 对于中国工程师来说，利用实时Linux开发嵌入式应用程序是他们面临的困难之一，本文以RTLinux为例，并结合最为业界关注的是RTAI进行讨论，尽管这两种实现方式在句法细节上存在差异，但工作方式基本一样，因此所讲述的内容对两者都适用。 在实时任务与用户进程相互通信的过程中，有些实时应用程序无需任何用户界面即可在后台平静地运行，然而，越来越多的实时应用程序确实需要一个用户界面及其它系统功能，如文件操作或联网等，所有这些功能都必须在用户空间内运行。问题是，用户空间操作是非确定性的，而且与实时操作不兼容。 幸运的是实时Linux具有一种可在时间上减弱实时与非实时操作的机制，这种机制表现为一种称为实时FIFO的驱动程序。 当insmod将rtl_fifo.o驱动程序插入Linux内核时，该驱动程序将自己注册为RTLinux的一部分，并成为Linux驱动程序。一旦插入Linux内核，用户空间进程和实时任务都可使用实时Linux FIFO。 在深入探讨实时FIFO的细节之前，还要回顾一下实时应用程序结构的某些部分(图1)。有效的嵌入式应用程序设计方法是将实时部分与固有的非实时功能分离开来(表1)。如果应用程序的任一部分，如用户界面、图形、数据库或网络仅需软实时性能，最好是将该部分写入用户空间。然后，仅将必须满足时序要求的那部分写成实时任务。 注意，RTLinux(PSC，便携式信号编码)和RTAI(LXRT，Linux实时扩展)的最新版本已采用了一种可在用户空间执行软和硬实时

任务的方法。 任何硬实时任务都是在RTLinux的控制下运行的，该任务一般可执行周期性任务、处理中断并与I/O设备驱动程序通信，以采集或输出模拟和数字信息。当实时任务需要告诉用户进程有一个事件将发生时，它便将这一消息送给实时FIFO。每一个FIFO都是在一个方向上传送数据：从实时任务到用户空间，或反之。因此，双向通信需要使用两个FIFO。任何读出或写入实时任务一侧的操作都是非模块操作，因此rtf_put()和rtf_get()都立即返回，而不管FIFO状态是什么。 从应用程序一侧来看，FIFO就像一个常规文件。缺省情况下，RTLinux安装程序将在/dev目录下创建64个实时FIFO节点；如果需要，还必须自己创建新的节点。例如，要创建/dev/rtf80，需采用如下命令： mknod c 150 80 ； chmod 0666 /dev/rtf80 其中，150是实时FIFO主数，而80是rtf80的次数。 从用户进程的角度看，实时FIFO可执行标准文件操作。从实时任务来看，FIFO有两种通信方式：直接调用RTLinux FIFO功能，或将FIFO作为一个RTLinux设备驱动程序，并使用open()、close()、read()和write()操作。要想将FIFO作为一个设备驱动程序，就必须将rtl_conf.h中的配置变量CONFIG_RTL_POSIX_IO设定为1。 rtf_create_handler()可设置处理程序功能。每次Linux进程读或写FIFO时，rtl_fifo驱动程序都要调用该处理程序。应注意的是，该处理程序驻留在Linux内核，因此当Linux需要调用时，从该处理程序进行任何内核调用都是安全的。从该处理程序到实时任务间的最好通信方法是使用旗语或线程同步功能。最后，FIFO驱动程序还必须对内核存储器进行配置。因此，实时线程内的rtf_create()不应调用。相反，可调用init_module()中

的rtf_create()功能及cleanup_module()中的rtf_destroy()功能。例如，列表1给出了一个采用两个FIFO的简单数据采集应用程序的实时部分。两个FIFO都是在init_module()创建，并赋予minor numbers 为1和2。在调用rtf_create(minor, size)之前，该程序在已创建该FIFO的情况下调用rtf_destroy(minor)。这种情况就是另一个模块在开发过程中未被调用。然后，调用rtf_create_handler(ID, &amp.pd_do_aout)以注册带该实时FIFO的数据采集模拟输出功能pd_do_aout()。注意，创建实时线程pp_thread_ep()是因为它是周期性的，其间隔为1/100秒。 每次周期性线程得到系统控制权后，它就调用rtf_put(ID,dataptr,size)以便将数据插入minor number为2的FIFO。Linux进程打开/dev/rtf2，从实时FIFO中读取并显示所采集的数据。该进程还打开/dev/rtf1，将数据写入其它实时FIFO。当用户移动屏幕滑动器以改变模拟输出电压时，进程就向该FIFO写入一个新的值。RTLinux便调用pd_do_aout()处理程序，随后pd_do_aout()利用rtf_get()从FIFO获得值，并调用实际的硬件驱动程序以设置模拟输出的电压。可以看到，实时任务和用户进程是异步使用FIFO的。 任务间的存储器共享 FIFO为用户进程和实时任务的连接提供了一种方便的机制，但将它们作为消息队列更合适。比如，一个实时线程可利用FIFO记录测试结果，然后用户进程就可读取该结果，并将之存入数据库文件。 许多数据采集应用程序涉及到内核及用户空间之间的大量数据。Linux内核v.2.2.x并没有为这些空间的数据共享提供任何机制，但v.2.4.0版本预计会包括kiobuf结构。为解决现有稳定内核的这个缺点，RTLinux包括mbuff驱动程序。该驱动程序可利用vmalloc()分配虚拟内核存储器

的已命名存储器区域，它采用的存储器分配和页面锁定技巧跟大多数Linux中bttv帧抓取器(frame-grabber)驱动程序所用的一样。 更具体地说，mbuff一页一页地将虚拟内存锁定到实际的物理内存页面。任何实时或内核任务，或用户进程在任何时间都可访问该存储器。通过将虚拟内存页面锁定到物理内存页面，mbuff可确保所分配的页面永久驻留在物理内存，而且不会发生页面错误。换言之，当实时或内核进程访问所分配的存储器时，它可确保VMM不被调用。注意：由于实时任务执行期间实时Linux冻结标准内核的执行，任何对VMM的调用都会引起系统暂停。如果它要访问并不位于物理RAM内的虚拟存储页面，那么即使正常的Linux内核驱动程序也会引起系统故障。 由于mbuff是一种Linux驱动程序，其功能可通过设备节点/dev/mbuff实现。该节点可显示几个录入点，其中包括可将内核空间地址映射到用户空间的mmap()。它还可以利用录入点ioctl()来控制。然而，并不需要复杂的结构及直接调用ioctl。相反，mbuff可为ioctl()调用提供一个包裹，而且仅仅调用两个简单的功能即可配置和释放共享的存储缓冲器。当然，不能从实时任务调用mbuff驱动程序，因为该驱动程序所调用的虚拟存储器分配功能本身是不确定性操作。分配共享存储器所需的时间依赖于主系统的存储器容量以及CPU速度、磁盘驱动器性能和存储器分配的现有状态。因此，只能从模块的Linux内核一侧来分配共享存储器，比如从init_module()或一个ioctl()请求开始。 那么，一个共享缓冲器到底能分配多少存储器呢？如果不是任务繁重的服务器或图形应用，建议至少为Linux保留8MB存储空间。为了获得优化的配置，可在限制存储器大小的同时测量实时应用程序的

性能，以确定需要多少存储空间。 列表2给出了如何从实时任务和用户进程方面访问共享的存储器。内核模块和用户任务采用同样的功能集。当然，要想使用insmod mbuff.o，还必须将之置于Linux内核中。例如，mbuff_alloc("buf_name", size)可将符号名buf_name分配给一个缓冲器，而mbuff_free("buf_name", mbuf)可将之释放。 当第一次调用带有符号缓冲器名的mbuff_alloc()时，mbuff执行实际的存储器分配。而当从内核模块或用户进程再次调用该功能时，它只是简单地增加使用数(usage count)及将指针返回现有的缓冲器。每次调用mbuff_free()都会减少使用数，直至为零，这时mbuff就去分配带符号名的缓冲器。这种方法从多个内核模块和用户进程获得一个指向同一共享缓冲器的指针，从而解决了问题。它还可确保共享缓冲器一直有效，直到最后的应用程序释放它。请注意，是实时内核还是用户进程执行实际的buf1配置依赖于谁先获得控制权。 