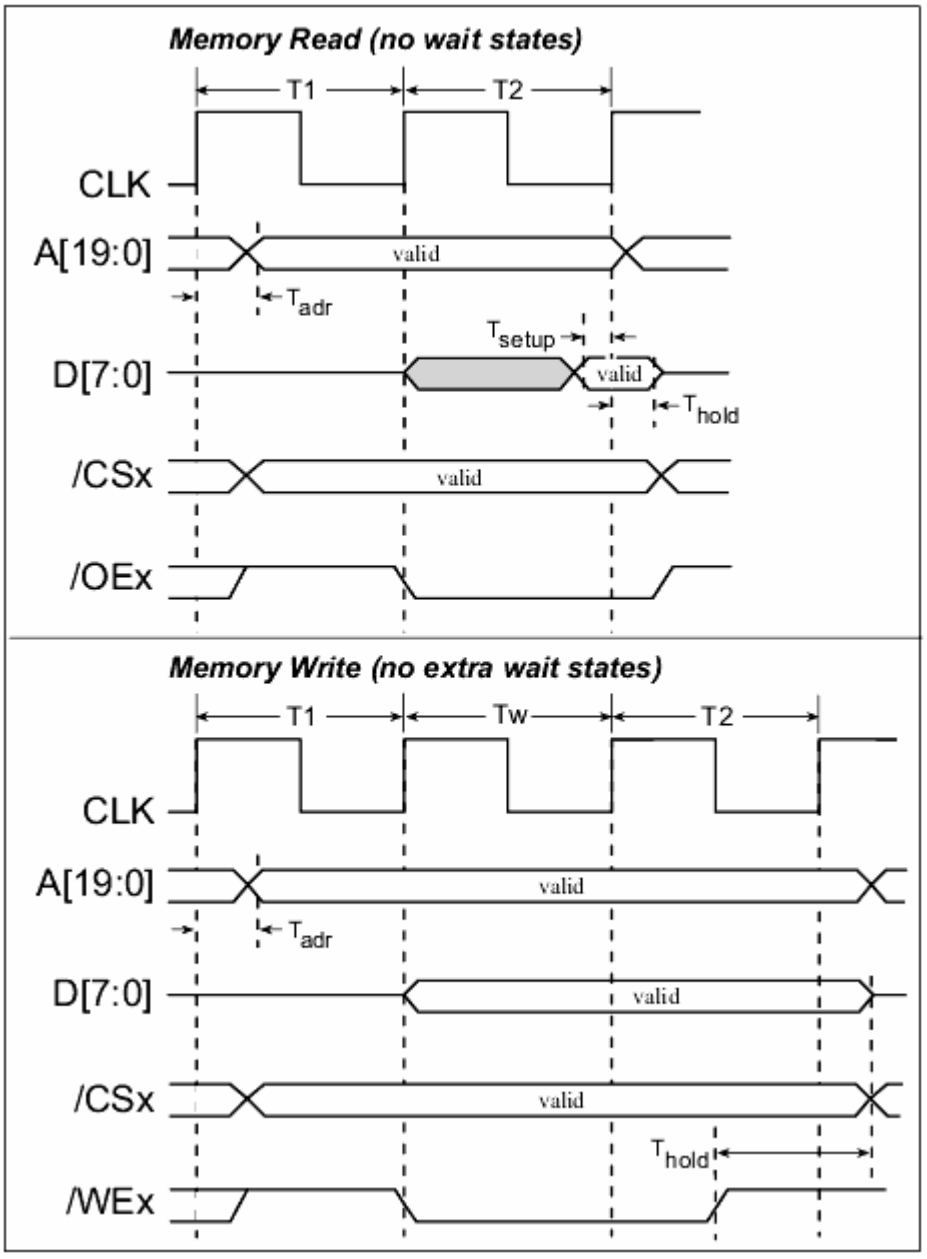

Memory Read (no wait states)
T1
T2
CLK
A[19:0]
valid
Tadr
Tsetup
D[7:0]
valid
Thold
/CSx
valid
/OEx
Memory Write (no extra wait states)
T1
Tw
T2
CLK
A[19:0]
valid
Tadr
D[7:0]
valid
/CSx
valid
Thold
/WEx

图 45 内存读和写周期

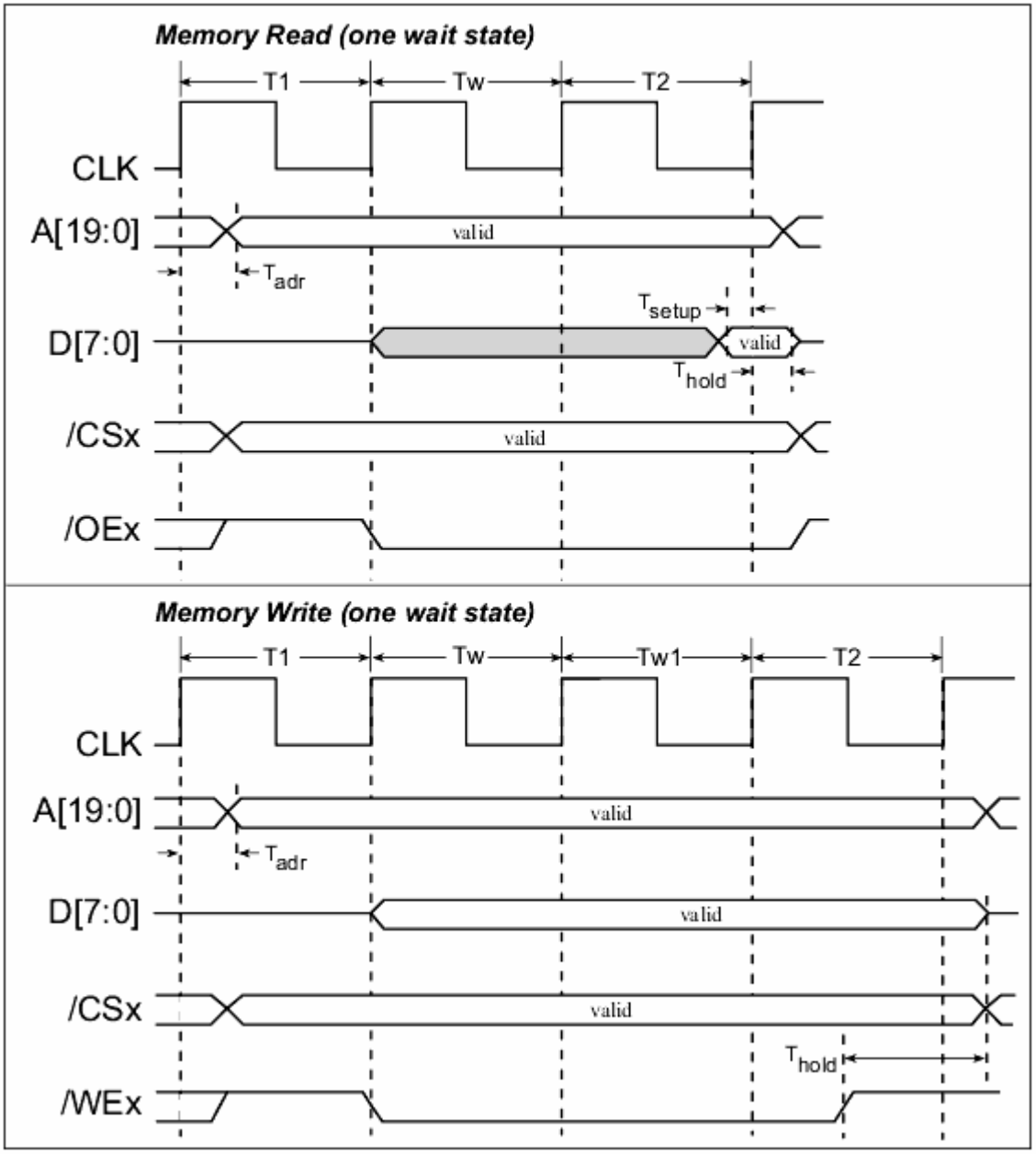


图 46 带等待状态的内存读和写操作

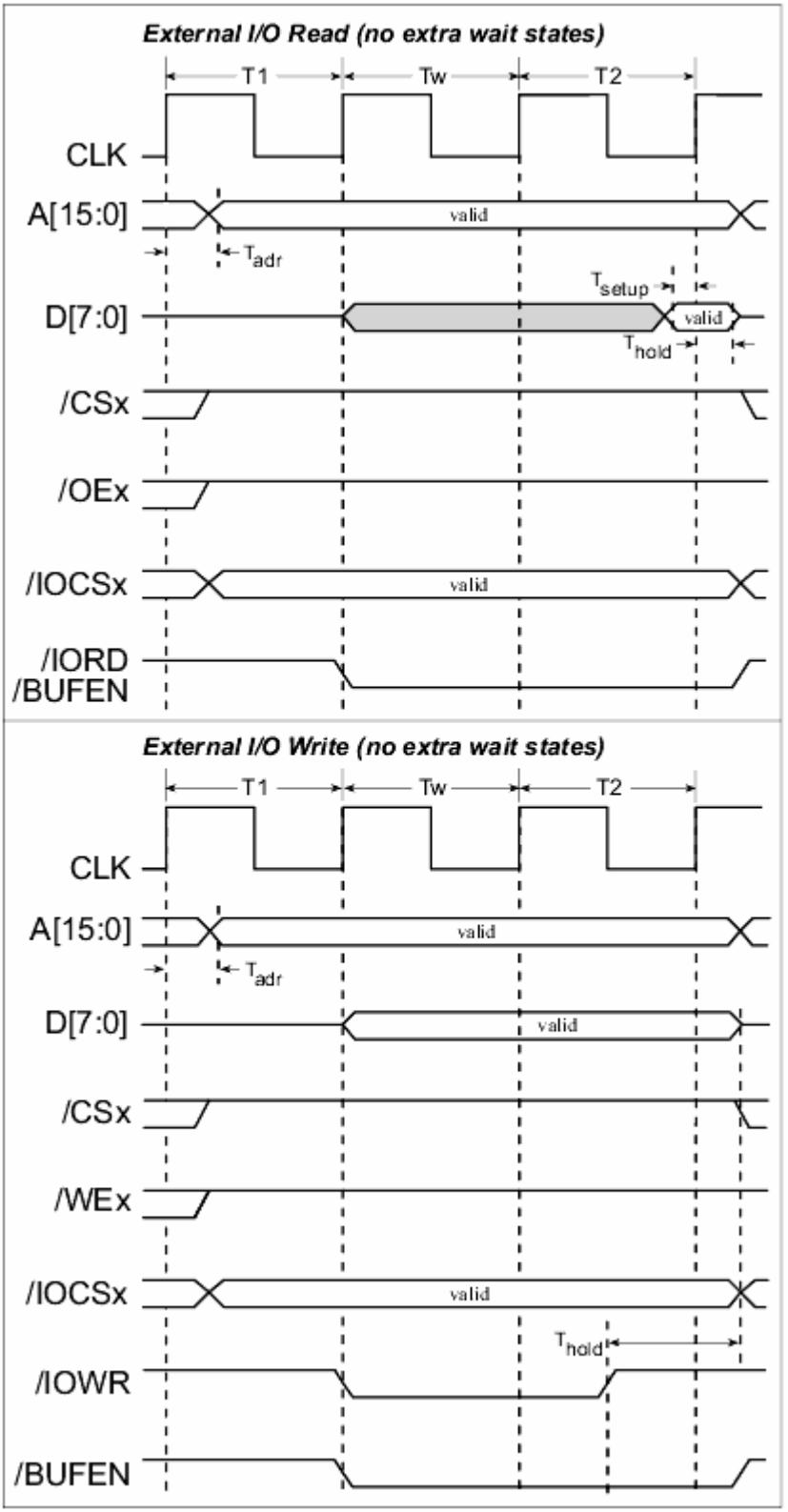


图 47 I/O 读和写周期（无额外等待状态）

表 49 内存和外部 I/O 读/写参数

参数		描述	值	
读参数	T_{adr}	从 CPU 时钟上升沿到地址有效的时间	Max.	7ns @20pF,5V(10ns @3.3V) 14ns @70pF,5V(19ns @3.3V)
	T_{setup}	读数据的建立时间	Min.	2ns @5V(3ns @3.3V)
	T_{hold}	读数据的保持时间	Min.	0ns
写参数	T_{adr}	从 CPU 时钟上升沿到地址有效的时间	Max.	7ns @20pF,5V(10ns @3.3V) 14ns @70pF,5V(19ns @3.3V)
	T_{hold}	写数据的保持时间，从/WEx 或/IOWR 变高后	Min.	1/2 CPU 时钟周期

15．1 电流消耗（Current Consumption）

典型电流与时钟频率和电压都成正比。主振荡器要求大约 6mA（5V）和 2mA（3V），它的电流独立独立于频率。处理器振荡器在 5V 和 15MHz 上专用的基本电流消耗大约是 42mA。可用下面的公式计算电流消耗：

（2）$I=（0.7）*(频率\ MHz)*(电压)+(0.35)*(电压-0.86)^2$

第一项代表处理器消耗的电流，它与电压和频率成正比。第二项是主振荡器消耗的电流，它独立于频率，但随电压平方而变动。主振荡器被禁止时，此项为 0。表 50 提供一些电流消耗的检验点。

表 50 在选择频率和电压上的典型电流（25℃）

时钟频率（MHz）	电压（V）	电流（mA）
29.4912	5	109
22.11	5	83
14.7456	5	58
14.7456	3.3	36
7.3728	3.3	19
3.6864	3.3	11
1.8432	3.3	6
0.9216	3.3	6
0.4608	3.3	3.14
0.032（睡眠模式）	5	0.280
0.032（睡眠模式）	4	0.173
0.032（睡眠模式）	3.3	0.113
0.0320（睡眠模式）	2.7	0.072

内存和系统中其他器件，包括上拉电阻、驱动负载的输出和浮动输入所消耗的电流，必须加到表 50 里的数值上。

32.768kHz 时钟振荡器和相关的实时时钟在 3V 工作时大概消耗 23μA（在 2.25V 上，如果由电池供

电，电流消耗大约是 11μA）。电源关断以及只有 32.768kHz 振荡器和时钟被供电时消耗的（典型）电流，由下式给出：

（3） 电流（μA）=5.44*（V-0.86）^{2}，

此处 V 是工作电压。这是必须由备用电池提供的电流，没有计入相关电路要求的电流。振荡器不会在低于 1.3V 下工作。导出上面公式时的测量工作，通过在 32.768kHz 振荡器电路里加一个 390kΩ串联电阻和 15pF 负载电容器进行。旁路电阻是 10MΩ。

如果处理器工作于 32.768kHz，则室温时使处理器工作所需的附加电流（主振荡器切断）如下给出：

（4）电流（μA）=7.5*（V^2）

低功耗模式下，电流消耗正比于电压的平方。3.0V 时大约是 67μA。加上使振荡器工作的 25μA，总的电流消耗大约 92μA（当处理器工作于 32.768kHz）。

RAM 或闪存在低频率时消耗的电流很重要，如果没有使用自动断电闪存或低能耗 RAM 的话。如果使用低能耗 RAM 来支持睡眠模式，睡眠模式循环必须拷贝到 RAM 并在 RAM 里执行。当试图在超低能耗睡眠模式下工作时，不能有浮动输入是很重要的。浮动输入（floating inputs）消耗基本的主要功耗。时刻注意，如果端口 D 的漏极开路输出没有拉低到零，它将造成浮动输入。上拉电阻消耗电流，且在超低能耗模式时必须避免使用或被禁止。当测试睡眠模式的工作时，最好连接一个安培表，以保证设置时没有把任何额外浮动输入或其他电流消耗部件包括进来。

16．Rabbit 软件（Rabbit Software）

这章介绍使用 Rabbit 微处理器时我们推荐的基本的低层次软件构造。如果接受这些建议，使用者会发现使用软件要容易得多，并可以避免各种易犯错误。更多的细节在 Dynamic C 手册或 Dynamic C 库的源代码里。

16．1 读和写 I/O 寄存器和影子寄存器（shadow registers）

Rabbit 有两个 I/O 空间：内部 I/O 寄存器和外部 I/O 寄存器。访问它们与访问数据内存相同，除了指令加一个前缀（ioi 或 ioe）以表明是内部还是外部 I/O 空间。

在 Dynamic C 里读和写 I/O 寄存器的最快方法，是在 C 程序里插入一小段汇编语言。举例如下。

```
// 计算数值并写到端口 A 数据寄存器
value=x+y
#asm
ld a,(value) ； 要写的数值
ioi ld (PADR),a ； 写此值到 PADR
#endasm
```

在这个例子里，ioi 前缀把到内存的存储动作改为到 I/O 端口的存储动作。前缀 ioe 用来写外部 I/O 端口。

用户可获得一系列可调用的函数来读和写 I/O 寄存器。

```
// 内部 I/O 寄存器调用
```

```
int RdPortI(int PORT); //  返回端口，高字节为 0
int BitRdPortI( int PORT, int bitcode) ;  //  位代码 0-7
//写 8 位到端口和影子寄存器
//  如果指向影子寄存器的指针是 NULL，不写影子寄存器
void WrPortI( int PORT, char *PORTShadow, int value);
//  写一个端口位 (各位以 7, 6, ... 1, 0 标识)
void BitWrPortI(int PORT, char *PORTShadow, int value, int bitcode);
//  同一个外部 I/O 寄存器
int RdPortE(int adr); //  返回端口内容，高字节 0
int BitRdPortE( int PORT, int bitcode) ;  //  位代码 0-7
int WrPortE( int PORT, char *PORTShadow, int value);
int BitWrPortE( int PORT, char *PORTShadow, int value, int bitcode);
```

为了读端口可以使用以下代码：

```
k=RdPortI(PDDR); //  返回端口 D 数据寄存器
```

如果端口是只写的，可以用影子寄存器查明端口的内容。例如，全局控制状态寄存器有很多只写的位。可通过参考影子寄存器来读这些位，只要写寄存器时其影子寄存器不断更新。

```
k=GCSRShadow;
```

为了写一个只写的寄存器并更新影子寄存器内容，可用下面的程序：

```
WrPortI(GCSR,&GCSRShadow,value); //更新寄存器和影子寄存器
BitWrPortI(GCSR,&GCSRShadow,1,5); // GCSR 的位 5 置位
```

在 WordPortI 程序里，如果不使用影子寄存器，可以用 NULL 代替指向影子寄存器的指针。影子寄存器的指针对 BitWrPortI 是强制性的。

16．2 影子寄存器

很多 Rabbit 内部 I/O 器件的寄存器是只写的。只写寄存器在芯片上节省了门控电路，使用较低的成本可获得较高的性能。典型的实现外部 I/O 寄存器的设计也有只写寄存器，目的是去除附加的硬件设备以节省成本，这样使读回寄存器的内容成为可能。只写寄存器很容易使用，只要内存位置（即影子寄存器）与每个只写寄存器相关联。为了便于记忆影子寄存器的名字，它们与本手册中的内部寄存器名字混合使用。例如，寄存器 PADR（端口 A 数据寄存器）有影子寄存器 PADRShadow。一些影子寄存器在 BIOS 文件里定义，如下所示。

```
//  内部 I/O 寄存器—影子寄存器
//并行端口
char PADRShadow,PBDRShadow, PCDRShadow, PCFRShadow;
char PDDRShadow, PDCRShadow, PDFRShadow, PDDCRShadow, PDDDRShadow;
char PEDRShadow, PECRShadow, PEFRShadow, PEDDRShadow;
char GCSRShadow;       //  全局控制/状态寄存器
```

```
char GOCRShadow;      // 全局控制
char GCDRShadow;      // 时钟倍频器
```

操作 I/O 寄存器和影子寄存器时，编程者必须记住，可以在一个运算序列的中间发生一个中断，这时中断程序可能操作同一寄存器。如果这种可能存在，必须为特定情况设计解决方法。一般情况下，操作寄存器和它们相关的影子寄存器时没必要一定禁止中断。

举个例子，考虑并行端口 D 数据方向寄存器（PDDDR）。这个寄存器是只写的，包含了相应于并行端口 D 的 8 个引脚的 8 个比特位。如果这个寄存器里某位为“1”，对应的端口引脚是输出，不然就是输入。可以很简单的想出解决方法，就是让应用程序的不同部分（如中断程序和后台程序）负责 PDDDR 中的不同位。下面的代码置位影子寄存器的某位，接着置位 I/O 寄存器。如果在 set 和 ldd 间发生了一个中断，并改变了影子寄存器和 PDDDR，正确值会在 PDDDR 里被置位。

```
ld  hl,PDDDRShadow     ;  影子寄存器的指针
ld  de,PDDDR           ;  让 de 指向 I/O 寄存器
set  5,(hl)            ;  置位影子寄存器的位 5
;使用 ldd 指令进行微传输
ioi  ldd    ; (io  de)<-(hl)   副作用: hl--, de--
```

这种情况下，带 I/O 前缀地使用 ldd 指令提供了从内存位置到 I/O 位置的方便的数据转移。更重要的是，ldd 指令是一个很微小的操作，因此在数据转移到 PDDDR 寄存器期间没有中断程序改变影子寄存器的危险。如果使用下面的两条指令代替了 ldd 指令，

```
ld a,(hl)
ld  (PDDDR),a   ;  输出到  PDDDR
```

那么在第一条指令发生之后有可能发生中断，并改变影子寄存器和 PDDDR 寄存器的内容，然后从中断返回后，第二条指令执行并把影子寄存器的过时拷贝存在 PDDDR 里，而设置它成为了一个错误值。

没有理由为许多可写的寄存器只用一个影子寄存器。在某些情况下，用写寄存器操作作为改变外设状态的简单方法，但所写的数据位被忽略。举个例子，写 Rabbit 串行端口的状态寄存器，可用来清除发送器中断请求，但数据位被忽略，因为状态寄存器实际上是只读寄存器（有附带在写寄存器操作上的特殊功能时除外）。某只写寄存器不必一定有影子寄存器的实例，是 Rabbit 串行端口里的发送器数据寄存器。发送器数据寄存器是一个只写寄存器，但没有理由为它设一个影子寄存器，因为任何存入的数据位马上就在串行端口上发送出去了。

16．3 定时器和时钟的使用

实时时钟或电池可供电时钟是一个 48 位计数器，每秒计数 32768 次。计数频率来自 32.768kHz 振荡器，它独立于主振荡器。另两个重要的器件也由 32.768kHz 振荡器驱动：周期性中断和看门狗定时器。假定所有时间量度标准都来自这个时钟而不是主处理器时钟，后者频率高的多（例如 22.184MHz），这样就允许主处理器振荡器使用较便宜的陶瓷谐振器，而可以不用石英晶体。陶瓷谐振器一般有 0.5%的错误，而石英晶体精确的多，每天只有几秒。

不允许对实时时钟频繁地读和写。读的过程很冗长，执行时间不确定。写的过程则更复杂。因此 Dynamic C 软件在内存里保留了一个长整型变量 SEC_TIMER。SEC_TIMER 由周期性中断每秒更新，

可由用户程序直接写或读。由于 SEC_TIMER 由与实时时钟相同的振荡器驱动，它们两者之间没有相对的时间增多或减少。作为标准启动码的一部分，SEC_TIMER 把实时时钟的位 15-46 拷贝到自身里。SEC_TIMER 保持从 1980 年 1 月 1 日上午 12 点开始到现在的秒数，可容纳从 1980 年开始的 136 年，即到 2116 年。另一个长整型定时器 MS_TIMER 以毫秒计数并可用相同的方式使用。MS_TIMER 大约每 6 周时间从最大计数绕回到零。使用这些计数器的软件能正确的测量间隔，即使计数器有绕回。

```
unsigned long int read_rtc(void); //  读实时时钟的位 15-46
void write_rtc(unsigned long int time); //  写位 15-46
//  注意：位  0-14  和  位  47  为 0
```

我们提供了两个实用程序，可用它们在传统格式时间（2000-1-10 17:34:12）和从 1980-1-1 以来的秒数之间进行时间切换。

```
//  把时间数据结构转换成秒数
unsigned long mktime(struct tm *timeptr);

//  把秒数转换成时间数据结构
unsigned int mktm(struct tm *timeptr, unsigned long time);
```

所使用的时间数据结构如下

```
struct tm {
        char  tm_sec;          //  0-59 秒
        char  tm_min;          //  0-59 分
        char  tm_hour;         //  0-23 小时
        char  tm_mday;         //  1-31 日
        char  tm_mon;          //  1-12 月
        char  tm_year;         //  00-150  年(1900-2050)
        char  tm_wday;         //   周 0-6  ，0==周日
    };
```

转换时没有利用星期几来计算这个长秒数，但是当把长秒数计算成这个结构时会产生它。程序 setclock.c 可用来在 Dynamic C 控制台里设置实时时钟里的日期和时间。

16．4 支持看门狗的软件

微处理器系统可因为各种原因而崩溃。软件 bug 或电气干扰是很常见的原因。系统崩溃时，典型情况下程序将进入一个无限循环，因为控制循环行为的参数已经破坏。典型情况是堆栈破坏，所有的程序都返回到随机地址。

对崩溃通常采取的处理措施是复位微处理器并重新启动系统。我们可以事先发觉崩溃，或者因为程序一致性检查检测到异常，或者因为应该周期执行的程序某部分并没有执行而看门狗超时。

在 Dynamic C 里支持对崩溃的直接检查，方式是校验和操作以及其他致命的错误原因。

虚拟看门狗系统允许为周期性执行的程序的不同部分建立多个虚拟看门狗。如果这些狗中任何一个超时，则硬件看门狗抑制瞬时断开（ hits）并超时，导致硬件复位。通过一个内存计数器数组实现虚拟看门狗。计数器通过中断程序每秒向下计数 16 次。断开某个看门狗通过调用一个向内存计数器里存入一个 1 到 255 之间的计数值来完成。虚拟看门狗可以分配、撤消分配、使能和禁止。缺省时只有一个虚拟看门狗，它在周期中断程序里被瞬时断开。如果周期中断停止工作，看门狗超时。虚拟看门狗的优点是，如果它们的任何一个失败，系统就可检测到错误。仅在把看门狗被断开的每个位置都包括进崩溃循环失败时，直接断开硬件看门狗才导致错误。

16．4．1 看门狗硬件

Rabbit 微处理器有一个硬件看门狗定时器。通过调用一个 BIOS 程序 hitwd()来断开看门狗，它把看门狗断开 2 秒钟；或者在 WDTCR 寄存器里存入一个特殊代码断开它，这个码决定断开时间。看门狗是一个 17 位计数器，它以 32768Hz（32.768kHz 振荡器提供）的频率向下计数到零。断开动作会存一个数到计数器，让计数器计数到零以延迟时间。如果断开不够频繁，计数器将到达零并执行处理器复位。

实践要求最好在程序断开看门狗之前极端小心。如果把对看门狗的断开不顾后果地在用户程序里到处分布设置，很可能看门狗没有任何实际用处，因为崩溃发生时程序进入一个无限循环，此循环包括了对看门狗的断开动作。

16．4．2 虚拟看门狗系统

缺省时有 10 个虚拟看门狗。可以用#define 改变这个值：

```
#define N_WATCHDOG 15 //  缺省 10 个
```

分配一个看门狗时，做如下调用：

```
N= VdGetFreeWd(char count); //配置一个看门狗，超时常数 count/16 秒
```

为断开这个看门狗，调用：

```
VdHitWd(int N); //  断开看门狗
```

从表里移去看门狗，调用：

```
VdReleaseWd(int N); //  撤消看们狗
```

17．Rabbit 标准 BIOS

（注意：本章内容仅用于理论目的。软件手册形式的升级将澄清这些问题。）

Rabbit BIOS 是一个软件包，它处理启动，关机和 Rabbit 的各种基本特性。通过提供标准软件来执行基本功能，用户可以从为了自己的需要而重新设计软件的必要里解脱出来。而且，使用 Z-World 测试过的软件，用户可以大大减少错误和 bug 的可能性。Z-World 为 BIOS 提供完全的源代码，所以用户可以修改它，这样用户考察 BIOS 的操作细节有了一个方便的参考，而这些操作细节在 BIOS 文档里并不明了。

总的来说，BIOS 可为每个不同的控制器主板定制。BIOS 在代码的开头有声明，定义了硬件培植和使用选项。

可获得一个通用用途的 BIOS，它可以在基于 Rabbit 的大多数系统上工作。通用用途的 BIOS 在形成新设计方案时很有用。

17．1 BIOS 的更多细节

BIOS 的编译独立于用户应用程序。它占用根代码段底部的空间。当用户程序从零地址开始执行时，它在 BIOS 里开始。BIOS 可包括进来的代码数量没有限制。如果用户把库作为 BIOS 的一部分进行编译，可以节省时间，因为除非特定的原因 BIOS 不会重新编译。

一般，频繁调用的程序或基本功能所需的程序都在 BIOS 里。当 Dynamic C 冷启动目标系统并下载了其二进制映像，则符号表被保持下来以使用户程序可调用 BIOS 的入口点。

BIOS 支持以下服务：

·系统启动，包括内存配置、等待状态和时钟速度。
·读并编程实时时钟。
·操作和管理周期性中断
·维护对时钟行走（ticks)、毫秒和秒数（1980 年以来）计数的内存计数器。
·看门狗定时器的操作和虚拟看门狗定时器系统的维护
·提供为电源管理目的而加速或降低系统时钟的程序。执行速度可在一个宽范围内进行控制。
·配置并行端口的操作模式和输入输出数据到端口的程序。
·初始化定时器并操作它们的程序。
·里用系统识别区域的的内嵌保护写闪存的程序。
·保存错误记录或操作记录和处理致命错误及看门狗超时的程序。
·基本的多任务服务
·通过编程端口支持通用用途参数配置的程序。这个系统可用在配置网络地址或定位常数和配置实时时钟等地方。
·下载管理器（在将来 BIOS 发行版里可获得）
·Modbus 从处理器

17．2　关于 BIOS 假定（BIOS Assumption）

BIOS 关于处理器的物理配置作出一些特定假定。期望处理器具有连接到/CS1，/WE1 和/OE1 的内存。如果有闪存的话，期望它连接到/CS0，/WE0 和/OE0。晶体频率期望值 n*0.6144*3MHz，或 4*0.6144MHz。

17．3 周期中断和实时时钟 BIOS 服务

实时时钟由 32.768kHz 振荡器驱动，振荡器可由电池供电。周期性中断使能时，每 16 个时钟周期或 488μs 发生一次。如果没有 32.768kHz 振荡器，可以置换为另一个周期中断，但这个可选项 Z-World 不支持，因为连接一个晶振的成本是很小的。

周期中断用来使几个内存计数器计数，它们用于通用软件。这些计数器计数时钟走动、毫秒和秒数。这些计数器与实时时钟和 32.768kHz 振荡器有严格的关系。一个秒计数器通过在每次时钟走动时把 65536/2048=32 加到一个 16 位字上生成，执行结果使一个秒计数器计数。一个毫秒计数器，通过在每次时钟走动时把 32000 加到一个 16 位字上生成，执行结果平均每毫秒发生一次，并驱动毫秒计数器。时钟走动计数器每秒计数 2048 次。

此外，周期中断通过对时钟程序进行周期调用而提供对功能切片（function slicing）和实时核心（real-time kernel）的支持。周期中断迫使其他中断关断（就是说，处理器优先级从 0 提高到 1）的最小保持时间大约是 35 个时钟周期。用这种方法，周期中断对中断等待时间几乎没有影响。

17．3．1 实时时钟支持

如果实时时钟由电池供电——这是 BIOS 里的一个选择项，BIOS 在启动时读实时时钟，并建立与时钟同步的从 1980 年以来的秒数的计数器。如果时钟是非电池供电的，此计数器设为 0，因而量度的时间是从启动以来的。启动时间和条件的记录作为一个调试帮助予以保留。正退出的复位的时间也被记录下来，原因与上相同，因为复位被保持在电池供电或非电池供电的内存里。

17．3．2 看门狗定时器支持

在一个组织良好的系统里，周期中断应该是指令可以瞬时断开看门狗定时器的唯一地方。通过设置很多虚拟看门狗定时器可达到这个目的。每个定时器都是一个 8 位计数器，每秒或实时时钟每走动 128 次，它向下数 16 次。如果任何一个计数器达到 0，程序关断中断并冻结它，直到硬件看门狗超时并复位处理器。用户程序必须周期地把每个虚拟狗断开，这通过调用一个带有看门狗数目和要存入的计数值的程序实现。虚拟狗数目本身是一个可通过调用可获得虚拟狗的程序来增加的参数，此程序增加另一个看门狗到此列表（非连接的列表）中，最多可达数组最大单元数目。最初没有一个看门狗被周期中断断开。虚拟狗可以在各中断间分布，以减少中断等待时间，如果所有可能的虚拟狗必须在一个中断里计数的话。应采取预防措施，保证崩溃不会导致偶然性断开看门狗。

17．3．3 电源管理支持

功耗和操作速度可以用粗略的同步控制为高或低，这通过改变时钟速度、时钟倍频器和内存等待状态来实现。控制范围可非常宽，有 16-1 或更高。此外，主时钟可切换到 32.768kHz 时钟。这种情况下，速度减低很剧烈，可为 500-1。每个时钟大约 30μs，一条典型指令的执行约 150μs。在这个低速度下，周期中断不能再工作，因为中断程序的执行太慢，不能跟上每 16 个时钟一次的中断。时钟走动一次的时间，只能执行大约 3 条指令。

超低功耗模式下使用的则是另一套不同的规则。用户建立一个无限循环，来决定什么时候退出超低模式。用户应该在这个循环里包含对一个查询程序地调用，这个查询程序是 BIOS 的一部分，它在每次被调用时更新内存计数器和看门狗的内容，其方式是直接读实时时钟和跟踪内存计数器。如果用户程序在看门狗定时器最大超时时间里不能进行完这个循环，用户应该在循环里做多次查询程序的调用。用户应该避免对看门狗定时器和实时时钟的不加区分的直接访问。实时时钟的最低几位不能在超低功耗模式下读取，因为相对于指令执行时间，它们计数太快。

17．3．4 写闪存的支持

BIOS 提供写闪存的程序。当要写的内存在主代码内存（primary code memory）里或在分离的特殊内存里，此程序以不同方式工作或者不同情况下有不同的程序。写主代码内存，需要冻结系统大约 10ms。而其他的写操作只要求用户在进行下一个写操作前等待写的完成。

为保护系统识别块（identification block），程序要测试相对于闪存（连接到/CS0、/WE0 和/OE0）开端的绝对内存地址，此闪存用来存储系统识别块。程序应该检查内存体控制寄存器的内容来完成这个测试。BIOS 里没有包含能实际写这个块的程序，这样非故意地偶然性写这个块就很困难。

18．Rabbit 指令

概要

装载立即数（Load Immediate Data）
8 位的索引式装载和存储（8-bit Indexed Load and Store）
16 位的索引式装载和存储
16 位装载并存入 20 位地址
寄存器到寄存器的转移（Registers to Registers Moves）
交换指令（Exchange Instructions）
栈操作指令（Stack Manipulation Instructions）
16 位算术和逻辑操作（16-bit Arithmetic and Logical Operations）
8 位算术和逻辑操作
8 位的置位，复位和测试指令（8-bit Bit Set,Rest and Test Insturctions）
8 位的增加和减少（8-bit Increment and Decrement）
8 位快速 A 寄存器操作（8-bit Fast A register Operations）
8 位移位和旋转（8-bit Shifts and Rotates）
指令前缀（Instruction Prefix）
块移动指令（Block Move Instructions）
控制指令--跳转和调用（Control Instructions-Jumps and Calls）
混合指令（Miscellaneous Instructions）
特权指令（Privileged Instructions）
指令以字母为顺序带二进制编码（Instructions in Alphabetical Order with Binary Encoding）

电子表格约定（Spreadsheet Conventions）

ALTD（"A"栏）符号键

标　记（flag）	描述
f	ALTD 选择备用标志
fr	ALTD 选择备用标志和寄存器
r	ALTD 选择辅助寄存器
s	ALTD 操作是特殊情况

IOI 和 IOE（"I"栏）符号键

标记	描述
b	IOI 和 IOE 影响源和目的端
d	IOI 和 IOE 影响目的端
s	IOI 和 IOE 影响源端

标志寄存器键

S	Z	L/V[1]	C	描述
*				符号标志受到影响
-				符号标志未受影响
	*			零标志受到影响
	-			零标志未受影响
		L		LV 标志包含逻辑校验结果
		V		LV 标志包含算术溢出结果
		0		LV 标志被清除
		*		LV 标志受到影响
			*	进位标志受到影响
			-	进位标志未受影响
			0	进位标志被清除
			1	进位标志置位

[1] L/V（逻辑/溢出）标记有两种作用——如果结果的 4 个最高有效位的任一位是 1，L/V 置 1 以进行逻辑操作，如果结果的 4 位最高有效位都是 0，L/V 置 0。

符号（Symbols）

Rabbit	Z180	含义
B	b	位选择： 000=位 0，001=位 1，010=位 2，011=位 3 100=位 4，101=位 5，110=位 6，111=位 7
Cc	cc	状态码选择： 00=NZ，01=Z，10=NC，11=C
D	d	7 位（有符号）置换，表示为二进制补码
Dd	ww	字寄存器选择目的端：00=BC，01=DE，10=HL，11=SP
dd`		字寄存器选择辅助寄存器：00=BC`，01=DE`，10=HL`
E	j	8 位（有符号）的置换被加到 PC 上。
F	f	状态码选择： 000=NZ（非 0），001=Z（0） 010=NC（无进位），001=C（进位） 100=LZ[1]（逻辑 0），101=LO[2]（逻辑 1） 110=P（正），111=M（负）
M	m	一个 16 位常数的最高有效位（MSB）
Mn	mn	16 位常数
N	n	8 位常数或 16 位常数的最低有效位（LSB）
r,r`	g,g`	字节寄存器选择： 000=B，001=C，010=D，011=E，100=H，101=L，111=A
V	v	重新启动地址的选择： 010=0020h，011=0030h，100=0040h，101=0050h，111=0070h
Xx	xx	字寄存器的选择：00=BC，01=DE，10=IX，11=SP
Yy	yy	字寄存器的选择：00=BC，01=DE，10=IY，11=SP
Zz	zz	字寄存器的选择：00=BC，01=DE，10=HL，11=AF

[1]如果结果的 4 个最高有效位都是 0，则为逻辑 0

[2] 结果的 4 个最高有效位任一为 1，为逻辑 1

１８．１装载立即数（Load Immediate Data）

指令	clk	A	I S Z V C	操作
LD IX,mn	8		- - - -	IX = mn
LD IY,mn	8		- - - -	IY = mn
LD dd,mn	6	r	- - - -	dd = mn
LD r,n	4	r	- - - -	r = n

１８．２装载并存入立即地址（Load and Stor to an Immediate Address）

指令	clk	A	I S Z V C	操作
LD (mn),A	10		d - - - -	(mn) = A
LD A,(mn)	9	r	s - - - -	A = (mn)
LD (mn),HL	13		d - - - -	(mn) = L; (mn+1) = H
LD (mn),IX	15		d - - - -	(mn) = IXL; (mn+1) = IXH
LD (mn),IY	15		d - - - -	(mn) = IYL; (mn+1) = IYH
LD (mn),ss	15		d - - - -	(mn) = ssl; (mn+1) = ssh
LD HL,(mn)	11	r	s - - - -	L = (mn); H = (mn+1)
LD IX,(mn)	13		s - - - -	IXL = (mn); IXH = (mn+1)
LD IY,(mn)	13		s - - - -	IYL = (mn); IYH = (mn+1)
LD dd,(mn)	13	r	s - - - -	ddl = (mn); ddh = (mn+1)

１８．３8 位索引式装载和存储

指令	clk	A	I S Z V C	操作
LD A,(BC)	6	r	s - - - -	A = (BC)
LD A,(DE)	6	r	s - - - -	A = (DE)
LD (BC),A	7		d - - - -	(BC) = A
LD (DE),A	7		d - - - -	(DE) = A
LD (HL),n	7		d - - - -	(HL) = n
LD (HL),r	6		d - - - -	(HL) = r = B, C, D, E, H, L, A
LD r,(HL)	5	r	s - - - -	r = (HL)
LD (IX+d),n	11		d - - - -	(IX+d) = n
LD (IX+d),r	10		d - - - -	(IX+d) = r
LD r,(IX+d)	9	r	s - - - -	r = (IX+d)
LD (IY+d),n	11		d - - - -	(IY+d) = n
LD (IY+d),r	10		d - - - -	(Iy+d) = r
LD r,(IY+d)	9	r	s - - - -	r = (IY+d)

１８．４16 位索引式装载和存储

指令	clk	A	I	S	Z	V	C	操作
LD (HL+d),HL	13		d	-	-	-	-	(HL+d) = L; (HL+d+1) = H
LD HL,(HL+d)	11	r		s	-	-	-	L = (HL+d); H = (HL+d+1)
LD (SP+n),HL	11				-	-	-	(SP+n) = L; (SP+n+1) = H
LD (SP+n),IX	13				-	-	-	(SP+n) = IXL; (SP+n+1) = IXH
LD (SP+n),IY	13				-	-	-	(SP+n) = IYL; (SP+n+1) = IYH
LD HL,(SP+n)	9	r			-	-	-	L = (SP+n); H = (SP+n+1)
LD IX,(SP+n)	11				-	-	-	IXL = (SP+n); IXH = (SP+n+1)
LD IY,(SP+n)	11				-	-	-	IYL = (SP+n); IYH = (SP+n+1)
LD (IX+d),HL	11		d	-	-	-	-	(IX+d) = L; (IX+d+1) = H
LD HL,(IX+d)	9	r		s	-	-	-	L = (IX+d); H = (IX+d+1)
LD (IY+d),HL	13		d	-	-	-	-	(IY+d) = L; (IY+d+1) = H
LD HL,(IY+d)	11	r		s	-	-	-	L = (IY+d); H = (IY+d+1)

１８．５16 位装载并存入 20 位地址

指令	clk	A	I	S	Z	V	C	操作
LDP (HL),HL	12				-	-	-	(HL) = L; (HL+1) = H. (Adr[19:16] = A[3:0])
LDP (IX),HL	12				-	-	-	(IX) = L; (IX+1) = H. (Adr[19:16] = A[3:0])
LDP (IY),HL	12				-	-	-	(IY) = L; (IY+1) = H. (Adr[19:16] = A[3:0])
LDP HL,(HL)	10				-	-	-	L = (HL); H = (HL+1). (Adr[19:16] = A[3:0])
LDP HL,(IX)	10				-	-	-	L = (IX); H = (IX+1). (Adr[19:16] = A[3:0])
LDP HL,(IY)	10				-	-	-	L = (IY); H = (IY+1). (Adr[19:16] = A[3:0])
LDP (mn),HL	15				-	-	-	(mn) = L; (mn+1) = H. (Adr[19:16] = A[3:0])
LDP (mn),IX	15				-	-	-	(mn) = IXL; (mn+1) = IXH. (Adr[19:16] = A[3:0])
LDP (mn),IY	15				-	-	-	(mn) = IYL; (mn+1) = IYH. (Adr[19:16] = A[3:0])
LDP HL,(mn)	13				-	-	-	L = (mn); H = (mn+1). (Adr[19:16] = A[3:0])
LDP IX,(mn)	13				-	-	-	IXL = (mn); IXH = (mn+1). (Adr[19:16] = A[3:0])
LDP IY,(mn)	13				-	-	-	IYL = (mn); IYH = (mn+1). (Adr[19:16] = A[3:0])

注意：

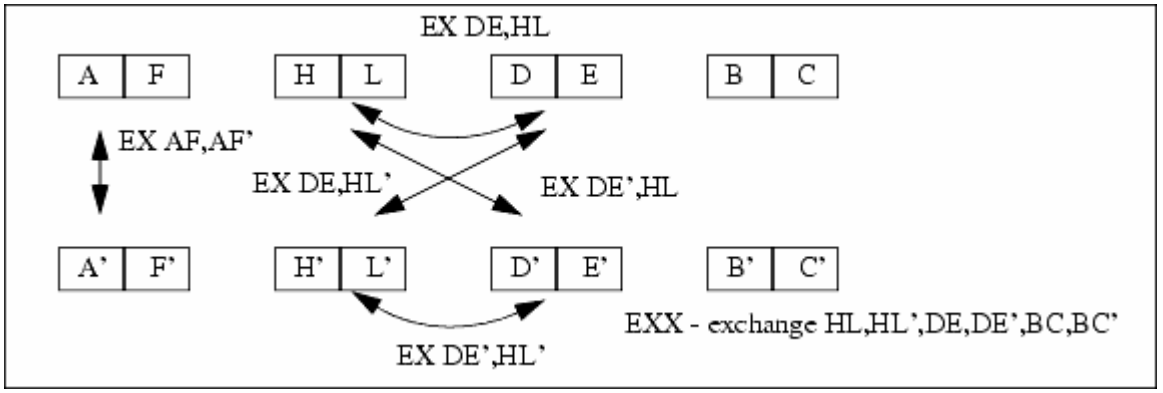
请注意 LDP 指令在 64K 页边界上会产生绕回。如果你试图跨越页边界写或读由于 LDP 指令操作 2 字节的数据，第二个字节将绕回并在页开始处被写入。这样，如果你在地址 0xn,0xFFFF 上进行取出或存入操作，得到的是 0xn,0xFFFF 和 0xn,0x0000 上的两个字节，而不是你所期望的 0xn,0xFFFF 和 0x(n+1),0x0000 上的两个字节。因此，不要在任何以 0xFFFF 结束的物理地址上使用 LDP 指令。

１８．６寄存器到寄存器的转移

指令	clk	A	I S Z V C	操作
LD r,g	2	r	- - - -	r = g- r,g any of B, C, D, E, H, L, A
LD A,EIR	4	fr	* * - -	A = EIR
LD A,IIR	4	fr	* * - -	A = IIR
LD A,XPC	4	r	- - - -	A = MMU
LD EIR,A	4		- - - -	EIR = A
LD IIR,A	4		- - - -	IIR = A
LD XPC,A	4		- - - -	XPC = A
LD HL,IX	4	r	- - - -	HL = IX
LD HL,IY	4	r	- - - -	HL = IY
LD IX,HL	4		- - - -	IX = HL
LD IY,HL	4		- - - -	IY = HL
LD SP,HL	2		- - - -	SP = HL
LD SP,IX	4		- - - -	SP = IX
LD SP,IY	4		- - - -	SP = IY
LD dd',BC	4		- - - -	dd' = BC (dd': 00-BC', 01-DE', 10-HL')
LD dd',DE	4		- - - -	dd' = DE (dd': 00-BC', 01-DE', 10-HL')

１８．７交换指令

指令	clk	A	I S Z V C	操作
EX (SP),HL	15	r	- - - -	H <-> (SP+1); L <-> (SP)
EX (SP),IX	15		- - - -	IXH <-> (SP+1); IXL <-> (SP)
EX (SP),IY	15		- - - -	IYH <-> (SP+1); IYL <-> (SP)
EX AF,AF'	2		- - - -	AF <-> AF'
EX DE',HL	2	s	- - - -	if (!ALTD) then DE' <-> HL else DE' <-> HL'
EX DE',HL'	4	s	- - - -	DE' <-> HL'
EX DE,HL	2	s	- - - -	if (!ALTD) then DE <-> HL else DE <-> HL'
EX DE,HL'	4	s	- - - -	DE <-> HL'
EXX	2		- - - -	BC <-> BC'; DE <-> DE'; HL <-> HL'



１８．８栈操作指令

指令	clk	A	I S Z V C	操作
ADD SP,d	4	f	- - - *	SP = SP + d -- d=0 to 255
POP IP	7		- - - -	IP = (SP); SP = SP+1
POP IX	9		- - - -	IXL = (SP); IXH = (SP+1); SP = SP+2
POP IY	9		- - - -	IYL = (SP); IYH = (SP+1); SP = SP+2
POP zz	7	r	- - - -	zzl = (SP); zzh = (SP+1); SP=SP+2 -- zz= BC,DE,HL,AF
PUSH IP	9		- - - -	(SP-1) = IP; SP = SP-1
PUSH IX	12		- - - -	(SP-1) = IXH; (SP-2) = IXL; SP = SP-2
PUSH IY	12		- - - -	(SP-1) = IYH; (SP-2) = IYL; SP = SP-2
PUSH zz	10		- - - -	(SP-1) = zzh; (SP-2) = zzl; SP=SP-2 --zz= BC,DE,HL,AF

１８．９16 位算术和逻辑操作

指令	clk	A	I S Z V C	操作
DC HL,ss	4	fr	* * V *	HL = HL + ss + CF -- ss=BC, DE, HL, SP
ADD HL,ss	2	fr	- - - *	HL = HL + ss
ADD IX,xx	4	f	- - - *	IX = IX + xx -- xx=BC, DE, IX, SP
ADD IY,yy	4	f	- - - *	IY = IY + yy -- yy=BC, DE, IY, SP
ADD SP,d	4	f	- - - *	SP = SP + d -- d=0 to 255
AND HL,DE	2	fr	* * L 0	HL = HL & DE
AND IX,DE	4	f	* * L 0	IX = IX & DE
AND IY,DE	4	f	* * L 0	IY = IY & DE
BOOL HL	2	fr	* * 0 0	If (HL != 0) HL = 1,

					标志置位以匹配 HL
BOOL IX	4	f	* * 0 0		If (IX != 0) IX = 1
BOOL IY	4	f	* * 0 0		if (IY != 0) IY = 1
DEC IX	4		- - - -		IX = IX – 1
DEC IY	4		- - - -		IY = IY – 1
DEC ss	2	r	- - - -		ss = ss - 1 -- ss= BC, DE, HL, SP
INC IX	4		- - - -		IX = IX + 1
INC IY	4		- - - -		IY = IY + 1
INC ss	2	r	- - - -		ss = ss + 1 -- ss= BC, DE, HL, SP
MUL	12		- - - -		HL:BC = BC * DE, 有符号 32 位结果。DE 未改变
OR HL,DE	2	fr	* * L 0		HL = HL | DE – 按位或
OR IX,DE	4	f	* * L 0		IX = IX | DE
OR IY,DE	4	f	* * L 0		IY = IY | DE
RL DE	2	fr	* * L *		{CY,DE} = {DE,CY} -- 随 CF 左移位
RR DE	2	fr	* * L *		{DE,CY} = {CY,DE}
RR HL	2	fr	* * L *		{HL,CY} = {CY,HL}
RR IX	4	f	* * L *		{IX,CY} = {CY,IX}
RR IY	4	f	* * L *		{IY,CY} = {CY,IY}
SBC HL,ss	4	fr	* * V *		HL=HL-ss-CY (cout if (ss-CY)>hl)

１８．１０ 8 位算术和逻辑操作

指令	clk	A	I	S	Z	V	C	操作
ADC A,(HL)	5	fr	s	*	*	V	*	A = A + (HL) + CF
ADC A,(IX+d)	9	fr	s	*	*	V	*	A = A + (IX+d) + CF
ADC A,(IY+d)	9	fr	s	*	*	V	*	A = A + (IY+d) + CF
ADC A,n	4	fr	*	*	V	*		A = A + n + CF
ADC A,r	2	fr	*	*	V	*		A = A + r + CF
ADD A,(HL)	5	fr	s	*	*	V	*	A = A + (HL)
ADD A,(IX+d)	9	fr	s	*	*	V	*	A = A + (IX+d)
ADD A,(IY+d)	9	fr	s	*	*	V	*	A = A + (IY+d)
ADD A,n	4	fr		*	*	V	*	A = A + n
ADD A,r	2	fr		*	*	V	*	A = A + r
AND (HL)	5	fr	s	*	*	L	0	A = A & (HL)
AND (IX+d)	9	fr	s	*	*	L	0	A = A & (IX+d)
AND (IY+d)	9	fr	s	*	*	L	0	A = A & (IY+d)
AND n	4	fr		*	*	L	0	A = A & n
AND r	2	fr		*	*	L	0	A = A & r
CP* (HL)	5	f	s	*	*	V	*	A - (HL)
CP* (IX+d)	9	f	s	*	*	V	*	A - (IX+d)
CP* (IY+d)	9	f	s	*	*	V	*	A - (IY+d)

CP* n	4	f	* * V *	A – n
CP* r	2	f	* * V *	A – r
OR (HL)	5	fr	s * * L 0	A = A | (HL)
OR (IX+d)	9	fr	s * * L 0	A = A | (IX+d)
OR (IY+d)	9	fr	s * * L 0	A = A | (IY+d)
OR n	4	fr	* * L 0	A = A | n
OR r	2	fr	* * L 0	A = A | r
SBC* (IX+d)	9	fr	s * * V *	A = A - (IX+d) – CY
SBC* (IY+d)	9	fr	s * * V *	A = A - (IY+d) – CY
SBC* A,(HL)	5	fr	s * * V *	A = A - (HL) – CY
SBC* A,n	4	fr	* * V *	A = A-n-CY (cout if (r-CY)>A)
SBC* A,r	2	fr	* * V *	A = A-r-CY (cout if (r-CY)>A)
SUB (HL)	5	fr	s * * V *	A = A - (HL)
SUB (IX+d)	9	fr	s * * V *	A = A - (IX+d)
SUB (IY+d)	9	fr	s * * V *	A = A - (IY+d)
SUB n	4	fr	* * V *	A = A – n
SUB r	2	fr	* * V *	A = A – r
XOR (HL)	5	fr	s * * L 0	A = [A & ~(HL)] | [~A & (HL)]
XOR (IX+d)	9	fr	s * * L 0	A = [A & ~(IX+d)] | [~A & (IX+d)]
XOR (IY+d)	9	fr	s * * L 0	A = [A & ~(IY+d)] | [~A & (IY+d)]
XOR n	4	fr	* * L 0	A = [A & ~n] | [~A & n]
XOR r	2	fr	* * L 0	A = [A & ~r] | [~A & r]

*SBC 和 CP 指令输出进位的反码。操作或虚操作是（A-B）时，如果 A<B，C 被置位。如果 A>=B，进位被清除。SUB 在 SBC 和 CP 相反方向输出进位。

１８．１１ 8 位的置位，复位和测试指令

指令	clk	A	I	S	Z	V	C	操作
BIT b,(HL)	7	f		s	-	*	-	(HL) & bit
BIT b,(IX+d))	10	f		s	-	*	-	(IX+d) & bit
BIT b,(IY+d))	10	f		s	-	*	-	(IY+d) & bit
BIT b,r	4	f		-	*	-	-	r & bit
RES b,(HL)	10		d	-	-	-	-	(HL) = (HL) & ~bit
RES b,(IX+d)	13		d	-	-	-	-	(IX+d) = (IX+d) & ~bit
RES b,(IY+d)	13		d	-	-	-	-	(IY+d) = (IY+d) & ~bit
RES b,r	4	r		-	-	-	-	r = r & ~bit
SET b,(HL)	10		b	-	-	-	-	(HL) = (HL) | bit
SET b,(IX+d)	13		b	-	-	-	-	(IX+d) = (IX+d) | bit
SET b,(IY+d)	13		b	-	-	-	-	(IY+d) = (IY+d) | bit
SET b,r	4	r		-	-	-	-	r = r | bit

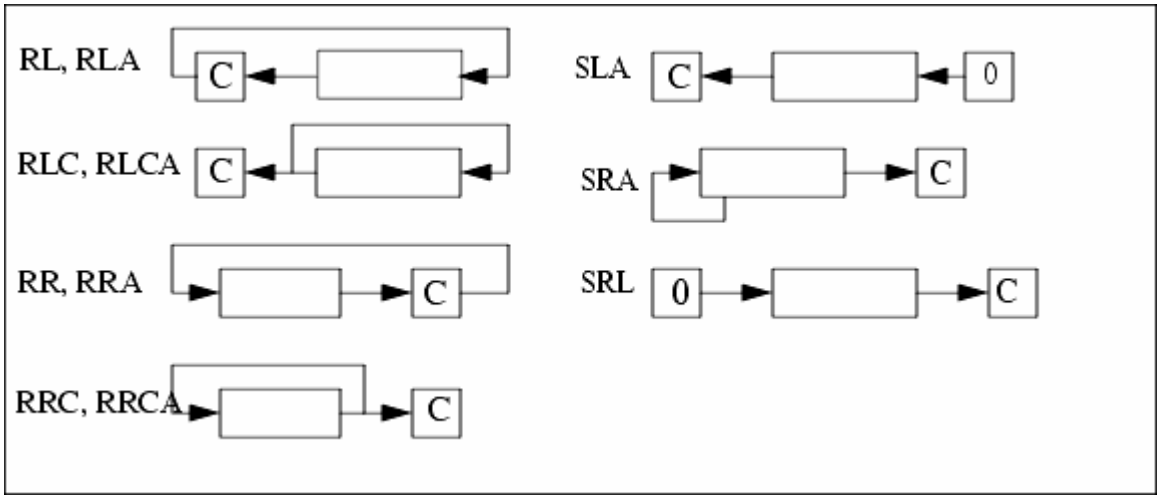
１８．１２　8 位增加和减少

指令	clk	A	I	S	Z	V	C	操作
DEC (HL)	8	f	b	*	*	V	-	(HL) = (HL) – 1
DEC (IX+d)	12	f	b	*	*	V	-	(IX+d) = (IX+d) –1
DEC (IY+d)	12	f	b	*	*	V	-	(IY+d) = (IY+d) –1
DEC r	2	fr	*	*	V	-		r = r – 1
INC (HL)	8	f	b	*	*	V	-	(HL) = (HL) + 1
INC (IX+d)	12	f	b	*	*	V	-	(IX+d) = (IX+d) + 1
INC (IY+d)	12	f	b	*	*	V	-	(IY+d) = (IY+d) + 1
INC r	2	fr	*	*	V	-		r = r + 1

１８．１３　8 位快速 A 寄存器操作

指令	clk	A	I	S	Z	V	C	操作
CPL	2	r	-	-	-	-		A = ~A
NEG	4	fr	*	*	V	*		A = 0 – A
RLA	2	fr	-	-	-	*		{CY,A} = {A,CY}
RLCA	2	fr	-	-	-	*		A = {A[6,0],A[7]}; CY = A[7]
RRA	2	fr	-	-	-	*		{A,CY} = {CY,A}
RRCA	2	fr	-	-	-	*		A = {A[0],A[7,1]}; CY = A[0]

１８．１４　8 位移位和旋转指令



指令	clk	A	I	S	Z	V	C	操作
RL (HL)	10	f	b	*	*	L	*	{CY,(HL)} = {(HL),CY}
RL (IX+d)	13	f	b	*	*	L	*	{CY,(IX+d)} = {(IX+d),CY}
RL (IY+d)	13	f	b	*	*	L	*	{CY,(IY+d)} = {(IY+d),CY}
RL r	4	fr	*	*	L	*		{CY,r} = {r,CY}
RLC (HL)	10	f	b	*	*	L	*	(HL) = {(HL)[6,0],(HL)[7]}; CY = (HL)[7]
RLC (IX+d)	13	f	b	*	*	L	*	(IX+d) = {(IX+d)[6,0], (IX+d)[7]}; CY = (IX+d)[7]

RLC (IY+d) 13 f b * * L * (IY+d) = {(IY+d)[6,0], (IY+d)[7]};
CY = (IY+d)[7]

RLC r 4 fr * * L * r = {r[6,0],r[7]}; CY = r[7]

RR (HL) 10 f b * * L * {(HL),CY} = {CY,(HL)}

RR (IX+d) 13 f b * * L * {(IX+d),CY} = {CY,(IX+d)}

RR (IY+d) 13 f b * * L * {(IY+d),CY} = {CY,(IY+d)}

RR r 4 fr * * L * {r,CY} = {CY,r}

RRC (HL) 10 f b * * L * (HL) = {(HL)[0],(HL)[7,1]};
CY = (HL)[0]

RRC (IX+d) 13 f b * * L * (IX+d) = {(IX+d)[0], (IX+d)[7,1]};
CY = (IX+d)[0]

RRC (IY+d) 13 f b * * L * (IY+d)={(IY+d)[0],(IY+d)[7,1]};
CY = (IY+d)[0]

RRC r 4 fr * * L * r = {r[0],r[7,1]};
CY = r[0]

SLA (HL) 10 f b * * L * (HL)= {(HL)[6,0],0};
CY =(HL)[7]

SLA (IX+d) 13 f b * * L * (IX+d) = {(IX+d)[6,0],0};
CY = (IX+d)[7]

SLA (IY+d) 13 f b * * L * (IY+d) = {(IY+d)[6,0],0};
CY = (IY+d)[7]

SLA r 4 fr * * L * r = {r[6,0],0};
CY = r[7]

SRA (HL) 10 f b * * L * (HL) = {(HL)[7],(HL)[7,1]};
CY = (HL)[0]

SRA (IX+d) 13 f b * * L * (IX+d) = {(IX+d)[7], (IX+d)[7,1]};
CY = (IX+d)[0]

SRA (IY+d) 13 f b * * L * (IY+d) = {(IY+d)[7], (IY+d)[7,1]};
CY = (IY+d)[0]

SRA r 4 fr * * L * r = {r[7],r[7,1]};
CY = r[0]

SRL (HL) 10 f b * * L * (HL) = {0,(HL)[7,1]};
CY = (HL)[0]

SRL (IX+d) 13 f b * * L * (IX+d) = {0,(IX+d)[7,1]};
CY = (IX+d)[0]

SRL (IY+d) 13 f b * * L * (IY+d) = {0,(IY+d)[7,1]};
CY = (IY+d)[0]

SRL r 4 fr * * L * r = {0,r[7,1]};
CY = r[0]

１８．１５　指令前缀

指令	clk	A	I S Z V C	操作
ALTD	2		- - - -	为下条指令替换目的寄存器
IOE	2		- - - -	I/O 外部前缀
IOI	2		- - - -	I/O 内部前缀

１８．１６　块移动指令

指令	clk	A	I S Z V C	操作
LDD	10	d	- - * -	(DE) = (HL); BC = BC-1; DE = DE-1; HL = HL-1
LDDR	6+7i	d	- - * -	if {BC != 0} repeat:
LDI	10	d	- - * -	(DE) = (HL); BC = BC-1; DE = DE+1; HL = HL+1
LDIR	6+7i	d	- - * -	if {BC != 0} repeat:

注意：

如果任何块移动指令加了 I/O 前缀，目的地址都指的是 I/O 空间。如果前缀是 IOI（内部 I/O），每使用一次前缀，增加一个块。如果前缀是 IOE，增加两个块和使能的 I/O 等待状态数。BC 从 1 转变到 0 时，V 标志被设置。如果没有设置 V 标记，执行另一个步骤，即重复执行这个版本的指令。在重复执行之间，可以发生中断，但不会在 LDD 或 LDI 的重复时发生。从中断返回的是指令的第一个字节，也就是这个前缀（如果有 I/O 前缀的话）。

１８．１７　控制指令—跳转和调用

指令	clk	A	I S Z V C	操作
CALL mn	12		- - - -	(SP-1) = PCH; (SP-2) = PCL; PC = mn; SP = SP-2
DJNZ j	5	r	- - - -	B = B-1; if {B != 0} PC = PC + j
JP (HL)	4		- - - -	PC = HL
JP (IX)	6		- - - -	PC = IX
JP (IY)	6		- - - -	PC = IY
JP f,mn	7		- - - -	If {f} PC = mn
JP mn	7		- - - -	PC = mn
JR cc,e	5		- - - -	if {cc} PC = PC + e
JR e	5		- - - -	PC = PC + e (如果 e=0，顺序执行下一指令)
LCALL xpc,mn	19		- - - -	(SP-1) = XPC; (SP-2) = PCH; (SP-3) = PCL; XPC=xpc; PC = mn; SP = (SP-3)
LJP xpc,mn	10		- - - -	XPC=xpc; PC = mn
LRET	13		- - - -	PCL = (SP); PCH = (SP+1); XPC = (SP+2); SP = SP+3
RET	8		- - - -	PCL = (SP); PCH = (SP+1); SP = SP+2
RET f	8/2		- - - -	if {f} PCL = (SP); PCH = (SP+1); SP = SP+2
RETI	12		- - - -	IP = (SP); PCL = (SP+1);

				PCH = (SP+2); SP = SP+3
RST v	10		- - - -	(SP-1) = PCH; (SP-2) = PCL;
				SP = SP - 2; PC = {R,v)
				v=10,18,20,28,38

１８．１８　混合指令

指令	clk	A	I	S	Z	V	C	操作
CCF	2	f	-	-	-	-	*	CF = ~CF
IPSET 0	4		-	-	-	-	-	IP = {IP[5:0], 00}
IPSET 1	4		-	-	-	-	-	IP = {IP[5:0], 01}
IPSET 2	4		-	-	-	-	-	IP = {IP[5:0], 10}
IPSET 3	4		-	-	-	-	-	IP = {IP[5:0], 11}
IPRES	4		-	-	-	-	-	IP = {IP[1:0], IP[7:2]}
LD A,EIR	4	fr	*	*	-	-		A = EIR
LD A,IIR	4	fr	*	*	-	-		A = IIR
LD A,XPC	4	r	-	-	-	-		A = MMU
LD EIR,A	4		-	-	-	-		EIR = A
LD IIR,A	4		-	-	-	-		IIR = A
LD XPC,A	4		-	-	-	-		XPC = A
NOP	2		-	-	-	-		空操作
POP IP	7		-	-	-	-		IP = (SP); SP = SP+1
PUSH IP	9		-	-	-	-		(SP-1) = IP; SP = SP-1
SCF	2	f	-	-	-	1		CF = 1
ZINTACK	10		-	-	-	-		(SP-1) = PCH; (SP-2) = PCL;
								SP = SP-2; IP = {IP[6:

１８．１９　特权指令

本小节描述特权指令。“特权”意味着在特权指令和下一指令之间，中断不可以发生。

下面的三条指令被许以特权。

```
LD SP,HL  ;  装载堆栈指针
LD SP,IY
LD SP,IX
```

装载堆栈的指针具有特权，所以它们后面可以跟随一个装载堆栈段（SSEG）寄存器的指令，而不用担心发生中断及堆栈指针和堆栈段寄存器之间会发生错误联系。举例：

```
        LD   SP,HL
IOI   LD   (STACKSEG),A
```

下面的指令有特权：

```
IPSET 0    ； 使 IP 左移位，并在位 1，0 里设置优先级 00
IPSET 1
IPSET 2
IPSET 3
IPRES      ；使 IP 循环右移两位，恢复原来的优先级
POP IP     ； 把 IP 寄存器从堆栈里弹出
```

修改 IP 寄存器的指令有特权，所以它们后面可以跟一条返回指令，这条指令确保可在另一个中断发生之前执行。这避免了堆栈不断增长。

```
RETI          ；从堆栈里弹出 IP，然后弹出返回地址
```

可用单条 RETI 指令设置返回地址和 IP。如果前面有 LD XPC，可以完成一次到所计算的地址的完全跳转或调用，而不会有中断发生。

```
LD A,XPC ； 获得并设置 XPC
LD XPC,A
```

指令 LD XPC，A 有特权，所以它后面可以跟随其他设置中断级别或程序计数器的代码，不会有干扰中断。

```
BIT B,(HL) ； 测试内存中的某位
```

指令 BIT B，（HL）有特权，使不禁止中断就实现信号量成为可能。使用下面的指令序列。其中有一位是信号量，而首要任务是设置拥有信号量的位和拥有操作信号量关联的资源的权力。

```
BIT B,(HL)
SET B,(HL)
JP z,ihaveit
；这里没有
```

SET 指令对标志位没有作用。由于 BIT 指令后不会发生中断，如果标志位为 0，意味着用 BIT 指令检验时信号量没有设置以及 SET 指令已经试图设置过信号灯量。如果 BIT 和 SET 指令之间允许发生一个中断，另一个程序也可以设置信号量，则此时两个程序会同时认为他们拥有这个信号灯。

１８．２０　操作码映射

（见原文）

19．Rabbit 和 Z80/Z180 指令的不同

Rabbit 与 Z80 和 Z180 的代码兼容性很高，而且很容易移植非依赖 I/O 的代码。不兼容的主要是涉及 I/O 指令或特定的硬件实现指令。Z80/Z180 废弃的而较重要的指令在 Dynamic C 汇编程序里自动用指令序列模拟。一些没有什么用处的指令被彻底废弃，不能模拟，使用这些指令的代码段必须重写。

下面的 Z80/Z180 指令已经被废弃，而且没有严格的替代指令：

DAA, HALT, DI, EI, IM 0, IM 1, IM 2, OUT, IN, OUT0, IN0, SLP, OUTI, IND, OUTD, INIR, OTIR, INDR, OTDR, TESTIO, MLT SP, RRD, RLD, CPI, CPIR, CPD, CPDR

这些操作码大多数处理 I/O 设备，因此不代表可移植代码。其中不涉及处理器 I/O 的操作码是 MLT SP, DAA, RRD, RLD, CPI, CPIR, CPD 和 CPDR。MLT SP 不是一条实用操作码。涉及十进制算术的代码 DAA、RRD 和 RLD 可以模拟，但模拟效果很差（可以获得状态寄存器里用于半进位的位，并可使用 push 和 pop af 指令获得访问权并设置和清除它），所以通常使用这些指令的代码应该重写。指令 CPI，CPIR，CPD 和 CPDR 是重复比较指令。这些指令用处不大，因为检测到相等的比较时扫描将停止，故不相等的比较会更有用。很难有效的模拟这些指令，所以建议重写使用这些指令的代码段，大多数情况下应该非常简单。

下面的操作码被废弃。

RST 0, RST 8, RST 30h

其他的 RST 指令都保留了下来，但中断向量重新定位到一个可变化的位置，此位置的基址由 EIR 寄存器配置。RST 可用一条 call 指令模拟，但不会由汇编程序自动完成，因为这些指令的大多数只由 Dynamic C 调试时使用。

下面的指令的操作码已改变。

EX (SP),HL　　----旧操作码 0E3h，新操作码 0EDh-054h

下面的指令使用与原先不同的寄存器名字。

```
LD A,EIR
LD EIR,A          ; R 寄存器
LD IIR,A
LD  A,IIR         ;I 寄存器
```

下面的 Z80/Z180 指令已废弃，不再支持。

```
CALL CC,ADR
JR (JP)   ncc,xxx ;  逆转条件
CALL ADR   xxx:
TST R      ( (HL),n)
```

PUSH DE
PUSH AF
AND r ((HL), n)
POP DE ； 在 h 里获得 a
LD A,d
POP DE

20．字母顺序的指令（带二进制编码的）

电子表格约定

ALTD（“A”栏）符号键

标记	描述
f	ALTD 选择备用标志
fr	ALTD 选择备用标志和寄存器
r	ALTD 选择辅助寄存器
s	ALTD 操作是特殊情况

IOI 和 IOE（“I”栏）符号键

标记	描述
b	IOI 和 IOE 影响源和宿端
d	IOI 和 IOE 影响宿端
	IOIE 和 IOE 影响源端

标记寄存器键

S	Z	L/V [1]	C	描述
*				符号标志受到影响
-				符号标志未受影响
	*			零标志受到影响
	-			零标志未受影响
		L		L/V 标志包含逻辑校验结果
		V		L/V 标志包含算术溢出结果
		0		L/V 标志清除
		*		L/V 标志受到影响
			*	进位标志受到影响
			-	进位标志未受影响
			0	进位标志清除
			1	进位标志置位

[1]L/V（逻辑/溢出）标志有两个用途——结果的 4 个最高有效位中任一个为 1，L/V 置 1 以进行逻辑操作；这 4 位都是 0，L/V 复位为 0。

符号（Symbol）

Rabbit	Z180	意义
b	b	位选择： 000=位 0，001=位 1，010=位 2，011=位 3 100=位 4，101=位 5，110=位 6，111=位 7
cc	cc	状态码选择： 00=NZ，01=Z，10=NC，11=C
d	d	7 位（有符号）置换，表示为二进制补码
dd	ww	字寄存器选择目的端：00=BC，01=DE，10=HL，11=SP
dd`		字寄存器选择辅助寄存器：00=BC`，01=DE`，10=HL`
e	j	8 位（有符号）的置换加到 PC 上
f	f	状态码选择： 000=NZ（非 0），001=Z（0） 010=NC（无进位），011=C（进位） 100=LZ[1]（逻辑 0），101=LO[2]（逻辑 1） 110=P（正号），111=M（负号）
m	m	16 位常数的最高有效位（MSB）
mn	mn	16 位常数
n	n	8 位常数或 16 位常数的最低有效位（LSB）
r,r`	g,g`	字节寄存器选择： 000=B，001=C，010=D，011=E 100=H，101=L，111=A
ss	ww	字寄存器选择（源端）：00=BC，01=DE，10=HL，11=SP
v	v	重新启动地址的选择： 010=0020h,011=0030h,100=0040h 101=0050h,111=0070h
xx	xx	字寄存器选择：00=BC，01=DE，10=IX，11=SP
yy	yy	字寄存器选择：00=BC，01=DE，10=IY，11=SP
zz	zz	字寄存器选择：00=BC，01=DE，10=HI，11=AF

[1]逻辑 0--结果的 4 位最高有效位都是 0 时

[2]逻辑 1—结果的 4 位最高有效位任一位是 1 时

（指令码一览表见原文）

附录 A

A．1 Rabbit 编程端口

编程端口在一个基于 Rabbit 的系统和 Dynamic C编程平台之间提供一个标准物理和电气接口。用一条专门的接口电缆和转换器把 PC 串行端口连接到编程端口，用一个 10 引脚的标准 2mm 连接器实现编程端口（当然，如果需要，用户可以改变连接器的硬件实现）。这么配置端口之后，PC 可以与目标系统通信，将它复位和重启。PC 串行接口上的 DTR 线用来驱动目标系统的复位线，它必须可由外部 CMOS 驱动器驱动。STATUS 引脚被基于 Rabbit 的目标系统用来在正测试的此系统中遇到断点时请求注意。SMODE 引脚由来自接口的+5V/+3V 电平上拉。当接口没有用大约 5kΩ电阻接地时，在主板上这些引脚的电平必须下拉。处于测试的目标系统为接口电缆提供+5V 或+3V 电压，此电缆用来为 RS—232 驱动器和接收器提供电源。

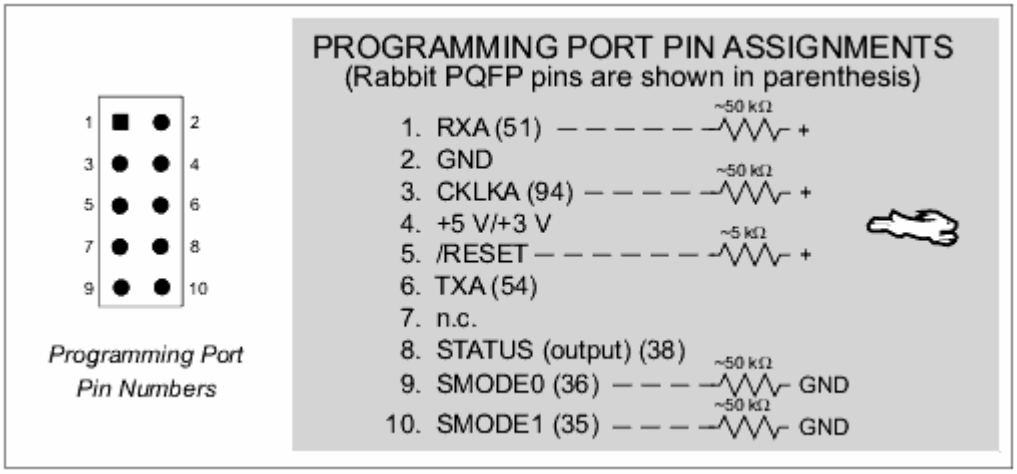


图 48 Rabbit 编程端口

A．1．1 编程端口作为诊断/配置（Diagnostic/Setup）端口使用

已经就绪的编程端口，可以作为一个方便的通信端口，用于现场配置、诊断或其他偶然的通信需求（例如，作为诊断端口）。把端口自动集成进入用户的软件方案可以有几种方法。如果端口的用途只是执行配置功能，也就是说，把配置信息写到闪存，那么可以通过编程端口复位控制器，然后执行冷启动，以开始执行专用于这个功能的特殊程序。

标准编程电缆把编程接口连接到 PC 编程端口。/RESET 线可通过操纵 PC 串行端口上的 DTR 来声称起作用，STATUS 线可由 PC 在串行端口上作为 DSR 来读取。PC 可以用脉冲式复位（pulsing reset）重启目标系统，然后，在一个短暂的延迟后，以 2400bps 发送一个特殊字符串。如果仅仅为了重新运行 BIOS，可以发送 80h，24h，80h 代码串。BIOS 启动后，它能判断编程电缆上的 PROG 连接器是否已连接，因为 SMODE0，SMODE1 引脚检测为高。这将导致 BIOS 认为它自身应该进入编程模式。接着，Dynamic C 编程模式可以有一条转义消息（escape message），它将使能串行端口的诊断功能。

使能诊断端口的另一种方法，是周期地查询串行端口看是否要开始通信，或为中断使能端口和等待。SMODEx 引脚可用来发信号，它还可以由查询检测到。然而，请记住 SMODEx 引脚在复位后有一个特殊功能，即它如果没有保持为低电平，它会禁止普通复位行为。RXA 和 CLKA 的上拉电阻阻止引脚浮动时可能发生的伪数据接收。

如果使用了定时串行模式，可以用两条触发线驱动它，这两条线具有可驱动性，其中的一条可以传感检测。这样就允许与没有异步串行端口但有两个输出信号线和一个输入信号线的器件通信。

如果冷启动模式没有使能，TXA 线（也称 PC6）在复位后是 0。检测编程端口上电缆的存在的一种可行方法，是连接 TXA 到某根 SMODE 引脚，然后在禁止冷启动模式之后提高 PC6 电位和读此 SMODE 引脚以测试连接的存在性。

A．1．2 备用编程端口

编程端口使用串行端口 A。如果用户在他的应用程序里需要使用串行端口 A，可以有一种备用编程方法，它使用同一个 10 引脚编程端口。为了让他自己的应用程序工作，用户应该使用端口 A 的备用 I/O 引脚，这些引脚与并行端口 D 共用引脚。利用并行端口 C 的引脚 6 和 7，这个 10 引脚编程端口上的 TXA 和 RXA 引脚于是成为一个并行端口输出和一个并行端口输入。利用这两个端口（即并口 C 和 D），加上 STATUS 脚作为输出时钟，用户可以创建一个用指令触发时钟和数据的同步定时通信端口。还可以用另一个基于 Rabbit 的主板，把这种定时串行信号转换成适用于 PC 的异步信号。由于目标系统控制了发送和接收的时钟，数据传送以开发中的目标系统所控制的速率进行。

本方案不允许中断，且为了此目的而使用外部中断也是不可取的。如果需要，可以在程序装载过程中使用串行端口，因为这在用户程序编译装载时间里没有冲突，在调试时才会有冲突。调试时，传送动作的本性使用户程序从某断点开始，要不就希望获取 PC 的注意。其他类型的消息的内容，是目标系统正运行时 PC 希望什么时候读或写目标系统内存。

目标系统触发时钟时，可以仅仅发送一条定时串行消息以获得 PC 的注意。中介通信板用它的定时串行端口接收这些未经请求的消息。为防止接收器超限，目标系统可在一条 SMODE 线上等待握手信号，或者会有合适的预先安排的延迟。

如果 PC 希望获得目标系统的注意，可以设置某根线来请求注意（SMODEx）。目标板在周期中断程序里检测这条线并处理这条完全消息。这会降低目标板的执行速度，但读取消息时，中断在目标系统上使能。如果消息过长会产生问题，中介板可以把长消息分割成一系列短消息

A．2 Rabbit 晶体的建议频率

表 48 提供了建议的 Rabbit 工作频率的列表。如果使用倍频器，晶体可以是工作频率的一半，最大可达约 29.5MHz。在这个工作时钟之外的频率，必须使用 X1 晶体或外部振荡器，因为振荡器波形的非对程性在时钟速度被加倍后会成为偏差。

附录 B

B．1 所有外设的控制寄存器的缺省值

所有外设控制寄存器的缺省值示于此附录。CPU 里受复位影响的寄存器，有堆栈点（SP）、程序计数器（PC）、IIR、EIR 和 IR。IP 寄存器所有位全设置为 1（即禁止所有中断），而所有其他列出的 CPU 寄存器复位时都设置为 0。

表 51 所有外设控制寄存器的缺省值

寄存器名	助记符	I/O 地址	R/W	复位
全局控制/状态寄存器	GCSR	0x0	R/W	1100 0000
实时时钟控制寄存器	RTCCR	0x1	W	0000 0000
实时时钟字节 0 寄存器	RTC0R	0x2	R/W	xxxx xxxx
实时时钟字节 1 寄存器	RTC1R	0x3	R	xxxx xxxx
实时时钟字节 2 寄存器	RTC2R	0x4	R	xxxx xxxx
实时时钟字节 3 寄存器	RTC3R	0x5	R	xxxx xxxx
实时时钟字节 4 寄存器	RTC4R	0x6	R	xxxx xxxx
实时时钟字节 5 寄存器	RTC5R	0x7	R	xxxx xxxx
看门狗定时器控制寄存器	WDTCR	0x8	W	0000 0000
看门狗定时器测试寄存器	WDTTR	0x9	W	0000 0000
全局输出控制寄存器	GOCR	0xE	W	0000 0x00
全局时钟倍加寄存器	GCDR	0xF	W	xxxx x000
MMU 指令/数据寄存器	MIMIDR	0x10	R/W	xxx0 0000
堆栈段寄存器	STACKSEG（Z180 CBR）	0x11	R/W	0000 0000
数据段寄存器	DATASEG（Z180 BBR）	0x12	R/W	0000 0000
段大小寄存器	SEGSIZE（Z180 CBAR）	0x13	R/W	1111 1111
内存体 0 控制寄存器	MB0CR	0x14	W	0000 0000
内存体 1 控制寄存器	MB1CR	0x15	W	xxxx xxxx
内存体 2 控制寄存器	MB2CR	0x16	W	xxxx xxxx
内存体 3 控制寄存器	MB3CR	0x17	W	xxxx xxxx
从端口数据 0 寄存器	SPD0R	0x20	R/W	xxxx xxxx
从端口数据 1 寄存器	SPD1R	0x21	R/W	xxxx xxxx
从端口数据 2 寄存器	SPD2R	0x22	R/W	xxxx xxxx
从端口状态寄存器	SPSR	0x23	R	0000 0000
从端口控制寄存器	SPCR	0x24	R/W	000x 0000
端口 A 数据寄存器	PADR	0x30	R/W	xxxx xxxx
端口 B 数据寄存器	PBDR	0x40	R/W	xxxx xxxx
端口 C 数据寄存器	PCDR	0x50	R/W	x0x0 x0x0
端口 C 功能寄存器	PCFR	0x55	W	x0x0 x0x0
端口 D 数据寄存器	PDDR	0x60	R/W	xxxx xxxx
端口 D 控制寄存器	PDCR	0x64	W	xx00 xx00
端口 D 功能寄存器	PDFR	0x65	W	xxxx xxxx
端口 D 驱动控制寄存器	PDDCR	0x66	W	xxxx xxxx
端口 D 数据方向寄存器	PDDDR	0x67	W	0000 0000
端口 D 位 0 寄存器	PDB0R	0x68	W	xxxx xxxx
端口 D 位 1 寄存器	PDB1R	0x69	W	xxxx xxxx
端口 D 位 2 寄存器	PDB2R	0x6A	W	xxxx xxxx
端口 D 位 3 寄存器	PDB3R	0x6C	W	xxxx xxxx

（续表）

端口 D 位 4 寄存器	PDB4R	0x6C	W	xxxx xxxx
端口 D 位 5 寄存器	PDB5R	0x6D	W	xxxx xxxx
端口 D 位 6 寄存器	PDB6R	0x6E	W	xxxx xxxx
端口 D 位 7 寄存器	PDB7R	0x6F	W	xxxx xxxx
端口 E 数据寄存器	PEDR	0x70	R/W	xxxx xxxx
端口 E 控制寄存器	PECR	0x74	W	xx00 xx00
端口 E 功能寄存器	PEFR	0x75	W	xxxx xxxx
端口 E 数据方向寄存器	PEDDR	0x77	W	0000 0000
端口 E 位 0 寄存器	PEB0R	0x78	W	xxxx xxxx
端口 E 位 1 寄存器	PEB1R	0x79	W	xxxx xxxx
端口 E 位 2 寄存器	PEB2R	0x7A	W	xxxx xxxx
端口 E 位 3 寄存器	PEB3R	0x7B	W	xxxx xxxx
端口 E 位 4 寄存器	PEB4R	0x7C	W	xxxx xxxx
端口 E 位 5 寄存器	PEB5R	0x7D	W	xxxx xxxx
端口 E 位 6 寄存器	PEB6R	0x7E	W	xxxx xxxx
端口 E 位 7 寄存器	PEB7R	0x7F	W	xxxx xxxx
I/O 体 0 控制寄存器	IB0CR	0x80	W	0000 0xxx
I/O 体 1 控制寄存器	IB1CR	0x81	W	0000 0xxx
I/O 体 2 控制寄存器	IB2CR	0x82	W	0000 0xxx
I/O 体 3 控制寄存器	IB3CR	0x83	W	0000 0xxx
I/O 体 4 控制寄存器	IB4CR	0x84	W	0000 0xxx
I/O 体 5 控制寄存器	IB5CR	0x85	W	0000 0xxx
I/O 体 6 控制寄存器	IB6CR	0x86	W	0000 0xxx
I/O 体 7 控制寄存器	IB7Cr	0x87	W	0000 0xxx
中断 0 控制寄存器	I0CR	0x98	W	xx00 0000
中断 1 控制寄存器	I1CR	0x99	W	xx00 0000
定时器 A 控制/状态寄存器	TACSR	0xA0	R/W	0000 xx00
定时器 A 控制寄存器	TACR	0xA2	W	0000 xx00
定时器 A 时间常数 1 寄存器	TAT1R	0xA3	W	xxxx xxxx
定时器 A 时间常数 4 寄存器	TAT4R	0xA9	W	xxxx xxxx
定时器 A 时间常数 5 寄存器	TAT5R	0xAB	W	xxxx xxxx
定时器 A 时间常数 6 寄存器	TAT6R	0xAD	W	xxxx xxxx
定时器 A 时间常数 7 寄存器	TAT7R	0xAF	W	xxxx xxxx
定时器 B 控制状态寄存器	TBCSR	0xB0	R/W	xxxx x000
定时器 B 控制寄存器	TBCR	0xB1	W	xxxx 0000
定时器 B MSB 1 寄存器	TBM1R	0xB2	W	xxxx xxxx
定时器 B LSB 1 寄存器	TBL1R	0xB3	W	xxxx xxxx
定时器 B MSB 2 寄存器	TBM2R	0xB4	W	xxxx xxxx
定时器 B LSB 1 寄存器	TBL2R	0xB5	W	xxxx xxxx
定时器 B 计数 MSB 寄存器	TBCMR	0xBE	R	xxxx xxxx
定时器 B 计数 LSB 寄存器	TBCLR	0xBF	R	xxxx xxxx

（续表）

串行端口 A 数据寄存器	SADR	0xC0	R/W	xxxx xxxx
串行端口 A 地址寄存器	SAAR	0xC1	W	xxxx xxxx
串行端口 A 状态寄存器	SASR	0xC3	R	0xx0 0000
串行端口 A 控制寄存器	SACR	0xC4	W	xx00 0000
串行端口 B 数据寄存器	SBDR	0xD0	R/W	xxxx xxxx
串行端口 B 地址寄存器	SBAR	0xD1	W	xxxx xxxx
串行端口 B 状态寄存器	SBSR	0xD3	R	0xx0 0000
串行端口 B 控制寄存器	SBCR	0xD4	W	xx00 0000
串行端口 C 数据寄存器	SCDR	0xE0	R/W	xxxx xxxx
串行端口 C 地址寄存器	SCAR	0xE1	W	xxxx xxxx
串行端口 C 状态寄存器	SCSR	0xE3	R	0xx0 0000
串行端口 C 控制寄存器	SCCR	0xE4	W	xx00 x000
串行端口 D 数据寄存器	SDDR	0xF0	R/W	xxxx xxxx
串行端口 D 地址寄存器	SDAR	0xF1	W	xxxx xxxx
串行端口 D 状态寄存器	SDSR	0xF3	R	0xx0 0000
串行端口 D 控制寄存器	SDCR	0xF4	W	xx00 x000