

基于 $\mu\text{C}/\text{OS-II}$ 的通讯协议栈的设计方法

袁 菲, 陆 阳

(合肥工业大学 计算机与信息学院, 安徽 合肥 230009)

摘 要: 嵌入式开发已经向着嵌入式操作系统方向发展, 网络功能对于嵌入式应用也显得日益重要。然而现有的嵌入式通讯协议并不能完全满足实际应用的需求, 从头开始一个通讯协议的设计, 必然要遇到定时器管理、响应时间控制、可移植性等一系列的问题。如果在现有的嵌入式操作系统上添加通讯功能, 就可以利用操作系统提供的功能来实现通讯协议, 从而使设计的复杂性大大降低, 并减轻工作量。文中提出了一种在现有的嵌入式操作系统(RTOS) $\mu\text{C}/\text{OS-II}$ 上开发通讯协议栈的方法, 并讨论了几个重要问题的实现方法。

关键词: 嵌入式; 协议栈; $\mu\text{C}/\text{OS-II}$

中图分类号: TN919

文献标识码: A

文章编号: 1005-3751(2006)01-0176-03

Design Method of Communication Protocol Stack Based on $\mu\text{C}/\text{OS-II}$

YUAN Fei, LU Yang

(School of Computer and Information, Hefei University of Technology, Hefei 230009, China)

Abstract: The developing of embedded system has been improved by embedded operating system. More and more developers emphasize the networking ability of embedded system. However the existing embedded communication protocol stack can't satisfy the practical application completely. If develop the new community protocol from the beginning, must face obstacles: counter management, responding time, portable, for example. If develop new community protocol based on some kind of embedded OS, so can take advantages of existing embedded OS functions and this method will simplify the complexity of designing and reduce the workload. This paper proposes a method of designing protocol stack based on $\mu\text{C}/\text{OS-II}$ and discusses the implementation of several keys issues.

Key words: embedded system; protocol stack; $\mu\text{C}/\text{OS-II}$

0 引言

随着嵌入式的发展, 嵌入式设备的网络功能显得越来越重要。尽管存在各种嵌入式通讯方法及协议, 但这些协议并不能完全满足实际嵌入式应用的要求, 如通讯效率不高、代码体积庞大、功能过于复杂等等。因此, 在某些场合下就需要定制满足特定要求的通讯协议。从头构建一个网络通讯协议相对比较复杂, 工程量比较大, 而且对以后的修改和维护都会带来挑战, 移植性也成问题。通过对现有的 RTOS 进行网络功能的扩展, 既可以达到简化通讯协议设计工作的目的, 也可以为日后的移植打下一个坚实的基础。文中探讨了在 $\mu\text{C}/\text{OS-II}$ 上进行通讯协议开发的一般方法。

1 协议设计的参考模型

一般来说为了降低网络设计的复杂性, 绝大多数的网络都会组织成相互叠加的层。每层的目的都是向上一层

提供特定的服务, 而把实现这些服务的细节对上一层进行了隐藏。在每一对相邻层之间是接口。接口定义了下层向上层提供哪些原语操作和服务。发送方机器上的第 n 层与接受方机器上的第 n 层进行通讯, 通讯过程中用到的规则和约定称为第 n 层协议, 一个特定系统所使用的一组协议称为协议栈。

OSI 参考模型很好地强化了服务、接口和协议的概念, 另外由于 OSI 参考模型是在协议发明之前就产生了, 意味着 OSI 模型不偏向于任何某些特定的协议, 因此该模型具有更好的通用性。尽管 OSI 模型是 7 层, 但在实际操作时可根据需要划分层次, 以及每层应提供的功能。按照 OSI 模型的思想, 每层都应执行一个明确定义的功能。这样可以减少层与层之间传递的信息数量, 也可以很容易地在不对其它层做修改的前提下, 对某层的实现细节做出修改^[1]。

因此, 按照层次的原则设计通讯协议是必须使用的, 其设计的最终结果就是呈层次结构的协议栈。

2 在通讯协议中使用 $\mu\text{C}/\text{OS-II}$ 的功能

$\mu\text{C}/\text{OS-II}$ 是相对廉价, 并提供源码的嵌入式操作系统(RTOS)。由于其提供源码, 所以可以比较容易地对其

收稿日期: 2005-04-29

作者简介: 袁 菲(1979-), 男, 安徽合肥人, 硕士研究生, 研究方向为计算机控制; 陆 阳, 研究员, 博导, 研究方向为计算机控制、现场总线技术、嵌入式系统。

进行功能上的扩展,而且 $\mu\text{C}/\text{OS-II}$ 的移植性非常好,现在已经移植到上百种的微处理器上。 $\mu\text{C}/\text{OS-II}$ 属于实时嵌入式内核,其主要的功能是任务(进程)管理及任务间的同步和通讯。利用 $\mu\text{C}/\text{OS-II}$ 进行协议栈的开发,其好处是显而易见的:1)利用 $\mu\text{C}/\text{OS-II}$ 的功能可减小开发的工作量,如超时的管理可交由 $\mu\text{C}/\text{OS-II}$ 。2)具有较好的移植性。 $\mu\text{C}/\text{OS-II}$ 全部使用 ANSI C 写成^[2],开发协议栈时大部分编码也应用 ANSI C。3) $\mu\text{C}/\text{OS-II}$ 是实时内核,性能不亚于甚至超过某些商业内核。

2.1 网络通讯的基本过程

网络通讯的基本过程是由发送方将要发送的分组从协议栈的高层传递给低层,协议栈的每一层都会对上一层传来的数据做一定的处理,然后由协议的物理层将数据变成二进制编码发送出去。接收方从传输介质上接收到高低电平信号,由物理层将其转换成上层协议可识别的分组,每层协议都会将数据处理后传递给更高层协议^[3]。

2.2 中断及设备驱动程序

一般底层协议都是用硬件来实现的,当有数据到达的时候便会产生一个中断。这部分工作往往由设备驱动程序来处理,设备驱动程序可以独立于 $\mu\text{C}/\text{OS-II}$,可以不依赖于 $\mu\text{C}/\text{OS-II}$ 中的任何服务,但不同的设备驱动程序应向协议栈提供统一的接口,如规定一个统一的数据结构。这样可以保证在向不同目标移植时,协议栈的实现不会因底层硬件的改变而改变。这部分细节可以参考 $\mu\text{C}/\text{OS-II}$ 中关于计时器的程序。

一旦设备驱动程序接收中断,就可以通知(POST)协议栈软件有数据到达。通知方式既可以用消息队列,也可以用邮箱,甚至可以是事件标志组^[2]。但功能有所区别,消息队列中的消息实际上是指向某个数据结构的指针,因此可以附加许多关于数据的额外信息。但在中断处理程序中不应对待到来的消息做过多的处理,否则会影响响应中断的效率,所以邮箱和事件标志组都是不错的选择。

2.3 子协议的实现

子协议既可以设计成独立的任务,也可以设计成子程序。将子协议设计成独立的任务好处有:

(1) 有利于使用操作系统的功能实现协议栈各层次之间数据交换。例如,可以用消息队列或邮箱在网络层和传输层之间传递数据。

(2) 不需要像子程序那样要维护函数接口的一致性,降低了软件维护的代价。 $\mu\text{C}/\text{OS-II}$ 任务都是形如 `void task(void *pdata)` 的函数^[2]。

(3) 在 $\mu\text{C}/\text{OS-II}$ 中有超时机制,将子协议设计成独立的任务还有利于处理超时和重发。减少协议设计中的计时器管理部分的工作量。

(4) 在 $\mu\text{C}/\text{OS-II}$ 中任务在本质上就是一个特殊的子程序,但任务是否执行是由操作系统进行控制的,只有在需要它执行的时候,它才会占用 CPU 资源,如网络层的任务需要处理传输层放在消息队列中的分组。而子程序

需要在应用程序中显式的调用,使用任务方式定义子协议可以简化协议栈的结构,降低子协议之间的耦合度,强化了层次间的接口。

在 $\mu\text{C}/\text{OS-II}$ 中最多可以有 64 个任务,每个任务都有一个唯一的优先级,而系统保留和建议保留了 8 个优先级,实际可供用户使用的优先级有 56 个,对于一般的嵌入式应用是足够了。对于协议栈的设计,要根据具体的需求来设计层次数量及功能。

2.4 层次间的数据交换

在 $\mu\text{C}/\text{OS-II}$ 现有的功能中由于邮箱只支持一条消息的传递,不太适合用作任务间的通讯。一般可以选用消息队列,消息队列中可保存若干条的消息,因此还可以被用作分组的缓存。位于某一层上的协议,既可能将分组向上传递(接受)也可能向下传递(发送)。

如果将发送和接受分成两个独立的任务,必然要增加对处理器的要求。首先, $\mu\text{C}/\text{OS-II}$ 中相互独立的任务需要有相互独立的栈,而栈空间的大小是在设计时由最大所需栈空间决定的,在运行时保持不变。所以在运行时必然会浪费一定的栈空间,过多的任务会造成过多的内存浪费。其次,在对每个任务进行初始化时,每个任务对应的栈中都要保存与任务切换相关的寄存器,具体保存的数量和 CPU 有关。两个任务比一个任务要多保存一倍的寄存器。再次,将发送和接收任务分开会导致更多的任务切换,而任务切换是 $\mu\text{C}/\text{OS-II}$ 内核中最耗时的工作之一,但另一方面短小精悍的任务比一个大的任务会有更高的效率。具体如何实现效率最好需要更进一步的试验。

图 1 给出了一个接收任务的示例:

```
void LowProtlReceive(void *pdata)
{
    for (;;) {
        LowMsg = OSQPend(LowMsgQueue, 0, &err);
        HighMsg = LowPrecess(LowMsg);
        OSQPost(HighMsgQueue, (void *)&HighMsg);
        OSTimeDly(DELAY - TIME);
    }
}
```

图 1 一个接收任务的示例

在上面的这个例子中,LowProtlReceive 是个低层协议的接收任务,它从自己的消息队列(LowMsgQueue)中接收数据(LowMsg),处理后(HighMsg)挂到它上层协议的接受队列(HighMsgQueue)中。在这种情况下只要接收队列 LowMsgQueue 中没有分组需要处理,任务 LowProtlReceive 就可以一直挂起,所以超时等待设置成 0,即无限等待。

如果将发送和接受放在同一个任务中,就需要一种机制来判断从消息队列中接受到的分组是向下传递还是向上传递的。一种解决办法是将发送和接收队列相互独立,在这种情况下能很容易区分出两种传递方向。另一种解决办法是在接受和发送中都使用同一条队列,但在分组上

打上特定的标记,然后在任务中根据标记决定分组的传递方向。

图 2 给出了一个示例,发送和接收队列相互独立。

```
void LowProt1(void * pdata)
{
    for (;;) {
        /* 接收 */
        LowMsg = OSQPend(LowReceiveMsgQueue, 2, &err);
        if (err != OS_TIMEOUT) {
            HighMsg = LowProcess(LowMsg);
            OSQPost(HighReceiveMsgQueue, (void *)&HighMsg);
        }
        /* 发送 */
        HighMsg = OSQPend(LowSendMsgQueue, 2, &err);
        if (err != OS_TIMEOUT) {
            LowMsg = HighProcess(HighMsg);
            OSQPost(LowSendMsgQueue, (void *)&LowMsg);
        }
        OSTimeDly(DELAY_TIME);
    }
}
```

图 2 一个使用接收和发送队列任务的示例

在上图中,任务的基本结构与图 1 是类似的,但发送和接收之间存在影响。发送任务的等待时间会影响接收任务,同样,接收任务的等待时间也会影响发送任务。在这种情况下,超时设置不能太长,上图中发送和接收超时都设置成 2,并且在继续执行时(可能由于有消息需要处理,也可能是由于超时发生)要检查继续执行的原因。如果是超时造成的继续执行,就需要跳过处理消息的步骤。

假设在接收队列中暂时没有分组需要处理,任务就会被暂时挂起,如果此时发送队列中来了分组,那么需要发送的分组就不能得到及时的处理,除非接收队列中也来了分组或等待接收队列超时把被挂起的任务唤醒了。如果将超时时间设置的过短,就会造成频繁的任务切换,把宝贵 CPU 时间浪费在任务切换上;如果将超时时间设置的过长,又会造成发送或接收的延迟。无论是哪一种情况都不是人们所愿意看到的。

2.5 优先级设置

$\mu C/OS-II$ 是一个可抢占的(preemptive)内核,而且 CPU 总是交给当前就绪队列中优先级最高的任务。由于任务的优先级在编译时已经决定,在运行时是不能更改的,所以优先级的设定显得很重要。有一点是可以肯定

的,协议栈是作为系统内核的一部分,被应用软件所调用的,所以协议栈任务的优先级应高于应用任务^[4]。

当底层协议收到分组到来或有分组需要发送的时候应该立即执行。接收缓慢会导致上层协议“饥饿”;发送缓慢会导致上层协议拥塞。上层协议通过设置较大的消息队列来保存暂时不能被底层协议发送的分组。由此,优先级应该按照层次关系由下向上依次降低,即在 $\mu C/OS-II$ 中由下向上优先级数值依次增加。

2.6 用户接口的设计

不管什么协议栈,要求用户直接操纵协议栈中的数据结构,既难以让用户接受也不合理。对数据结构的直接操纵,需要用户了解底层的机制,这对一个协议栈的使用者来说往往是不必要的,而且也很不方便。提供一系列的 API 函数,更能为大家所接受, $\mu C/OS-II$ 正是这么做的。

用户通过类似普通函数调用的形式调用 API,在 API 函数的内部判断用户的输入情况,处理与协议栈有关的数据结构,将实现细节完全隐藏在 API 内部。这方面的设计可以参考比较成熟的 TCP/IP 协议的不同实现版本,如 Epilogue 和 BSD^[5],从不同版本的比较中能了解到有哪些因素影响移植性,哪些因素会对用户的使用造成影响。

3 结束语

简单讨论了在 $\mu C/OS-II$ 下设计协议栈的方法,讨论了一些在实现上的细节。但协议栈的设计是个系统的工作,有些方案工作未能再深入讨论下去,有些实现方案还需要通过试验才能确定最优的方法。这些工作有待进一步的研究。

参考文献:

- [1] Tanenbaum A S. 计算机网络(第 4 版)[M]. 潘爱民译. 北京:清华大学出版社, 2004. 31-40.
- [2] Labrosse J J. 嵌入式实时操作系统 $\mu C/OS-II$ (第 2 版)[M]. 邵贝贝,等译. 北京:北京航空航天大学出版社, 2003.
- [3] 雷霖. 现场总线控制网络技术[M]. 北京:电子工业出版社, 2004.
- [4] 刘鹏,张翔,戴国骏. 基于 $\mu C/OS-II$ 的嵌入式 $\mu C/OS-II$ 协议研究[J]. 杭州电子工业学院学报, 2004, 24(1): 60-63.
- [5] 史克宁. Epilogue 和 BSD 协议栈的实现机制的比较[J]. 微机发展, 2004, 14(4): 84-88.

(上接第 175 页)

参考文献:

- [1] 宋关福,钟耳顺. 组件式地理信息系统研究与开发[J]. 中国图像图形学报, 1998, 3(4): 313-316.
- [2] MapInfo Corporation. MapX Developer Guide V4. 5[M]. US: MapInfo Corporation, 2000.
- [3] 王德文,赵文清. 基于 MapX 的地理信息系统的设计与实现[J]. 微机发展, 2003, 13(5): 59-61.
- [4] 徐造林,陈一梅. 基于 GIS 的闽江航道模拟与预测系统设计[J]. 计算机工程, 2003, 29(12): 146-148.
- [5] 罗云启,罗毅. 数字化地理信息系统 MapInfo 应用大全[M]. 北京:北京希望电子出版社, 2001.