uIP　一个免费的TCP/IP栈

原文：Adam Dunkels
adam@dunkels.com
2002年2月15日
翻译：张伟林　　2003年5月17日
okelinchang@163.com

摘要

这个文档描述uIP TCP/IP栈。 uIP TCP/IP栈是使用于低至8位或16位微处理器的嵌入式系统的一个可实现的极小的TCP/IP协议栈。现时，uIP代码的大小和RAM的需求比其它一般的TCP/IP栈要小。

uIP栈使用一个基于编程模块事件去减少代码的大小和RAM的使用量。基于系统的底层和uIP之间的接口的回应会在文档里描述。系统的底层和uIP之间的接口是隐蔽的。文档后面包含了一些uIP应用编程例子。

uIP 的代码和这个文档的新版本可以在uIP的主页下载 http://dunkels.com/adam/uip/。

这个文档描述了uIP的0.6版。

1 引言

新近这些年里，人们对连接一个甚至只是小装置到一个现有的IP网络例如全球因特网的兴趣增加了。为了可以通过因特网通讯，一个可实现的TCP/IP协议栈是必须的。uIP是一个可实现的TCP/IP协议组件的一个非常重要的部分。uIP的实现目标是保持代码大小和储存器使用量最小。现时，uIP代码的大小和RAM的需求比其它一般的TCP/IP栈要小。uIP使用C编程语言，它可以自用分发和使用于商业和非商业目的。

其它的TCP/IP栈，储存器经常用于数据缓存，等待一个数据已经成功送达的确应信号。 事实上，数据包丢失了，数据必须重发。有特色的是，数据是缓存在RAM里，如果需要重发数据，应用程序可以快速重生数据。例如，一个HTTP服务器服务的大部分是ROM里的静态和半静态页，不需要在RAM里缓存静态内容。所以，如果一个包丢失了，HTTP服务器可以容易地从ROM里重生数据。数据简单地从原先的位置读回来。uIP的优越性是允许应用程序参加数据重发。

这个文档由以下部分组成，第2节描述在系统和应用的立场上怎样使用uIP。第3节详细讨论协议实现细节。第4 节覆盖了uIP的配置，第5节描述uIP的结构部分。最后，第6节提供一些uIP的应用编程实例。

2　uIP的接口技术

uIP可以看作是一个代码库为系统提供确定的函数。图 1 展示了uIP，系统底层和应用程序之间的关系。uIP提供三个函数到系统底层， uip_init()， uip_input()，和uip_periodic()。应用程序必须提供一个回应函数给uIP。当网络或定时事件发生时，调用回应函数。 uIP提供许多函数和堆栈交互。

要注意的就是uIP提供的大部分函数是作为C的宏命令实现的，主要是为了速度，代码大小，效率和堆栈的使用。

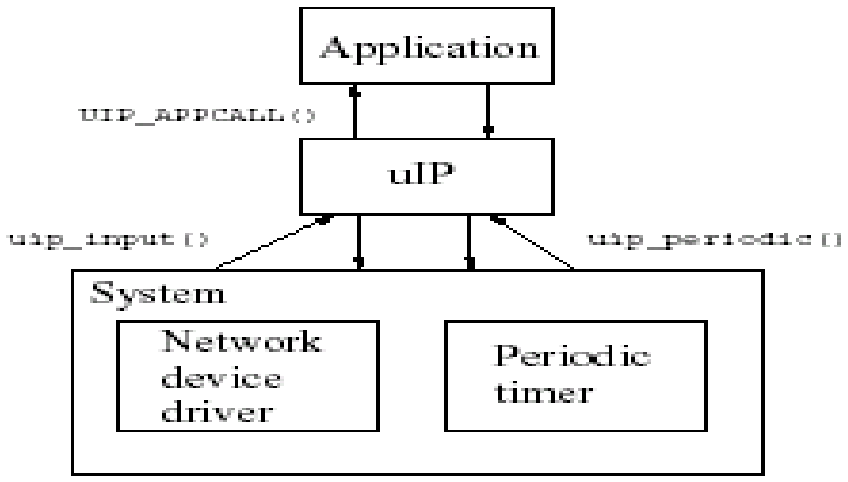

图 1　uIP 就好像一个库

2.1 uIP应用接口

BSD套节字接口使用于大部分的操作系统，它不适合微系统，因为在应用设计里，它逼使一个线程基于编程模块。一个多线程环境代价重大，因为，不但在线程管理里涉及增加代码的复杂性，而且保存每线程堆栈需要额外的储存器，还有执行任务切换的时间开销也摊派在这里。微型系统不会有足够的资源去实现一个多线程环境，因此需要这个环境的应用接口不适合uIP。

相反，uIP使用一个基于编程模块的事件，模块是实现应用程序作为一个C函数被uIP调用的地方，uIP响应一定的事件。uIP调用应用在，当接收数据时，当数据成功送达另一方中止连接时，当一个新的连接建立时，或者当数据需要重发时。 应用程序也周期性地循环等待新数据。应用程序只提供一个回应函数；它提升了应用程序处理不同的网络服务的不同的端口和连接的映射

uIP与其它TCP/IP栈不同的是，当正在重发工作，它需要应用程序的帮助。其它TCP/IP栈缓存传输数据在储存器里，直到在连接的最后数据确应成功发送。如果数据需要重传，堆栈在没有通知应用程序下监视着重传工作。通过这种方法，当要等待一个确应，数据必须缓存在储存器里，如果产生一个重发，应用程序可以快速重新生成数据。为了减少储存器的使用量，uIP利用的论据是应用程序可以重新生成发送的数据和让应用程序参加重发。

2.1.1 uIP应用事件

应用程序必须作为C函数去实现，uIP在任何一个事件发生时调用UIP_APPCALL()。表 1 列出可能的事件和每个事件的对应测试函数。测试函数用于区别不同的事件。函数是作为C宏命令实现的，将会是零值或非零值。注意的是某些函数可以在互相连接时发生(也就是新数据可以在数据确应的同时到达)。

表 1: uIP应用事件和对应的测试参数

一个数据包到达，确应先前发送到数据	uip_acked()
应用程序的新数据包已经到达	uip_newdata()
一个远程主机连接到监听端口	uip_connected()
一个到达远程主机的连接成功建立	uip_connected()
计时时间满重发	uip_rexmit()
计时时间满周期性轮询	uip_poll()
远程主机关闭连接	uip_closed()
远程主机中断连接	uip_aborted()
由于太多重传，连接中断	uip_timedout()

当应用程序调用时，uIP设置全局变量uip_conn去指向当前连接的uip_conn结构 (图 5) 。这可以用来区别不同的服务。一个典型的应用是检查uip_conn->lport (当地TCP端口号)去决定那个

服务连接应该提供。例如，如果值uip_conn->lport等于80，应用程序可以决定启动一个HTTP服务，值是23是启动TELNET服务。

2.1.2 接收数据

如果uIP测试函数uip_newdata()值为1,远程连接的主机有发送新数据。uip_appdata指针指向实际数据。数据的大小通过uIP函数uip_datalen()获得。在数据不是被缓冲后，应用程序必须立刻启动。

2.1.3 发送数据

应用程序通过使用uIP函数uip_send()发送数据。uip_send()函数采用两个参数；一个指针指向发送数据和数据的长度。如果应用程序为了产生要发送的实际数据需要RAM空间，包缓存(通过uip_appdata指针指向)可以用于这方面。

在一个时间里应用程序只能在连接中发送一块数据。因此不可以在每个应用程序启用中调用uip_send()超过一次；只有上一次调用的数据将会发出后才可以。注意，调用uip_send()以后会改变某些全局变量，在应用函数返回前它不能被调用。

2.1.4 重发数据

如果数据在网络中丢失，应用程序必须重发数据。无论数据收到或没有收到，uIP保持跟踪，和通知应用程序什么时候察觉出数据是丢失了。如果测试函数uip_rexmit()为真，应用程序要重发上一次发出的数据。重发就好像原来那样发送，也就是通过uip_send()。

2.1.5 关闭连接

应用程序通过调用uip_close()关闭当前连接。这会导致连接干净地关闭。为了指出致命的错误，应用程序可以中止连接和调用uip_abort()函数完成这个工作。

如果连接已经被远端关闭，测试函数uip_closed()为真。应用程序接着可以做一些必要的清理工作。

2.1.6 报告错误

有两个致命的错误可以发生在连接中，不是连接由远程主机中止，就是连接多次重发上一数据和被中止。uIP通过调用函数报告这些问题。应用程序使用两个测试函数uip_aborted()和uip_timedout() 去测试那些错误情况。

2.1.7 轮询

当连接空闲时，uIP 在每一个时候周期性地轮询应用程序。应用程序使用测试函数 uip_poll()去检查它是否被轮询过。

2.1.8 监听端口

uIP维持一个监听TCP端口列表。通过uip_listen()函数，一个新的监听端口打开。当一个连接请求在一个监听端口到达，uIP产生一个新的连接和调用应用程序函数。如果一个新连接产生，应用程序被调用，测试函数uip_connected()为真。

2.1.9 打开连接

作为uIP的0.6版，在uIP里面通过使用uip_connect()函数打开一个新连接。这个函数打开一个新连接到指定的IP地址和端口，返回一个新连接的指针到uip_conn结构。如果没有空余的连接槽，函数返回空值。为了方便，函数uip_ipaddr()可以用于将IP地址打包进两个单元16位数组里，通过uIP去代表IP地址。

使用两个例子,在图 2 和图 3 展示。第一个例子展示了怎样打开一个连接去远端TCP端口8080。如果没有足够的TCP连接插槽去允许一个新连接打开，uip_connect()函数返回NULL和通过uip_abort()中止当前连接。第二个例子展示怎样打开一个新连接去指定的IP地址。这例子里没有错误检查。

```
void connect_example1_app(void) {
    if(uip_connect(uip_conn->ripaddr, 8080) == NULL) {
```

```
        uip_abort();
    }
}
```

图 2:打开一个连接去当前连接的远端的端口8080

```
void connect_example2(void) {
    u16_t   ipaddr[2];
    uip_ipaddr(ipaddr, 192,168,0,1);
    uip_connect(ipaddr, 8080);
}
```

图 3: 打开一个到主机192.168.0.1上端口8080的连接

2.1.10 数据流控制

通过函数uip_stop()和uip_restart()，uIP提供存取TCP数据流的控制途径。设想一个应用程序下载数据到一个慢速设备，例如磁盘驱动器。如果磁盘驱动器的作业队列满了，应用程序不会准备从服务器接收更多的数据，直到队列排出空位。函数uip_stop()可以用于维护流控制和停止远程主机发送数据。当应用程序准备好接收更多数据，函数uip_restart()用于告知远程终端再次发送数据。函数uip_stopped()可以用于检查当前连接是否停止。

2.2 uIP/系统接口

从系统的立场看， uIP由3个C函数 uip_init(),uip_input(), 和 uip_periodic()。uip_init()函数用于初始化uIP堆栈和在系统启动期间调用。当网络设备驱动器读一个IP包到包缓存时，调用函数uip_input()。周期性运行是调用uip_periodic()，代表的是一秒一次。调用uIP函数是系统的职责。

2.2.1 uIP/设备驱动接口

当设备驱动放一个输入包在包缓存里(uip_buf)，系统应该调用uip_input()函数。函数将会处理这个包和需要时调用应用程序。当uip_input()返回，一个输出包放在包缓存里。包的大小由全局变量uip_len约束。如果uip_len是0，没有包要发送。

2.2.2 uIP/周期计时接口

周期计时是用于驱动所有uIP内部时钟事件，例如包重发。当周期计时激发，每一个TCP连接应该调用uIP函数uip_periodic()。连接编号传递是作为自变量给uip_periodic()函数的。类似于uip_input()函数，当uip_periodic()函数返回，输出的IP包要放在包缓存里。图 4 展示了调用uip_periodic()函数和监视输出包的一小段代码。在这个特别的例子，函数netdev_send()是网络驱动的部分，将uip_buf数组的目录发出到网上。

```
for(i = 0; i < UIP_CONNS; ++i) {
    uip_periodic(i);
    if(uip_len > 0)
    netdev_send();
}
```

图 4:周期计时和uIP的接口的例子代码.

2.3 uIP 函数总结

表 2 包含了所有uIP提供的函数

表 2: uIP 函数总结

系统接口	
uip_init() uip_input() uip_periodic()	初始化uIP 处理输入包 处理周期计时事件

应用程序接口	
uip_listen() uip_connect() uip_send() uip_datalen() uip_close() uip_abort() uip_stop() uip_stopped() uip_restart()	开始监听端口 连接到远程主机 在当前连接发送数据 输入数据的大小 关闭当前连接 中止当前连接 停止当前连接 查找连接是否停止 重新启动当前连接
测试函数	
uip_newdata() uip_acked() uip_connected() uip_closed() uip_aborted() uip_timeou() uip_rexmit uip_poll()	远程主机已经发出数据 确应发出的数据 当前连接刚连上 当前连接刚关闭 当前连接刚中止 当前连接刚超时 数据重发 应用程序循环运行
其它	
uip_mss() uip_ipaddr() htons(),ntohs()	获得当前连接的最大的段大小 将IP地址结构打包 在主机和网络之间转换字节次序

3 实现协议

uIP实现了TCP/IP协议组的四个基本协议； ARP [Plu82],IP [Pos81b], ICMP [Pos81a] 和 TCP [Pos81c]。链路层协议例如PPP可以实现作为uIP下面的设备驱动。应用层协议例如 HTTP, FTP 或SMTP可以实现为uIP之上的应用程序。

3.1 地址解析协议 | ARP

ARP协议映射了IP地址和以太网MAC物理地址，它在以太网上的TCP/IP操作是需要的。ARP在uIP里实现的是包含一个IP到MAC地址的映射。当一个IP包要在以太网上发出，查询ARP表，去找出包要发送去的MAC地址。如果在表里找不到IP地址，ARP请求包就会发出。请求包在网络里广播和请求给出IP地址的MAC地址。主机通过发出一个ARP回应，响应请求IP地址。当uIP给出一个ARP回应，更新ARP表。

为了节省储存器，一个IP地址的ARP请求覆盖发出的请求输出IP包。它是假定上层将重新发送那些被覆盖了的数据。

每十秒表更新一次，旧的条目会被丢弃。默认的ARP表条目生存时间是20分钟。

3.2 网际协议 | IP

uIP的IP层代码有两个职责：验证输入包的IP头的正确性和ICMP 和TCP协议之间多路复用。IP层代码是非常简单的，由9条语句组成。事实上，uIP的IP层极大地简化了，它没有实现碎片和重组。

3.3 因特网信息控制协议 | ICMP

在uIP里，只有一种ICMP信息实现了： ICMP回响信息。ICMP回响信息常常用于ping程序里

的检查主机是否在线。在uIP里，ICMP回响处理在一个非常简单的方式。ICMP类型字段的改变是从 \echo"类型到 \echo reply"类型，从而 ICMP调整校验和。其次，IP地址里的IP头交换，包发回到原先的发送者。

3.4 传输控制协议| TCP

为了减少储存器的使用，uIP里的TCP没有实现发送和接收数据的调整窗口。输入的TCP段不会通过uIP缓存，但必须立即由应用程序处理。注意这不能避免应用程序自己缓冲数据。输出数据时，uIP不能在每个连接有超过一个未解决的TCP段。

3.4.1 连接状态

在uIP，每个TCP连接的完全态包含当地和远端的TCP端口编号，远程主机的IP地址，重发时间值，上一段重发的编号，和连接的段的最大尺寸。除此之外，每个连接也可以保持一些应用状态。三个序列号是，期望接收的下一个字节的序列号，上一发送段第一字节的序列号，下一发送字节的序列号。连接的状态由uip_conn结构表现，可以在图 5 看到。一个uip_conn结构数组用于在uIP里保持所有的连接。数组的大小等于同时的最大数量的连接，它在编译时间里设置(看第 4 节)。

```
struct uip_conn {
      u8_t    tcpstateflags; /* TCP状态和标志. */
      u16_t    lport, rport; /*当地和远端端口. */
      u16_t    ripaddr[2]; /*同等远端的IP地址. */
      u8_t    rcv_nxt[4]; /*  我们期待接收的下一个序列号. */
      u8_t    snd_nxt[4]; /*上一个发送的序列号. */
      u8_t    ack_nxt[4]; /*  通过从远端的下一个应答去应答序列号. */
      u8_t    timer; /*  重发时间r. */
      u8_t    nrtx; /*计算特殊段的重发数量. */
      u8_t    mss; /*  连接的最大段大小. */
      u8_t    appstate[UIP_APPSTATE_SIZE];
};
```

图 5: uip_conn 结构

3.4.2输入处理

TCP输入处理和检验TCP校验和一起开始。如果校验和是对的，在当前活动的TCP连接之间,源、目的端口号和IP地址复用包。没有活动的连接符合输入包时，如果包不是一个监听端口的连接请求，包丢弃。如果包是一个关闭端口的请求，uIP 发一个 RST包回应。

如果发现了一个监听端口，uip_conn结构数组扫描任何一个非活动连接。如果发现一个，数组由新连接的端口号和IP地址填充。 如果连接请求携带一个TCP MSS (最大段大小)选择，它会分析，再次检查当前最大段大小MSS去决定当前连接的MSS，前后者的最小值会被选择。最后，一个回应包发去确应开启连接。

应该将输入包送去一个已经活跃的连接，包的序列号和从远端主机来的期望的下一个序列号一起被检查 (uip_conn结构里的rcv_ nxt 变量显示于图 5)。如果序列号不是期望得到的下一个，包会被丢掉和发一个ACK去指出期望得到的下一个序列号。紧接着，检查输入包里的确应号，看看是否确应连接的所有输出数据。它做了后，应用程序会知道这个事实的。

当序列号和确应号被检测过，依靠当前TCP状态，包将会被不同地处理。如果连接在SYN-RCVD状态和输入包确应先前发送的SYNACK包，连接将会输入ESTABLISHED状态，调用应用函数去通知已经完全连接。在连接的建立状态，如果有新数据由远端主机发送或者远端主机确应之前发送的数据，就调用应用函数。

当应用函数返回，TCP检查应用程序是否还有数据要发。如果有，一个TCP/IP包会形成在包

缓存里。

3.4.3 输出处理

输出处理过程比输入处理直接和简单得多。 基本上，所有TCP和IP头字段由uip_conn 结构里的值充满，计算TCP和IP的校验和。当uip_process()函数返回，包通过网络设备驱动发出去。

3.4.4 重发

当uIP通过periodic_timer被调用时，重发就进入运作(看段 2.2.2)。连接里有些特殊的数据(也就是数据发出了去但仍没有确应的)通过UIP_OUTSTANDING位在uip_conn 结构里的TCP状态标志变量标记(图 5)。那个连接，时间变量减少。当时间到达零，上一段必须重发和调用应用函数去做真正的重发。如果一个特殊段重发编号超出一个可设置的界限，连接会结束和发一个RST段到远端连接结束，调用应用函数去通知它连接超时。

3.4.5 重置TCP

TCP规格要求如果TCP头里的序列号和确应号在当前连接的接收窗口失去了，有RST (复位)标志设置的包必须断开连接。为了减少代码的大小，uIP不严格遵守这个规定。相反，如果一个有RST标志设置的包在连接里到达，连接会消灭那些不重要的序列号和确应号值。这个行为将会在将来的uIP版本修订。

4 配置 uIP

uIP的设置隐藏在一个叫uipopt.h的单独头文件里。这个文档不及包含了那些项目特性的设置选项 (例如uIP网点的IP地址和同时发生连接的最大值)，而且有结构和C编译器的特殊选项。文档是独立的和有注释说明的。

5 构做具体函数

当 IP，ICMP和TCP协议在单一的C函数里实现(uip_process()函数)，它们需要四个支持函数的帮助。支持函数实现32位添加和计算校验和。在uIP的0.4版本，支持函数的实现由实际的协议拆分,这是为了易于手工将支持函数汇编。在支持函数频繁被调用后，有实际的增益是使那些函数运行得越来越快

四个支持函数中有两个是计算IP和TCP校验和，uip_ ipchksum()，uip_tcpchksum()，另两个是执行TCP序列号的32位添加：uip_add_ack_nxt()和uip_add_rcv_ nxt()。uIP分类包括简单的支持函数C实现。

uip_ipchksum()计算和返回网间IP头的校验和[BBP88]，但没有进行检验和的位非操作。IP头可以在uip_buf数组的头20字节找到。

uip_tcpchksum()函数计算TCP校验和。TCP校验和是TCP头和数据的网间校验。这个函数事实上有点复杂，TCP头和TCP数据可以在不同的储存器位置找到。在uip_buf数组开头可以找到TCP头的20字节(也就是在&uip_buf[20]) ，uip_appdata指针指向TCP数据的开头。TCP数据的大小可以通过从整个包中减去IP和TCP头得出。包的大小包含在uip_len这个全局变量里。因为uIP不支持数据流里的IP或者TCP选择，IP和TCP头的大小是40字节。

6 应用例子

这节提供一些简单的uIP应用例子

6.1 一个简单的应用例子

第一个例子非常简单。应用程序监听输入连接的端口1234。当一个连接建立了，应用程序通过说“OK”回应所有发送给它的数据。

图 6 显示了应用程序的实现。应用程序调用example1_init()初始化，uIP的回叫函数是example1_app()，在这个例子里，可设置的变量UIP_APPCALL应该要定义在example1_app。

初始化函数调用uIP函数uip_listen()去注册一个监听端口。实际的应用函数example1_app()使用测试函数uip _newdata()和uip_rexmit()去确定为什么调用它。如果调用应用程序是因为最远端发了数据给它，它回应一个"ok"。如果调用应用函数是因为数据在网络里丢失和需要重发，它也发送一个 "ok"。

注意，这个例子显示了一个完全的uIP应用。应用程序不需要处理所有类型事件例如：uip _connected()或uip_timedout()。

```
void example1_init(void) {
  uip_listen(1234);
}
void example1_app(void) {
  if(uip_newdata() || uip_rexmit()) {
  uip_send("ok\n", 3);
}
}
```

图 6: 一个非常简单的应用程序

6.2 一个更高级的应用

第二个例子只是稍微比第一个高级一点，显示了程序中状态段怎样在uip_conn结构中使用。

这个应用程序和第一有些相似，它监听一个输入连接的端口和回应发送给它的数据一个"ok"。最大的不同是当连接建立时，这个程序打印输出一个欢迎信息"Welcome!"。

程序怎样实现，表面上是操作上的小改动产生效果上很大的不同。复杂性增加的原因是如果数据在网络里丢失，程序必须知道那个数据需要重发。如果"Welcome!"信息丢失了，程序必须重发欢迎信息，如果其中一个"ok"信息丢失，程序必须发一个新的"ok"。

程序知道只要"Welcome!"信息没有被远程主机确应，它就可能在网络里丢失了。但一旦远程主机已发了一个确应回来，程序可以应为欢迎信息已经被接收，知道一些丢失的数据是一个"ok"信息。 因此程序可以在两个之中的一个状态：在WELCOME-SENT状态，"Welcome!"已经发出但没有被确应，或者WELCOME-ACKED状态，"Welcome!"已被确应。

当远程主机连接到应用程序时，程序发一个"Welcome!"信息和发它的状态去WELCOME-SENT。当欢迎信息被确应了，程序进入WELCOME-ACKED状态。如果程序从远程主机收到任何新数据，它回应一个"ok" 。

如果应用程序被请求重发上一条信息，它看程序在那个状态。如果程序在WELCOME-SENT状态，它知道先前的欢迎信息没有被确应，它再发一个"Welcome!"信息。如果程序在WELCOME-ACKED状态，它知道上一条信息是 "ok" ，只发一条信息。

应用的实现可以在图 7看到。图 8显示了应用的设置

```
struct    example2_state {
  enum {WELCOME_SENT, WELCOME_ACKED} state;
    };
void example2_init(void) {
  uip_listen(2345);
    }
void example2_app(void) {
  struct    example2_state *s;
  s = (struct    example2_state *)uip_conn->appstate;
  if(uip_connected()) {
    s->state = WELCOME_SENT;
```

```
      uip_send("Welcome!\n", 9);
      return;
      }
   if(uip_acked() && s->state == WELCOME_SENT) {
      s->state = WELCOME_ACKED;
      }
   if(uip_newdata()) {
      uip_send("ok\n", 3);
      }
   if(uip_rexmit()) {
      switch(s->state) {
         case WELCOME_SENT:
         uip_send("Welcome!\n", 9);
         break;
         case WELCOME_ACKED:
         uip_send("ok\n", 3);
         break;
            }
      }
}
```

图 7: 一个高级应用

```
#define UIP_APPCALL example2_app
#define UIP_APPSTATE_SIZE sizeof(struct    example2_state)
```

图 8: 应用程序设置

6.3 区分应用程序

如果系统要运行多个应用程序，区分它们的其中一个技术是使用连接的远程终端或当地终端的TCP端口编号。图 9 显示了怎样将上面的例子结合在一个应用程序里。

```
void example3_init(void) {
   example1_init();
   example2_init();
}
void example3_app(void) {
   switch(uip_conn->lport) {
      case htons(1234):
      example1_app();
      break;
      case htons(2345):
      example2_app();
      break;
   }
}
```

图 9: 两个应用程序组合和使用不同的当地端口

6.4 接收大量数据

这个例子显示了一个简单的程序连接去主机，发一个文件的HTTP请求和从慢速设备如磁盘

驱动器下载文件。这个显示了怎样使用uIP的流控制函数。程序显示在图 10。

当连接已经建立，一个HTTP请求发去服务器。这是唯一的发送数据，应用程序知道是否需要重发数据，那么请求要重发。所以要合理的将两个事件结合在例子里。

当应用程序从远程主机接收数据，它通过使用函数device_enqueue()发这些数据去设备。要注意的是这个例子假设这个函数复制数据去它自己的缓存。在uip_appdata缓存的数据将会被下一个输入数据覆盖。

如果等待设备的队列满了，应用程序通过调用uIP函数uip_stop()停止来之远程主机的数据。应用程序确保它不接收任何数据，直到调用uip_restart()。应用程序轮询事件，用于检查队列是否不在满，如果没满，通过uip_restart()数据流重新开始。

```
void example4_init(void) {
  u16_t   ipaddr[2];
  uip_ipaddr(ipaddr, 192,168,0,1);
  uip_conn(ipaddr, 80);
}
void example4_app(void) {
  if(uip_connected() || uip_rexmit()) {
    uip_send("GET /file HTTP/1.0\r\nServer:192.186.0.1\r\n\r\n",48);
    return;
  }
  if(uip_newdata()) {
    device_enqueue(uip_appdata, uip_datalen());
    if(device_queue_full()) {
      uip_stop();
    }
  }
  if(uip_poll() && uip_stopped()) {
    if(!device_queue_full()) {
      uip_restart();
    }
  }
}
```

图 10: 一个接收大量数据的应用程序

6.5 一个简单的网络服务器

这个例子显示了一个非常简单的文件服务器程序和使用端口编号去决定发送那个文档。如果文档适当格式化一下，这个简单的程序可以作为一个有静态网页的网络服务器。图 11 显示了实现方法。

应用程序的状态包括一个指向要发送数据的指针和剩下发送数据的大小。当一个远程主机连接去应用程序，当地端口编号用于决定那个文档要发送。第一块数据使用uip_send()发送。由于有监管，至多的最大段字节数据发送了。

应用程序由肯定应答驱动。当数据被确应，新数据可以发送。如果没有数据要发送了，连接使用uip_close()关闭。

```
struct example5_state {
  char    *adaptor;
  unsigned    int    dataleft;
```

```
};
void example5_init(void) {
   uip_listen(80);
   uip_listen(81);
}
void example5_app(void) {
   struct    example5_state *s;
   s = (struct    example5_state)uip_conn->appstate;
   if(uip_connected()) {
      switch(uip_conn->lport) {
         case htons(80):
         s->dataptr = data_port_80;
         s->dataleft = datalen_port_80;
         break;
         case htons(81):
         s->dataptr = data_port_81;
         s->dataleft = datalen_port_81;
         break;
      }
      uip_send(s->dataptr , uip_mss() < s->dataleft? uip_mss(): s->dataleft);
      return;
   }
   if(uip_acked()) {
      if(s->dataleft < uip_mss()) {
         uip_close();
         return;
      }
      s->dataptr += uip_mss();
      s->dataleft -= uip_mss();
      uip_send(s->dataptr , uip_mss() < s->dataleft? uip_mss(): s->dataleft);
   }
}
```

图 11: 一个简单的文件服务器

参考

[BBP88] R. Braden, D. Borman, and C. Partridge. Computing the internet checksum. RFC 1071, Internet Engineering Task Force, September 1988.

[Plu82] D. C. Plummer. An ethernet address resolution protocol. RFC 826,Internet Engineering Task Force, November 1982.

[Pos81a] J. Postel. Internet control message protocol. RFC 792, Internet Engineering Task Force, September 1981.

[Pos81b] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[Pos81c] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.