TOPPERS中级实装讲座

（H8/3069F版：网络）

基础篇：第2天

TOPPERS工程

培训工作组

本教材的使用条件

NEXCESS中级课程02
应用实时OS的软件设计技术

关于本教材的注意事项

1. 关于著作权的表述

2. 关于本资料如果有任何意见、建议、想法或疑问，请通过电子邮件与TOPPERS工程办事处进行联系。

3. 关于本资料的内容，出于调整和改善的目的，可能不预先通知的情况下进行进行内容修订。

日程表

- 第1天

1.	开发环境的确认	0.5小时
2.	基本程序设计	1.5小时
3.	ITRON规范的概要	0.5小时
4.	ITRON API解说	2.5小时
5.	碗面定时器的开发	1.0小时

- 第2天

1.	TCP/IP的概要	1.0小时
2.	ITRON TCP/IP API 规范	1.0小时
3.	TINET（TCP/IP栈）	0.5小时
4.	TCP服务器程序设计	2.0小时
5.	TCP客户端程序设计	1.0小时
6.	总结	0.5小时

TCP/IP的基础

1. TCP/IP基础
2. ITRON TCP/IP规范
3. TINET（TCP/IP栈）
4. TCP服务器程序
5. TCP客户端程序
6. 总结

网络协议

- 协议
  - 进行通信时的规范、步骤
- **TCP/IP**
  - **Internet的标准协议群**
  - **IP**
    - 没有可靠性的无连接包发送协议
  - **TCP**
    - 具有可靠性的流式发送协议
  - **安装在4.2BSD中，且迅速普及**
  - **在RFC(Request for Comments) 文件中进行规定**
    - 只保留了安装过、有实绩的部分

在网络中进行通信时，为了使要通信的数据正确地发送与接收，需要规定数据的形式、格式、意思、步骤等。这些通信时的规范以及步骤等就叫做网络协议，或叫做协议。网络协议并不是只有一种，而是根据使用目的不同存在几种规范。在协议中由特定的目的规定的协议群叫做协议组。协议组中有OSI、TCP/IP、IPX/SPX、NetBIOS/NetBEUI、AppleTalk等。

TCP/IP并不是由某个标准化组织规定的，而是以研究人员和开发人员为中心的利用者在反复的讨论和开发中不断进行各种改良并在很多机器中都进行使用的协议组。尤其是随着工作站和PC的普及，TCP/IP逐渐成为互联网的标准协议组。IP是Internet Protocol的简称，是连接了多个网络时通过网络将信息包发送给对方计算机的协议。TCP是应用程序与IP之间的介质，是用于确保送信端的数据正确到达的协议。

据说TCP/IP的普及主要是因为得到了Berkeley版UNIX的BSD UNIX的采用。

在IETF (Internet Engineering Task Force)正在对TCP/IP进行研究讨论。一旦IETF的研究讨论到了实用的阶段，就会在RFC文件中进行总结并公开。

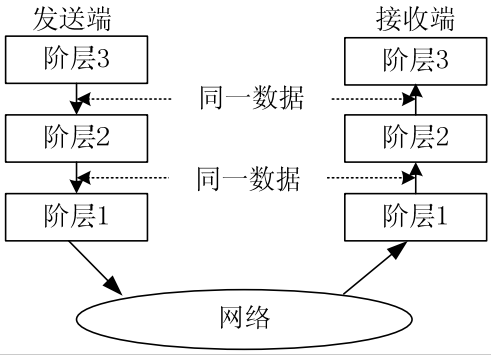
层次化的必要性

- 要完全控制一个庞大的协议是很困难的

  ➡ 在各“层”中限定并规定功能，累积各“层”使通信整体上顺利控制。

层次化原理

- 在接收方的层*n*中，接收与发送方的层*n*发送的内容完全相同的内容。



2007/01/14 TOPPERS工程認定 7

TOPPERS

在一个庞大的协议中，控制通信的规范和步骤等是很困难的。在网络的层次化模型中，OSI参考模型非常有名。OSI是将协议分为7层的通信模型。它以层为单位来制定各种作用的协议，每一层在发送方和接收方都是处理同等级别的数据。而且，越是下层越接近于硬件，越是上层越接近于应用程序。

上层	⑦应用层
	⑥显示层
	⑤会话层
下层	④传输层
	③网络层
	②数据链路层
	①物理层

TCP/IP是5层的模型。

TCP/IP互联网的协议层次为5层

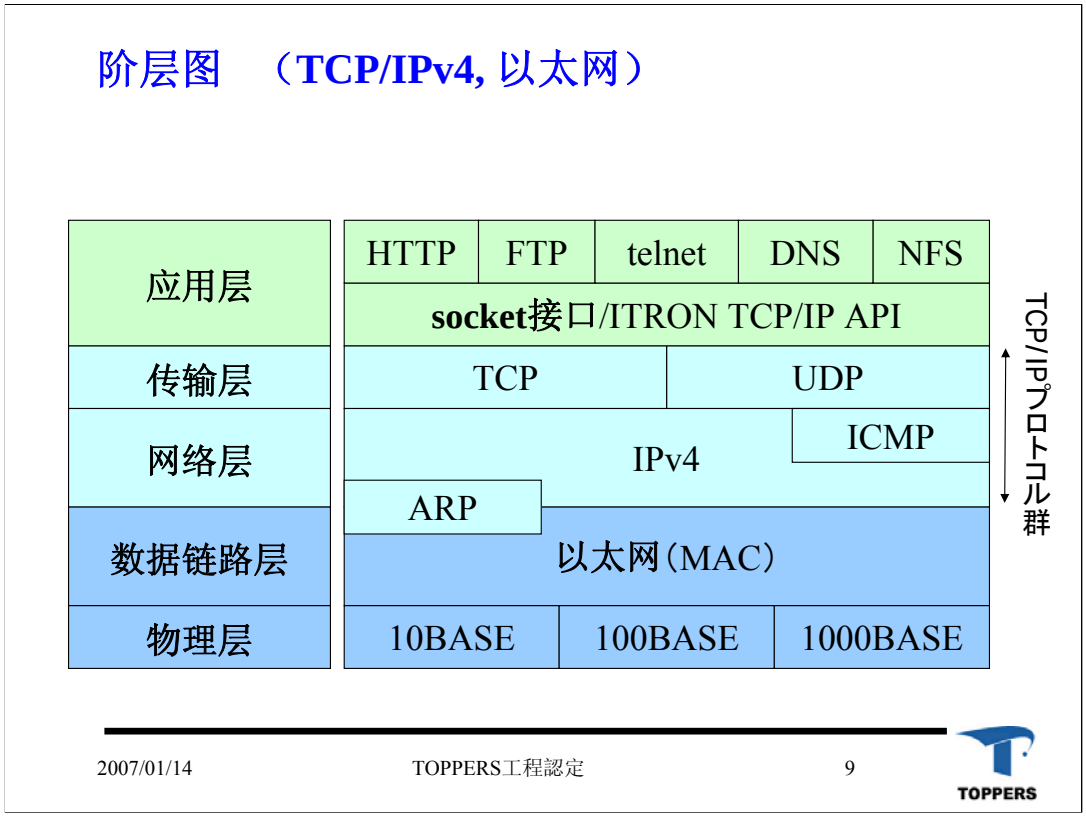
	名称	功能
5	应用层	控制各种应用程序
4	传输（TCP、UDP）层	控制通信主体的进程间的通信及确保可靠性
3	网络（IP)层	通过控制路径来控制节点间（计算机间）的通信
2	数据链路层	控制邻接结点间的通信
1	硬件层	硬件的控制

TOPPERS協会

TCP/IP协议组一般为5层。但有时也将两个数据链路层与一个硬件层作为一个网络接口层，这样就是4层。

如果要与OSI参考模型相比较进行说明的话，在数据链路层中，作为LAN广泛使用以太网，作为广域连接广泛使用应用了模拟调制解调器和ISDN的PPP。在网络层中使用IP，在传输层中使用TCP和UDP。TCP是面向连接的协议，它重视可靠性。而与其相对的UDP是以实现无连接的轻便通信为目的。

TCP/IP的应用层相当于OSI参考模型的会话层、显示层、应用层。应用层有各种协议。



上面是以IPv4的以太网为对象的断层图。

ICMP是在接收IP包时中途发生错误而不能正常接收时将错误发送给IP包的发送源的协议。ARP是在数据链路层使用以太网时将IP地址转换为以太网的MAC地址（物理地址）的协议。

MAC地址是Media Access Control Address的简称，长度为48位。该地址一般是在生产网络接口卡时写入到ROM中。MAC地址的前24位是以太网机器的厂家和经销商固有的代码，后24位是机器固有的序列号。

应用层的协议

- **HTTP : Web系统用协议**
  - **– 互联网爆炸性普及的牵动力**
- **DNS : 主机名与IP地址的解析**
- **telnet : 远程终端**
- **FTP : 文件传输**
- **SMTP, POP3 : 电子邮件**
- **NFS、SMB : 文件系统**
- **SNMP : 网络管理**

下面介绍一下应用层的主要协议。

HTTP是Hyper Text Transfer Protocol的简称，是用于要求获取WWW网页以及传输HTML文件、图像文件的协议。HTTP本身的主要任务就是要求获取并传输网页和文件等。显示等是由WWW浏览器实现的。

DNS是Domain Name System的简称。除了IP地址外，还要给PC和工作站取主机名。在一般的通信中是以主机名为对象的，所以需要一种转换主机名与IP地址的结构。像这样管理名称与编号信息的服务叫做名称服务，实现该服务的结构就是DNS。

telnet是通过TCP/IP网络远程利用其他计算机的协议。

FTP是File Transfer Protocol的简称，是用于在TCP/IP网络的计算机之间进行文件传输的协议。

SMTP是Simple Mail Transfer Protocol的简称，是用于将邮件发送到对方邮件服务器上的协议。

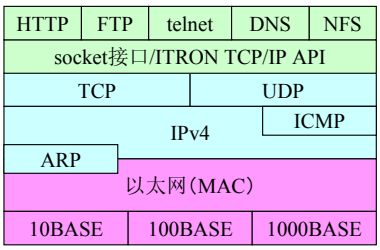
POP是Post Office Protocol的简称，是用于从邮件服务器中读取邮件的协议。

NFS是Network File System的简称，SMB是Server Message Block的简称。它们都是通过网络将其他计算机的局部存储器作为自己的的文件系统进行处理的协议。NFS主要是由Sun Microsystems公司开发的，而SMB主要是由微软公司开发的。

SNMP是Simple Network Management Protocol的简称。正如它的名字一样，它是用来管理网络上的机器的协议。

数据链路层・物理层: 以太网

- 包交换型
  - 以包为单位发送数据
- 从总线型到星型
- 从带宽共享Hub到切换Hub
- 最佳努力型（best effort）
  - 也可能被抛弃、
  - 带宽共享型也会有冲突发生
- 从10Mbps到1Gbps（10Gbps）
- 在接口分配了48位的以太网地址（MAC地址）



2007/01/14 TOPPERS工程認定 11

TOPPERS

以太网（Ethernet）是在1973年由Xerox公司的帕洛阿尔托研究所开发的，在1980年由Xerox公司、DEC公司、Intel公司共同规定为DIX以太网。后来又由IEEE规定为IEEE802.3标准。以太网的特征是通过CSMA／CD方式互相发送包。这样，如果在电缆中发生冲突就会重新发送。最初的以太网是总线型的连接方式，但在高速化的发展中已发展为星型了。

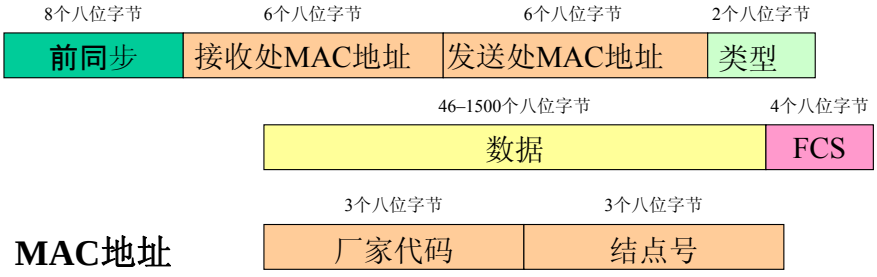
以太网的表记方法如下所示。

例如，用10Base-T的组合来表示内容。

项目	意思
传输速度	10：10Mbps
	100：100Mbps
	1000：1000Mbps(1Gbps)
传输方式	Base：基带
	Broad：宽带
传输媒体	-T：双扭线电缆
	-F：光纤
	-SX：光纤（短波长）
	-LX：光纤（长波长）
	-CX：同轴电缆

以太网帧

- 长度可变，大于等于64个八位字节，小于等于1518个八位字节
- 如果接收到帧，则根据类型选择处理的协议栈



类型

IPv4	**0x0800**
IPv6	**0x86dd**
ARP	**0x0806**

八位字节（octet）
一个八位字节相当于8位
由于没有明确定义1字节是多少位，所以作为明确表示8位的单位进行使用，用在通信领域

2007/01/14 TOPPERS工程認定 12

TOPPERS

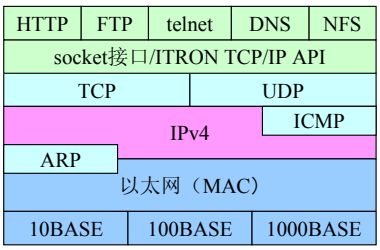
以太网上通信的数据叫做以太网帧，形式为帧格式。在通信领域内，将8位的数据叫做八位字节（单位）。有时计算机中使用的1字节并不一定是8位，所以为了明确表述而使用此单位。

以太网的帧是大于等于64个八位字节且小于等于1518个八位字节的长度可变的帧。

上述内容是DIX以太网的帧格式，在IEEE802.3中作为类型使用的部分是长度的设定。前同步信号是用于表示接在下面的信号是帧的标识信号。它后面有发送目的地和发送源的MAC地址，最后以叫做FCS（Frame Check sequence）的确认帧的数据是否正确的位串结束。

网络层：IP（Internet Protocol）

- 没有可靠性的无连接包发送服务
- 路径选择与中继控制
  - 路径选择表
- Datagram的分隔和再构成
- 最佳努力型（best effort）
  - 有时包也会被废弃
- 地址为32位
  - 地址枯竭问题



IP具有将要通信的数据以包为单位进行分配并发送到接收方IP地址的机器上的功能。此时，即使是在多个网络错综复杂的环境下也必须传输数据。在多个网络的传输中，也有可能包会被路由器丢掉，这叫做最佳努力型（最善努力）的协议。

为了在多个网络相互连接的环境下发送包，IP会观察IP地址的网络地址来确定应该经由的网络。此功能叫做路径选择或Rooting。路径信息记述在路径选择表中。该表也可以手动设定，但最近使用路径选择协议来自动创建路径选择表成为了主流。

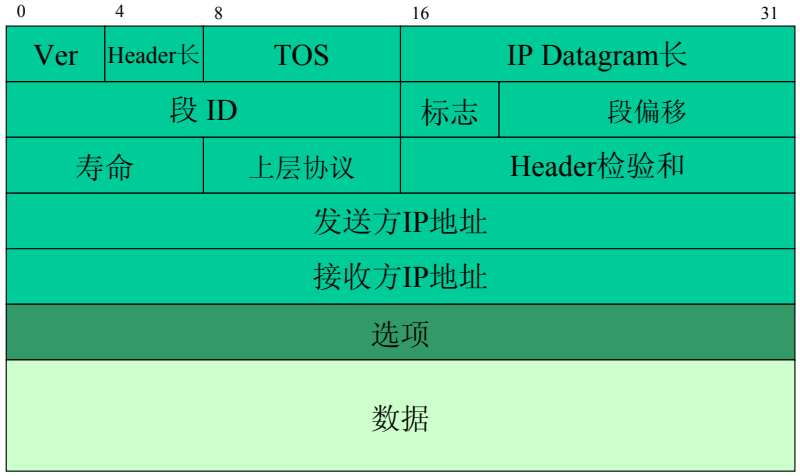
IP地址为IPv4时是32位（约43亿），这比约60亿的地球人口要少。为了有效利用，一直在想办法通过DHCP来自动分配IP地址或使用全局地址与私人地址来进行分配等。

全局地址是全世界都可以访问的地址，在互联网上不能重复。而私人地址是在相对封闭的网络中使用的地址，其设定范围如下所示。

- 10.0.0.0～10.255.255.255
- 172.16.0.0～172.31.255.255
- 192.168.0.0～192.168.255.255

IPv4 Datagram

- 数据的基本传输单位
- 由Header与数据区域构成



2007/01/14 TOPPERS工程認定 14

TOPPERS

来自于TCP、UDP的数据根据需要进行分配，并由IP作成IP包。上图是IPv4的Datagram。由IPHeader与数据构成。下面介绍一下Header的字段。

Ver：是IP的版本，为IPv4时就加一个4。

Header长：IPHeader的长度。单位是4个八位字节（32位），所以没有选项时就是5。

TOS：服务类型。由8位构成，意思如下所示：

BIT0、1、2（优先级）、BIT3（要求低延迟）、BIT4（要求高处理能力）、BIT5（要求高可靠性）、BIT6（要求低成本）、BIT7（未使用）

IPDatagram长：是包含IPHeader和数据的长度，单位为八位字节。

碎片ID：是表示多个IP包是连续数据的标识符，连续的包有相同的ID。

标志：是包分配相关的信息，各BIT都有各自的意思。虽然BIT0是未使用，但必须为0。

BIT1是能不能进一步分配的标志，而BIT2是表示是否是最后一个包的标志。

碎片偏移：是表示已进行了碎片化的IP包在原文的什么位置的信息，单位为8个八位字节。

寿命：是IP包在网络上的生存时间，单位为hops。

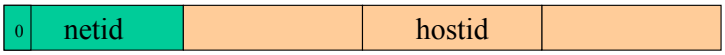
上层协议：表示TCP、UDP、ICMP等上层协议。

Header检验和：是Header的检验和值。

IPv4的Class型地址

- 对各主机分配32位的地址

类A（网络7位、主机24位）



类B（网络14位、主机16位）

类C（网络21位、主机8位）

- 类的划分是固定的，类B不足
- 地址的使用效率降低
  - 类C不足，而类B中却过多
- 与拓扑（地理位置）无关
  - 路径信息增多
- 最多约40亿

2007/01/14 TOPPERS工程認定 15

TOPPERS

对IPv4的IP地址进行了类的分配。在最初的设计中分为下述三类。

・类A　用途：大规模网络

范围：0.0.0.0～127.255.255.255

可以对应于一个网络的主机地址除了主机部分全是0和全是1的之外有16,777,214台。

・类B　用途：中规模网络

范围：128.0.0.0～191.255.255.255

网络数为16,384，可以对应于一个网络的主机地址除了主机部分全是0和全是1的之外有65.534台。

・类C　用途：小规模网络

范围：192.0.0.0～255.255.255.255

网络数为2,097,152，可以对应于一个网络的主机数为254台。

除此之外，还有作为多方向对话地址的类D，以及将来扩展用的类E。

而存在的问题就是类的分配是固定的，类B总是不足。主机地址也是类C比较少，而类B又过多。另外，IP地址与地理是没有关系的，路径信息有增多的趋势。

应用层程序的识别：端口号

IP是计算机之间的通信

- 是用来识别连接对象的应用层的应用程序的编号
  - 进程号在启动进程时进行分配，不一定总是同一个编号
  - 要与进程号不一样
- Well Known Port（服务器用）
  - 从0到1,023
  - 经常使用到的服务器应用程序的端口号如下所示
    - HTTP（80）、FTP（20和21）、SMTP（25）、DNS（53）
  - 定义文件： c:¥windows¥system32¥drivers¥etc¥services
- 一般用户用的Well Known Port
  - 从1,024到49,151
- 自动设定以及可以随便使用的范围
  - 从49,152到65,535

2007/01/14 TOPPERS工程認定 16 TOPPERS

IP地址用于指定通信对象的计算机，而端口号是用于指定计算机上的软件的编号。端口号中已注册的端口号叫做Well-Known Port。

端口	TCP/UDP	内容	端口	TCP/UDP	内容
7	TCP,UDP	回波	80	TCP	HTTP(WWW)
21	TCP	FTP	110	TCP	POP
23	TCP	TELNET	119	TCP	NNTP
25	TCP	SMTP	123	UDP	NTP
43	TCP	WHOIS	143	TCP	IMAP2
53	TCP,UDP	域	161	UDP	SNMP
67	TCP	DHCP服务器端	179	TCP	BGP
68	TCP	DHCP客户端	220	TCP	IMAP3
70	TCP	Gopher	443	TCP	HTTPS(WWW)
79	TCP	FINGER			

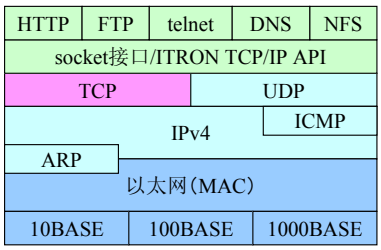
Windows用的端口号定义文件在 C:¥windows¥system32¥drivers¥etc¥services中。

端口号的分配

端口号	用途
0～1023	系统用的Well Known端口号
1024～49151	用户用的Well Known端口号
49152～65535	动态分配或私人用端口号

传输层：TCP（Transmission Control Protocol）

- 具有可靠性的连接型协议
- 面向流
  - 发送接受数据以流的形式传送
- 全双工通信
- 错误控制（重发控制）
- 顺序控制
- 流程控制与拥挤控制
- 终点（通信对象的识别）
  - IP地址与端口号的组合
- 连接（通信路径）
  - 通过终点的组合进行识别



TOPPERS

TCP是面向连接的协议，通信时在TCP间建立连接，并提供应用程序只需对TCP间的通信路径发送信息或接收信息就可以实现通信的步骤。为了关闭通信路径要断开连接。如果使用TCP进行通信，通信路径的数据就会被作为连续的流数据进行处理。

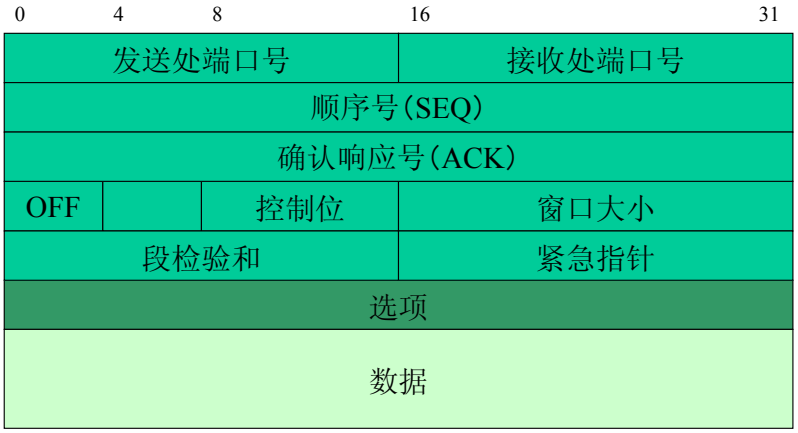
TCP进行肯定响应。发送信息后，收到了肯定响应才会发送下一个数据。不仅在数据没有到达时可以使用确认号要求重新发送，在肯定响应没有到达时也可以在等待一定时间后重新发送。

TCP的发送与接收是在发送方与接收方交换顺序号，而且TCP段每前进一步都会增加顺序号，进行顺序控制。而可以接收的数据量是通过窗口大小进行交换，进行流程控制。

在计算机上执行多个网络应用程序时，需要多个通信路径。IP地址与端口号的组合叫做终点，TCP在区分多个通信路径时就是通过发送与接收的终点的组合来进行区分的。

TCP段

- 顺序号　　：在发送字节流中的位置
- 确认响应号：期待下一个要接收的八位字节的编号



2007/01/14　　TOPPERS工程認定　　18

TOPPERS

TCP段由TCPHeader和数据构成。
发送源端口号：数据的发送源的端口号。
发送目的地端口号：数据的发送目的地的端口号。
顺序号：表示数据在发送数据流中的位置的编号。
肯定响应号：下一个想要接收的数据的的八位字节位置。在正常的发送接收中，肯定响应返回的肯定响应号与之后要发送的顺序号是一致的。
OFF：从TCPHeader的开始到数据的开始的八位字节数。
控制BIT：是表示不是普通的数据发送而是特殊的发送的BIT，有6位。

URG (Urgent)	紧急处理
ACK (Acknowledgement)	肯定响应
PSH (Push)	传输强制
RST (Reset)	虚拟通信路径的强制断开
SYN (Synchronize)	同步通信
FIN (Fin)	传输结束

窗口大小：肯定响应时下一个可以发送的数据量。
段检验和：判断TCP数据是否正确到达的检验和信息
紧急指针：控制BIT的URG为1时有效，表示从数据的开始到哪为止是紧急数据。

建确立连接

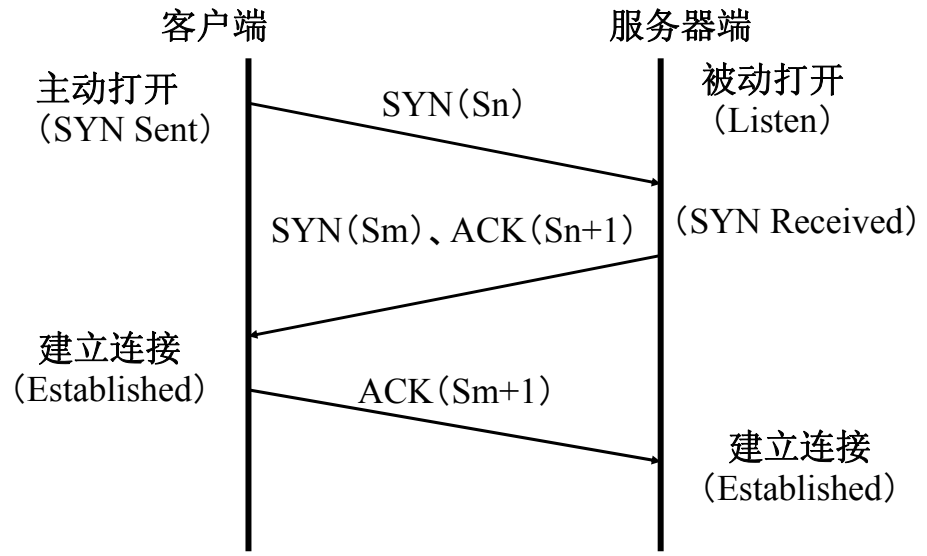
- **3way握手（信息交换）**
  - **客户端与服务器之间3次交换段**
- **被动（Passive）打开**
  - **服务器端接受请求的连接**
- **主动（Active）打开**
  - **客户端向服务器提出连接请求**
- **互相交换初始顺序号**

将TCP建立连接时的步骤称为3次握手（信息交换）。首先，主动（Active）打开方将最初的SYN段发送给被动（Passive）打开方。被动打开方接受并返回SYN＋ACK段。主动打开方接收后建立主动打开的连接。主动打开方将ACK段返回给被动打开方，然后被动打开方也建立连接。通过这3段来建立连接。（下一页）

主动打开方是客户端，被动打开方是服务器端。

最初的主动打开方的SYN段的顺序号是初始顺序号，来自于被动打开方的SYN＋ACK段的肯定响应号设置为主动打开方的初始顺序号加1，被动打开方的初始顺序号设置为顺序号。第3个的主动打开方的肯定响应号设置为主动打开方的初始顺序号加1，顺序号设置为主动打开方的初始顺序号加1。就是按照这样的顺序互相交换初始顺序号。

建立连接


客户端
服务器端
主动打开
（SYN Sent）
SYN（Sn）
被动打开
（Listen）
SYN（Sm）、ACK（Sn+1）
（SYN Received）
建立连接
（Established）
ACK（Sm+1）
建立连接
（Established）

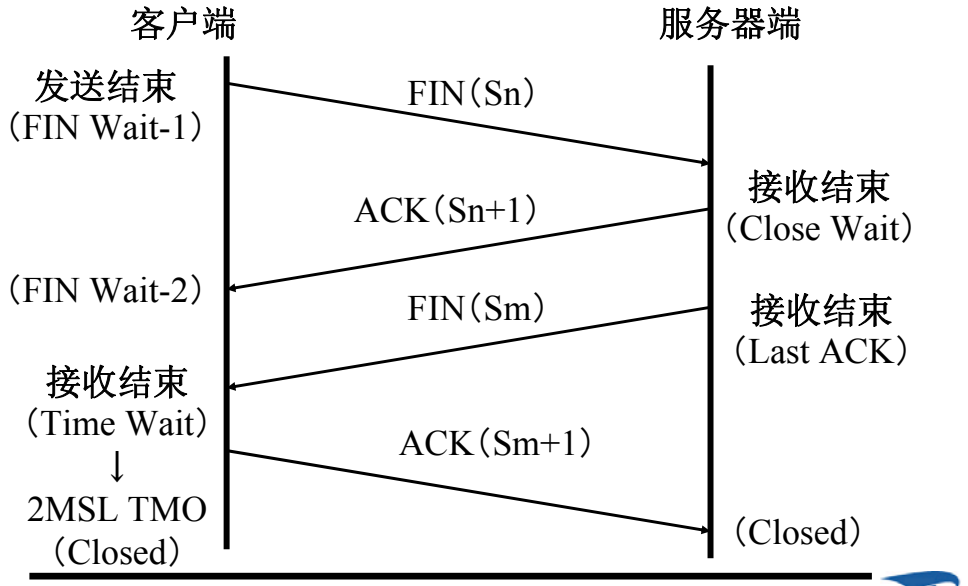
断开连接

- 通知对方无发送数据
  - 多数情况都是从客户端断开
  - WWW从服务器断开
  - 需要2MSL的超时
    - MSL（Maximum Segment Lifetime）:网络内的段的最大保留时间.RFC793中为2分钟
- 交换FIN段与ACK段
- 半关闭
  - 即使对方的发送结束了，也可以发送

连接的断开是通知对方无发送数据，并释放通信路径。是由服务器端断开还是由客户端断开取决于应用程序，但似乎多数都是由客户端断开的。请求断开的一方必须等待2MSL。MSL（Maximum Segment Lifetime）表示包在网络中能够停留的最长时间。换句话说，超过了MSL包还没有到达时，可以认为包由于某种原因丢失了。也就是说，为了断开，如果发行FIN后作为往返待机2MSL，就可以判断为已经没有接收数据。

断开的步骤是通过交换FIN段和ACK段实现的。而且并不是完全断开，有时也会出现由于没有发送数据而只是停止发送的半关闭状态。

断开连接


客户端
服务器端
发送结束
（FIN Wait-1）
FIN（Sn）
接收结束
（Close Wait）
ACK（Sn+1）
（FIN Wait-2）
FIN（Sm）
接收结束
（Last ACK）
接收结束
（Time Wait）
↓
2MSL TMO
（Closed）
ACK（Sm+1）
（Closed）

高可靠性通信

- 顺序号（SEQ）
  - 数据的偏移
- 肯定响应号（ACK）
  - 已接收的数据偏移加1
    也就是期待下一个要接收的数据的偏移
- 发送的同时启动重发定时器
- 重发最多12次
- 每次重发时超时值都会被设为2倍

使用TCP通信路径进行通信时，TCP块上有顺序号（SEQ）和肯定响应号（ACK）。发送方将表示在当前发送数据中的位置的偏移放入并发送出去。接收方接收后，把接收到的数据的下一个偏移放到肯定响应号中并作为肯定响应返回。如果正常进行发送接收，发送方的数据的顺序号与肯定响应的肯定响应号应该是一致的。

发送不能正常到达接收方时，肯定响应号的值比发送方的顺序号要小。发送方将顺序号返回到该位置，并重新发送数据。

发送方在进行发送的同时启动重发定时器。在即使发生了此定时器的超时也不返回肯定响应时，发送方重新发送数据。重发最多12次，每次重发时超时值都会被设为2倍。

重发超时的决定

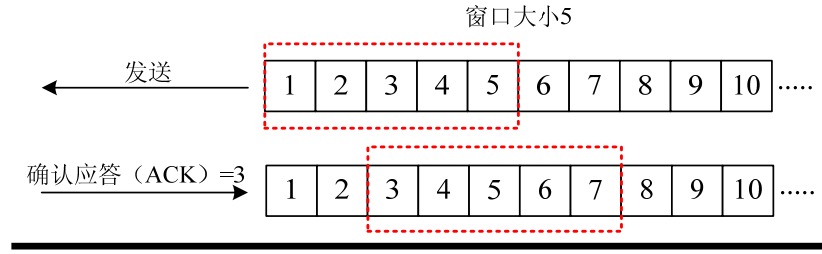
- 往返时间（RTT）的测定
  – 到对某段的SEQ返回确认响应ACK的时间

- 重发超时值（RTO）
  RTO＝α RTO＋（1－α）RTT
  – 即使RTT突然增减也不会很敏感做出反应
  – FreeBSD中实际使用的算式比较复杂

往返时间（RTT、Round Trip Time）是到对应于某段的肯定响应返回来的时间。在TCP中定期测定RTT。重发时使用的超时值叫做重发超时值（RTO、Retransmission TimeOut）。该值由RTT计算。

高效率通信与流程控制

- 滑动窗口控制
  - 接收方向发送方通知可以接收的数据量
- 最开始发送1段（1MSS）
  - MSS（MAximum Segment Size）：最大段长
- 缓慢启动
  - 每次有响应窗口大小都会增加1段
- 倍数减少拥挤回避
  - 段丢失后窗口大小减半（最小到1段）



TOPPERS

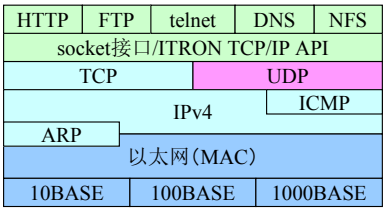
在多个通信路径中同时进行发送或接收等时候，接收方会出现无法接收所有数据的状态。因此，接收方可以使用TCPHeader的窗口大小将可以接收的数据量通知给发送方。

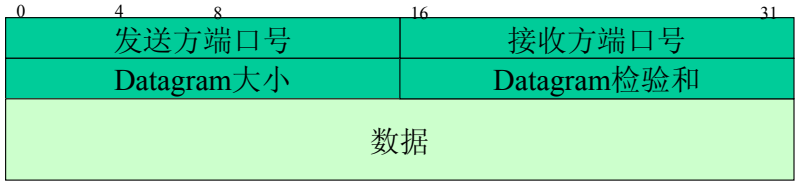
在一个TCP包中，不进行IP分配的通信的最大长度叫做MSS（Maximum Segment Size）。比如说以太网，由于它最大可以发送1500个八位字节的数据，所以除去IPHeader和TCPHeader的长度，1460个八位字节就是MSS。最开始的发送确实是MSS。先将接收用的窗口大小设定为最小，每次返回肯定响应时窗口都会增大。窗口大小是一共可以接收的总数据量，所以将多个段一起发送来减少肯定响应的方法应该可以提高发送的效率。

每当有肯定响应的时侯，窗口都会从已经接收的段向新接收的段偏移，所以叫做滑动窗口控制。（上图的说明）

传输端口层：UDP（User Datagram Protocol）

- 没有可靠性的无连接型协议
- 与IP的不同
  – 不是计算机间的通信，而是有端口号的应用程序之间的通信
- 系统开销小，速度快
- 应用层的例子
  – DNS（512个八位字节为止是UDP）
  – NFS（文件系统自己确保可靠性）
  – VoIP等流数据
- UDP Datagram





TOPPERS

UDP是无连接型的协议，不需要像TCP一样连接。所以，每次应用程序发送数据时都必须指定发送对象。UDP是有端口号的应用程序间的通信。因此，它系统开销较小，而且能够实现高速的通信。另一方面，出现网络上的数据消失或顺序替换等问题时必须由应用程序来对应。

UDP的Datagram由UDPHeader和数据构成。UDPHeader的构成如下所示：

发送源端口号：数据的发送元的端口号。

发送目的地端口号：数据的发送目的地的端口号。

Datagram大小：UDPHeader和数据的总量，为八位字节。

Datagram检验和：使用了UDP模拟Header的检验和值，是UDP中唯一确保可靠性的数据。

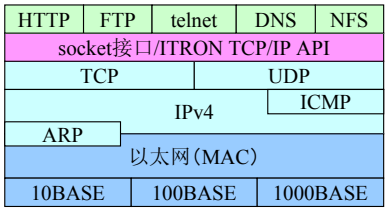
将来的网络层：IPv6

- 对应IPv4的地址枯竭问题
- IP地址为128位
  - 16E×16E 或 4G×4G×4G×4G
- IPHeader的简化
  - 固定长（40个八位字节）
  - 路由器中传输的高速化（硬件处理）
  - 扩展Header的导入
- 地址分配的自动化
- 层次化的地址分配
- 通信质量管理与安全性

IPv6是随着TCP/IP网络的扩大而诞生的拥有128位的IP地址的新IP协议。

socket接口

- 是BSD UNIX下的应用程序和TCP/IP协议间的接口。在很多OS中使用
  - Windows下为WinSock
- socket
  - 用于通信的端点
  - 不需要绑定到特定的终点地址上就可以生成

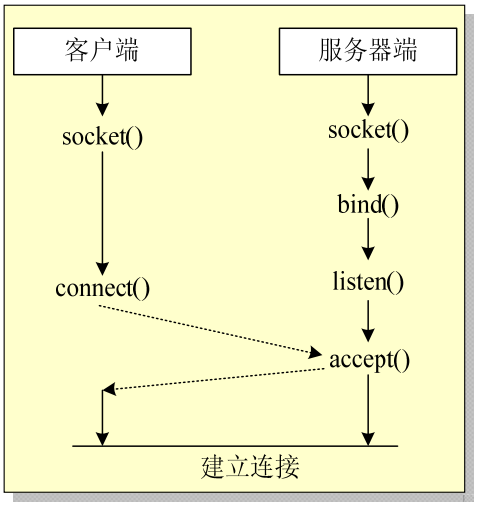


socket接口是BSD UNIX下应用程序与TCP/IP协议间的接口，在很多OS中被使用。它与Windows下的叫做WinSock的UNIX用socket接口在规范上有一些差异。

socket是将用于应用程序发送接收数据的结构进行了抽象化的对象。发送接收的结构就像打开文件读写数据一样，程序就能实现。

socket接口：API

- Socket的生成
  - socket(pf、type、protocol)
- 地址的绑定
  - bind(socket、localaddr、addrlen)
- 设置为socket被动模式
  - listen(socket、qlength)
- 接受连接
  - accpet(socket、addr、addrlen)
- 连接到服务器
  - connect(socket、destaddr、addrlen)
- 数据接收
  - readv(descriptor、buffer、length)
- 数据发送
  - write(socket、buffer、length)
- 结束
  - close(socket)



TOPPERS

下面介绍一下使用socket接口来建立连接的步骤。

服务器端通过socket()函数创建连接用的TCP socket。然后通过bind()函数给socket分配端口号。通过listen()函数进行被动打开并进入等待状态。客户端通过socket()函数生成客户端用的socket。通过connect()函数进行主动打开。通过客户端的主动打开解除服务器端的等待状态。通过accpet()函数生成通信用的socket，由该socket实现数据的发送与接收。

ITRON TCP/IP API 规范

1. TCP/IP基础
2. ITRON TCP/IP规范
3. TINET（TCP/IP栈）
4. TCP服务器端程序
5. TCP客户端程序
6. 总结

嵌入式系统与TCP/IP

- 嵌入式系统的互联网连接
  – PDA、打印机、DVD录像机、投影仪
- 互联网协议的留用
  ！并不一定是互联网协议，如果有其他合适的协议也可以用
  ➡ PC端可以使用现有的软件
  例）Web浏览器
  万能用户接口工具
  ➡ 不需要创建PC端的用户接口工具

TCP/IP变得越来越重要

ITRON TCP/IP Ver.1.00.01作为ITRON上的TCP/IP协议API，是在1998年Embedded TCP/IP技术委员会的活动结果通过了ITRON专业委员会的审查，被认证为“嵌入式系统的标准TCP/IP API”。该API是应用层与传输层之间的接口相关的标准。在认证时，作为嵌入式系统用的TCP/IP API有很多应用了UNIX用socket接口的软件部件，但是该规范是构建以ITRON为平台的小规模嵌入式系统最合适的规范。

在嵌入式系统方面，像PDA、打印机、DVD录音机、投影仪等连接到互联网或LAN上使用的机器也在逐渐增多。在这种情况下，它可以连接到目前互联网连接机器的主体PC机上，而且无需变更PC端的程序，可以利用现有的软件。这样就可以提升嵌入式系统的价值。

嵌入式系统的特性

- 很多（尤其是小规模的）嵌入式系统具有的特性
  - –要进行的处理是确定的
  - – 对硬件资源的限制很严格
  - – 要求实时性
  - – 要求高度可靠性
- 动态内存管理vs.静态内存管理
  - – 在嵌入式系统中，希望能尽量静态管理内存（多数情况都可以管理）
  - – 不得不进行动态管理时，内存管理的原则最好交给应用程序

为了考察嵌入式系统中最合适的网络接口，下面重新总结一下嵌入式系统的特性。以ITRON为对象的小规模嵌入式系统具有以下特性。

1）嵌入式机器本身具有特定的功能，一般都规定了要进行的处理。

2）产品价格差较大，对硬件资源的限制也比较严格。尤其是很多都要求内存的限制以及更高的性能。

3）需要控制要求实时性的机器。

4）形成产品的时候要求高度可靠性。

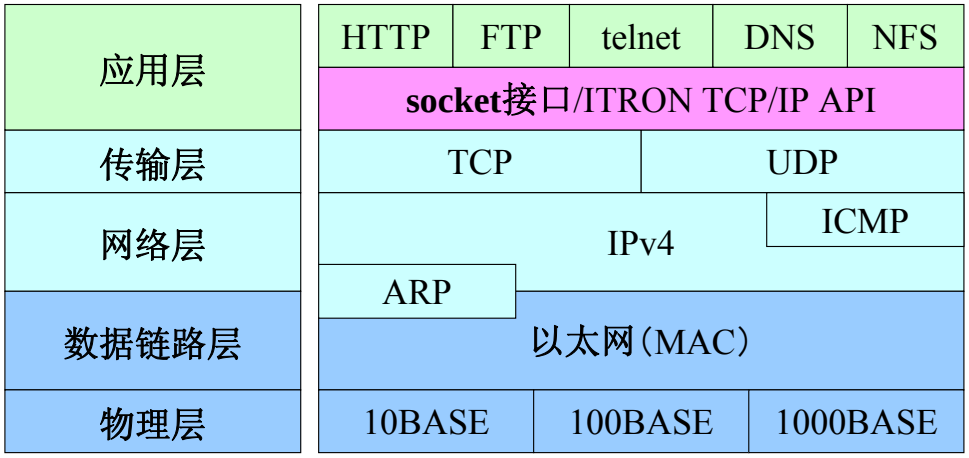
嵌入式系统的内存管理方式大致可以分为动态内存管理方式和静态内存管理方式两种。

动态内存管理是指在嵌入式系统启动后根据需要从共享内存管理空间中获取内存、释放内存来进行内存管理的方式。而与其相对的静态内存管理方式是指在进行编译、链接（创建、生成）时确定需要的内存的管理方式。以ITRON为对象的嵌入式系统多是进行静态的内存管理，这是因为其开发方式大多都是采用1链接方式。当有些系统要求必须进行动态管理时，还是将内存管理的方法交由应用程序确定的方式比较容易实现最优化。

TCP/IP协议栈的API

- **TCP和UDP的API最重要**

  ➡下面对此进行论述



TOPPERS

关于TCP/IP协议栈的各层，从物理层到传输层对应于PC端各层的功能，所以应用程序无需为了实现网络功能而特别注意。那么，对应用层与传输层之间的API、尤其是TCP/IP中与实现TCP和UDP的程序之间的API进行标准化就十分重要。

以后会继续对TCP与UDP之间的API进行讨论。

socket接口及其问题点

- 目前作为TCP与UDP的API，为BSD UNIX设计的socket接口（或者其变形）被广泛使用
- 被指出不适合（尤其是小规模的）嵌入式系统
  - 不依赖协议的通用接口
    例）指定通信对象时的地址形式
  - 必须在协议栈内进行动态内存管理
    例）UDP包的接收
  - 数据的复制次数增多
    read/write
  - RTOS的任务模型与UNIX的进程模型的不同
    fork/sElEct/单句柄

2007/01/14 TOPPERS工程認定 34

TOPPERS

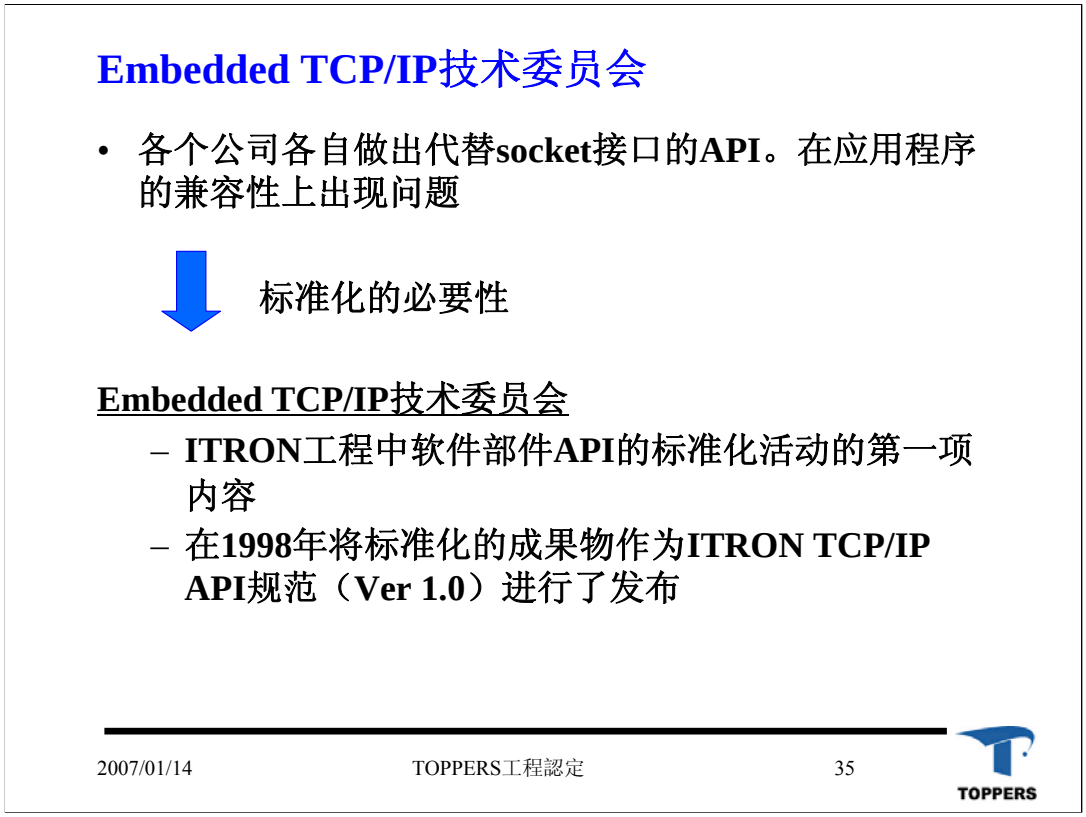
在目前的嵌入式系统中，作为TCP与UDP的API，为BSD UNIX设计的socket接口或根据RTOS将其进行最优化后的版本被广泛使用。由于socket接口是以UNIX为基本OS进行设计的，所以被指出有几点不适合于小规模的嵌入式系统。

1）在指定socket接口的地址形式时，可以按照地址族指定IP地址和端口号。而且，不管是TCP还是UDP都使用同等的函数。而在嵌入式系统中特殊化为TCP/IP的地址指定就足够了，还是根据协议进行最优化的方式比较合适。

2）在接收UDP包等时候，在协议栈内为了实现包的排队等待需要进行动态内存管理。即使是静态内存管理，也需要对协议栈能够运行的API进行扩展。

3）数据的复制次数增多。

4）由于RTOS的任务模型与UNIX的进程模型的系统调用不一样，而导致对RTOS的最优化很难实现。



在制定规范的1996年前后，开发嵌入式系统的TCP/IP的中间件的各个公司各自准备socket接口最优化的API，这就可能导致使用网络的应用程序的兼容性上出现问题。由此，从1997年开始临时组建了Embedded TCP/IP技术委员会，发起了ITRON工程中软件部件API的标准化活动。在1998年5月出台了“嵌入式系统的标准TCP/IP API规范”，并通过了ITRON专门委员会的审查，被认证为“ITRON TCP/IP API规范（Ver.1.00）”。

ITRON TCP/IP API的设计规范

（a） 基于socket接口
- 有很多程序员已适应了socket
- 想活用用socket编写的软件资源
- 在上面加载库来实现socket兼容

（b） 重视API的易懂易编程
- 尽量避免复杂或不易使用的API
- 服务调用错误详细的信息要清晰

（c） 重视硬件资源（处理器的能力、内存容量）的有效活用
- 应用程序能够控制内存不足时的动作
- 使协议栈内部动态内存管理的必要性降到最小
- 具备能够减少数据次数的省复制API

ITRON TCP/IP API的设计方针：

(a)由于很多程序员都已经适应了socket接口，而且用socket接口开发的软件数量很多，所以要基于socket
接口来作成设计规范。而且，要设定为通过在定义的API上加载库就能实现与socket接口可兼容的API。

(b)要尽量避免复杂或不易使用的API。最好能知道每个API的详细错误信息，而且设计的目标是API的易懂性以及易编程性。

(c)对硬件资源（处理器能力、内存容量）的限制很严格。所以，要将程序、数据等的大小控制到最小，这样才能完全发挥处理器的能力。缓冲器的内存空间要为最小限，应用程序要能够控制内存不足时的动作，要尽量实现在协议栈内部不需要动态内存管理。

ITRON TCP/IP API的设计规范

（d） 定义各协议最适合的API

– 分别定义TCP和UDP的API

– 也使应用程序的开发人员更容易掌握所需的资源量

（e） 考虑应用于实时性系统

– 为进入等待状态的服务调用准备同样的超时和无阻断式调用

（f） 考录在静态设置很充分的API

– “静态API”

➡ 写系统构成文件就能进行静态设定

例）以特定的端口号中进行TCP等待

（g） 沿袭ITRON规范，也适用于其他的RTOS

– 服务调用、参数、错误等的名称均遵照ITRON规范

TOPPERS

(d) 为各TCP与UDP的协议定义最合适的API。而且最好能知道各API的详细错误信息。这样应用程序的开发人员能很容易把握需要的资源量。

(e) 为进入等待状态的服务调用准备同样的超时和无阻断调用，考虑实时性系统的应用。

(f) 要对应于ITRON4.0规范的“静态API”的设定，而且静态API要有全面的API规范。

(g) 服务调用、参数、错误等的名称均遵照ITRON规范，而且也要适用于其他RTOS。

ITRON TCP/IP 的主概念

- 通信端点
  - 通过网络进行的通信服务的端点（对应socket）
  - 根据各协议的种类进行定义
    - UDP的通信端点（UDP communication end point）
    - TCP的接受窗口（TCP reception point）
    - TCP的通信端点（TCP communication end point）

      socket接口中等待TCP连接的socket不使用于数据的发送与接收
  - 各种类在整个系统中通过唯一的ID号进行识别

← (e)(g)方针

TOPPERS

通信端点（end point）对应于socket接口的socket。通信端点通过IP地址与端口号进行识别，它相当于ITRON Kernel规范的对象。通信端点根据协议或功能的不同而种类不同，在整个系统中通过唯一的ID号进行识别。

下面解释一下上述说明。通过TCP构成服务器时，虽然socket接口中准备了用于等待TCP连接的socket，但由于在连接时会生成新的socket，所以不会使用于数据的发送与接收。ITRON TCP/IP API规范中的通信端点在连接时会迁移到处于连接状态的通信端点，而且直接使用于通信。从这一点来讲，通信端点比socket更接近于实装。

ITRON TCP/IP 的主概念

- 无阻断调用
  - 服务调用中出现阻塞状况时，
    继续处理并从服务调用返回
    与UNIX的异步I/O的区别
  - 处理结束时通过回调来通知
  - 也可以中途取消处理
- 回调
  - 将来自于协议栈的事件通知给应用程序
    （不依赖OS的）方法
    - 按各通信端点在应用程序中定义
    - 在协议栈的context下执行
      –假设为无内存保护的OS
    - 有设置事件标志的使用方法

在ITRON TCP/IP API规范中，为可以进入等待状态的API准备了超时和无阻断调用。

超时是指在经过了一定的时间处理也不结束时取消处理并从API返回（此时从API返回E_TMOUT错误）。所以在发生超时时，原则是通过调用API的对象状态不发生变化。（在API的功能中，取消处理时无法返回到原来的状态时为异常。）

非阻塞调用的设定是通过在各服务调用的超时指定中设定TMO_BNLK来通知给API的。通过该设定进入API中的等待状态时，继续处理并从API返回（此时从API返回E_WBLK错误）。所以，即使从API返回也继续进行处理，在处理结束时（或取消处理时）利用回调通知应用程序处理已结束。而且，由于从API返回后也继续进行处理，所以在通过无阻断指定调用传递指向数据空间的指针的API时，在处理结束前请不要将此空间用作其他目的。

ITRON TCP/IP规范中的回调是指用于将协议栈中发生的事件通知给应用程序的不依赖于OS的手法。引起回调的事件大致可以分为无阻断调用的处理结束通知和其他事件。

回调例程由应用程序进行定义，而且在依赖于实装的context下执行。所以，最好记述为在任何context下都可以执行，而且在回调例程内不可以进入等待状态。

回调例程是对各通信端点逐一定义。引起回调的事件的种类作为参数传递给回调例程。对同一个通信端点的回调例程是嵌套的，所以不会被调用。另外，TCP接收口没有回调例程。

超时设定的详细内容请参考“ITRON TCP/IP API规范书”的1.5.3节的常数。

ITRON TCP/IP 的主概念

- 服务调用的返回值与错误代码
  - 服务调用的返回值 ← (g)方针
    - 正数或者0 …正常结束
    - 负数 …错误代码
  - 错误代码由主错误代码和子错误代码组成
    - 主错误代码 …在规范中进行标准化
    - 子错误代码 …取决于实装，假设在调试中使用 ← (d)方针
- 静态API ← (f)方针
  - 写在系统构成文件中，用于在初始化通信端点时静态生成的写法

TOPPERS

各API的返回值依照ITRON规范的规定。发生错误时返回负值的错误代码，正常执行时返回0或正值。正常执行时的返回值的意思根据各API进行定义。

错误代码由主错误代码和子错误代码组成。主错误代码和子错误代码都是负值，组合后的错误代码也是负值。主错误代码的记忆码、值以及意思与ITRON Kernel规范的错误代码一样，进行标准化。但ITRON Kernel规范中不足的错误代码（E_WBLK、E_CLS、E_BOVR）要追加定义。子错误代码取决于实装。以下错误代码可能会在所有的API中作为错误返回。

- E_SYS 系统错误（协议栈的内部错误）
- E_NOMEM 内存不足
- E_NOSPT 未支持功能
- E_MACV 内存访问错误

静态API可以静态定义接收口和通信端点。

TCP的服务调用一览

- 通信端点的生成/删除
  - TCP_CRE_REP　TCP的接收口的生成(静态API)　标准
  - tce_cre_rep　TCP接收口的生成　扩展
  - tcp_del_rep　TCP接收口的删除　扩展
  - TCP_CRE_CEP　TCP通信端点的生成（静态API）　标准
  - tcp_cre_cep　TCP通信端点的生成　扩展
  - tcp_del_cep　TCP通信端点的删除　扩展
- 连接/断开
  - tcp_acp_cep　等待要求连接（被动打开）　标准
  - tcp_con_cep　要求连接（主动打开）　标准
  - tcp_sht_cep　数据发送的结束　标准
  - tcp_cls_cep　TCP通信端点的关闭　标准
- 数据的发送接收
  - tcp_snd_dat　数据的发送　标准
  - tcp_rcv_dat　数据的接受　标准

关于ITRON TCP/IP API规范中用于控制TCP的通信端点，有等待来自对方的连接要求的接收口（TCP Reception Point，简称为rep）和作为连接端点使用的TCP通信端点（TCP Communication End Point，简称为cep）。TCP通信端点通过迁移8种状态来实现连接、数据的发送接收、结束等各种操作。

不仅有相当于socket接口的read/write处理的数据发送接收API，还有在数据发送接收时可以减少数据复制次数的效率更高的API。这种API叫做省复制API。

根据一定的标准将各种API分为标准功能和扩展功能。为了实现TCP/IP协议的功能，至少要安装标准功能。

TCP的服务调用一览

- 数据发送接收（省复制API）
  - tcp_get_buf　发送缓冲器的获取　标准
  - tcp_snd_buf　缓冲器内的数据发送　标准
  - tcp_rcv_buf　接收缓冲器的获取　标准
  - tcp_rel_buf　接收缓冲器的释放　标准
- 紧急数据的发送接收・其他服务调用
  - tcp_snd_oob　紧急数据的发送　扩展
  - tcp_rcv_oob　紧急数据的接收　扩展
  - tcp_can_cep　挂起处理的取消　标准
  - tcp_set_opt　TCP通信端点选项的设定　扩展
  - tcp_get_opt　TCP通信端点选项的读取　扩展
- 回调
  - 无阻断调用的结束　标准
  - 紧急数据的接收　扩展

客户端的连接步骤

- 与socket接口进行对比

  socket: socket → (bind) → connect →

  端点的生成→ （自节点的→连接→地址设定）

  ITRON : tcp_cre_cep → tcp_con_cep →

- 程序范例

```
/* TCP通信端点的生成（写在系统配置文件中）*/
TCP_CRE_CEP(CEPID, {0, NADR, SBUFSZ, NADR, RBUFSZ,callback});
```

```
/* 建立连接 */
dstaddr.ipaddr  = REMOTE_IPADDR;
dstaddr.portno  = REMOTE_PORTNO;
if (( ercd = tcp_con_cep(CEPID, NADR, &dstaddr, TMO_FEVR)) != E_OK){
                /* 错误处理 */
}
```

2007/01/14 TOPPERS工程認定 43

TOPPERS

通过socket接口创建TCP客户端程序时，首先要通过socket函数创建TCP socket。关于客户端程序，在IP地址单一时是可以在协议栈内自动设定的，所以不需要通过bind函数进行自节点的地址设定。通过connect函数建立与由服务器的IP地址和端口号指定的socket间的连接。

ITRON TCP/IP API规范中相当于socket函数的是tcp_cre_cep服务调用，它生成TCP通信端点。相当于bind函数和connect函数的是tcp_con_cep服务调用，它使用TCP通信端点连接对方的IP地址与端口号。

TCP_CRE_CEP TCP通信端点的生成 【标准】
tcp_cre_cep 【扩展】

【静态API】

TCP_CRE_CEP(ID cepid, {ATR cepatr, VP sbuf, INT sbufsz
VP rbuf, INT rbufsz, FP callback});

【C语言API】

ER ercd = tcp_cre_cep(ID cepid, T_TCP_CCEP *pk_ccep);

【参数】

ID	**cepid**	**TCP通信端点ID**
T_TCP_CCEP	***pk_ccep**	**TCP通信端点生成信息**

内容: TCP通信端点属性
发送用窗口缓冲器的首地址及大小
接收用窗口缓冲器的首地址及大小
回调例程的地址

2007/01/14 TOPPERS工程認定 44

TOPPERS

关于通信端点的生成，在静态API中是使用TCP_CRE_CEP服务调用，而在动态API中是使用tcp_cre_cep服务调用来实现。

pk_ccep内容如下所示：

ATR	cepatr	TCP通信端点属性 由于在API规范中没有指定内容，所以指定为0就可以
VP	sbuf	发送用窗口缓冲器的首地址
INT	sbufsz	发送用窗口缓冲器的大小
VP	rbuf	接收用窗口缓冲器的首地址
INT	rbufsz	接受用窗口缓冲器的大小
FP	callback	回调例程的地址

API规范中定义的错误代码为以下值：

E_OK	正常结束
E_ID	非法ID号
E_RSATR	预定属性
E_PAR	参数错误(Pk_ccep的地址、sbuf、sbufsz、rbuf、rbufsz、callback是非法的）
E_OBJ	对象状态错误（指定的ID的TCP通信端点已生成）

API规范中认可的实装是通过指定作为窗口缓冲器的首地址的NADR(-1)来实现在协议栈内确保缓冲器。

tcp_con_cep 连接要求（主动打开）　　【标准】

【C语言API】

ER ercd = tcp_con_cep(ID cepid, T_IPV4EP *p_myaddr,
T_IPV4EP *p_dstaddr, TMO tmout);

【参数】

ID	**cepid**	**TCP通信端点ID**
T_IPV4EP	**myaddr**	**自己的IP地址和端口号**
T_IPV4EP	**dstaddr**	**对方的IP地址和端口号**
TMP	**tmout**	**超时指定T_TCP_CCEP**

【特记事项】

将myaddr设为NADR时，自己的IP地址和端口号在协议栈内部指定（相当于在socket接口不调bind的情况。也可以单方面指定

【结构体】

```
typedef struct{ UW ipaddr;   /* IP地址 */
                UH  portno;  /*端口号 */
}T_IPV4EP
```

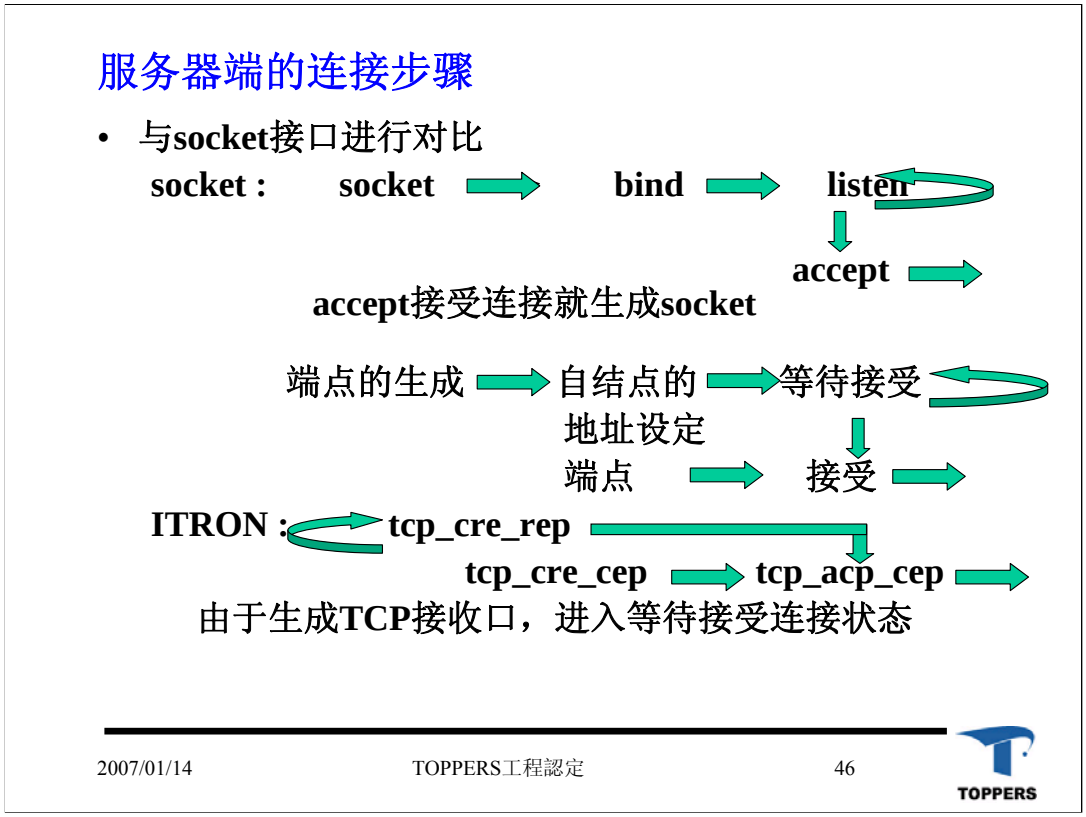
2007/01/14　　TOPPERS工程認定　　45

TOPPERS

在TCP客户端程序中，需要对服务器主动连接。在ITRONTCP/IP API规范中是使用tcp_con_cep服务调用实现此功能。tcp_con_cep服务调用使用指定的TCP通信端点连接指定的对方的IP地址和端口号，连接结束之前在服务调用内为等待状态。

当myaddr指定为NADR(-1)时IP地址和端口号在协议栈内指定。将自己的IP地址(ipaddr)指定为IPV4_ADDRANY(=0)、端口号(portno)指定为TCP_PORTANY(=0)时也是一样，会在协议栈内自动指定。

通过调用tcp_con_cep，TCP通信端点进入主动打开状态，在处理结束时变为连接状态。由于超时或tcp_can_cep取消了tcp_con_cep的处理时，TCP通信端点返回到未使用状态。



通过socket接口创建服务器程序时，使用socket函数创建TCP socket。然后使用bind函数与创建了指定的IP地址和端口号的socket建立关联。再通过listen函数许可客户端的连接要求。关于accpet函数，在连接要求到达处于Listen状态的socket的端口号之前，函数内为等待状态。一旦接收到连接要求，accpet函数就会创建新socket的描述符并返回。

在ITRON TCP/IP API规范中，通过tcp_cre_rep服务调用创建TCP接收口，并通过tcp_cre_cep服务调用创建TCP通信端点。通过tcp_acp_cep服务调用，在TCP接收口与TCP通信端点等待连接。

服务器端的连接步骤

程序例子

```
/* TCP接收口与TCP通信端点的生成（记述在系统配置文件中） */
TCP_CRE_REP(REPID, {0, {IPV4_ADDRANY, LOCAL_PRTNO}});
TCP_CRE_CEP(CEPID, {0, NADR, SBUFSZ, NADR, RBUFSZ, callback});
```

```
/* 等待接受连接 */
if ((ercd = tcp_acp_cep(CEPID, REPID, &dsaddr, TMO_FEVR) < 0){
    /* 错误处理 */
}
```

- 多任务服务器
  - –为多任务服务器时，对完全相同的程序改变通信端点ID（cepid）并使其在多个任务中运行就可以
    - TCP接收口在所有任务中是共享的
    - 在TCP接收口中形成等待连接的队列

想创建对应于多个客户端的服务器程序时，可以以多任务服务器的形式来实现。多任务服务器根据想同时接收的客户端的数量来创建TCP通信端点和任务。如果在各任务中使用tcp_acp_cep等待接收的话，就会按照接收到的TCP通信端点的顺序来解除等待状态，而且可以实现服务器功能。

TCP_CRE_REP TCP接收口的生成　　【标准】
tcp_cre_rep　　【扩展】

【静态API】

```
TCP_CRE_REP(ID repid, {ATR repatr,
                {UW myipaddr, UH  myportno});
```

【C语言API】

```
ER ercd = tcp_cre_rep(ID repid, T_TCP_CREP *pk_crep);
```

【参数】

ID	repid	TCP接收口ID
T_TCP_CREP	*pk_crep	TCP接收口信息

内容: TCP接收口属性
　　自己的IP地址和端口号

【特记事项】

– 有自动指定自己的IP地址的指定

2007/01/14　TOPPERS工程認定　48　TOPPERS

关于TCP接收口的生成，在静态API中是通过TCP_CRE_REP服务调用实现的，而在动态API中是通过tcp_cre_rep服务调用实现的。

pk_crep内容如下所示：

ATR	cepatr	TCP接收口属性 由于在API规范中没有指定内容，所以指定为0就可以
T_IPV4EPmayaddr		自己的IP地址(myipaddr)和端口号(myportno)

API规范中定义的错误代码为以下值：

E_OK	正常结束
E_ID	非法ID号
E_RSATR	预定属性
E_PAR	参数错误(pk_crep的地址、IP地址端口号是非法的）
E_OBJ	对象状态错误（指定的ID的TCP接收口已生成，端口号已使用，违反了特权端口）

tcp_cre_rep生成指定的ID的TCP接收口，并使其在指定的IP地址和端口号内进入等待状态。自己的IP地址指定为IPV4_ADDRANY(=0)时，等待接收对自己的所有IP地址的连接要求。指定了IP地址时，只接收对指定的IP地址的连接要求。

tcp_acp_cep 连接请求（被动打开） 【标准】

【C语言API】

```
ER ercd = tcp_acp_cep(ID cepid, ID repid,
                      T_IPV4EP *p_dstaddr, TMO tmout);
```

【参数】

ID	cepid	TCP通信端点ID
ID	repid	TCP接收口ID
TMP	tmout	超时指定T_TCP_CCEP

【返回参数】

T_IPV4EP	p_dstaddr	对方的IP地址和端口号

【特记事项】

- dstaddr中返回请求连接的对象的IP地址和端口号
- 多个tcp_acp_cep接收到哪个取决于实装情况

在TCP服务器程序中，需要接收来自于客户端的请求被动进行连接。在ITRONTCP/IP API规范中是使用tcp_acp_cep服务调用实现此功能，tcp_acp_cep等待对指定的TCP接收口的连接要求。有连接要求时，使用指定的TCP通信端点进行连接，并返回对方的IP地址和端口号。在连接结束前tcp_acp_cep为等待状态。

对同一个TCP接收口可以同时发行多个tcp_acp_cep。此时，哪个TCP通信端点接收连接要求取决于实装。

通过调用tcp_acp_cep，TCP通信端点进入被动打开等待状态，在处理结束时变为连接状态。由于超时或tcp_can_cep取消了tcp_acp_cep的处理时，TCP通信端点返回到未使用状态。

数据的发送接收

- 标准API
  - 与socket接口的write/read基本相同
    - 严格来讲是与异步I/O模式的write/read相同
- 省复制API
  - 能够减少数据的复制次数的高效率的API
  - 直接访问由协议栈管理的缓冲器（窗口缓冲器）
  - 没有使协议栈直接使用应用程序管理的缓冲器的方法
    - 应用程序的复杂化
    - 数据长度过短时效率提不到提升

标准的数据发送接收使用tcp_snd_dat服务调用和tcp_rcv_dat服务调用。此API使用缓冲器进行数据的发送接收，当发送空间中无可用空间或还未接收数据时为等待状态。

省复制API的数据发送与接收是提供了一种可以直接访问协议管理的发送接收缓冲器的API。通过使用该API直接访问发送接收缓冲器来提高通信效率就是省复制API的目的。省复制API可以提高通信效率，但也可能会使数据的发送接收程序变得复杂。

tcp_snd_dat 数据的发送　　【标准】

【C语言API】

```
ER ercd = tcp_snd_dat(ID cepid, VP data, INT len,
                                                TMO tmout);
```

【参数】

ID	cepid	TCP通信端点ID
VP	data	发送数据的首地址
INT	len	要发送的数据的长度
TMO	tmout	超时指定

【返回参数】

ER	ercd	放入到发送缓冲器中的数据的长度/错误代码

【特记事项】

- 发送缓冲器中即使只放入1字节也会返回
- 返回值比发送数据的长度短时，存在没有发送的数据
- 多个发送请求挂起时发生错误

tcp_snd_dat用于从指定的TCP通信端点发送数据。当数据装入到发送缓冲器时会从该API返回。当空的发送缓冲器长度比要发送的数据短时会一直放入数据，直到发送缓冲器变满，并将发送缓冲器中的数据量作为返回值返回。当发送缓冲器中没有空间时进入等待状态，一直到有空间为止。如果在有tcp_snd_dat或tcp_get_buf挂起的同一TCP通信端点上发行tcp_snd_dat，就会出现E_OBJ错误。

错误代码为以下值：

正值	正常结束（放入到发送缓冲器中的数据的长度）
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（data、len、timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接或发送结束 tcp_snd_dat、tcp_get_buf挂起）
E_WBLK	接收无阻断调用
E_TMOUT	轮询失败或超时
E_RLWAI	取消处理、强制解除等待状态
E_CLS	TCP连接被断开

tcp_rcv_dat　数据的接收　【标准】

【C语言API】

```
ER ercd = tcp_rcv_dat(ID cepid, VP data, INT len,
                                              TMO tmout);
```

【参数】

ID	cepid	TCP通信端点ID
VP	data	要放入接收数据区域的首地址
INT	len	要接收的数据的长度
TMO	tmout	超时指定

【返回参数】

ER	ercd	放入到发送缓冲器中的数据的长度/错误代码

【特记事项】

- 多个接收请求挂起时出现错误
- TCP连接异常断开后也可以取出缓冲器内的数据
- 如果由通信对方断开，返回值为0

tcp_rcv_dat用于从指定的TCP通信端点接收数据。当取出接收缓冲器中的数据时会从该API返回。当接收缓冲器中的数据长度比要接收的数据短时会一直取出数据，直到接收缓冲器变空，并将取出的数据量作为返回值返回。接收缓冲器为空时，在接收数据之前为等待状态。如果对方正常断开连接且接收缓冲器中没有数据，会从API返回0。

如果在有tcp_rcv_dat或tcp_rcv_buf挂起的同一TCP通信端点上发行tcp_rcv_dat，就会出现E_OBJ错误。

错误代码为以下值：

正值	正常结束（取出的数据的长度）
0	数据结束（连接正常断开）
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（data、len、timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接 tcp_rcv_dat、tcp_rcv_buf挂起）
E_WBLK	接收无阻断调用
E_TMOUT	轮询失败或超时
E_RLWAI	取消处理、强制结束等待状态
E_CLS	断开TCP连接、接收缓冲器为空

断开步骤

与socket接口的不同

- **close**
  - –把文件描述符和**socket**的对应分开。
    socket开始断开。断开结束前**socket**一直保留
- **tcp_cls_cep**
  - –进行通信端点的断开。在通信端点变为未使用状态之前一直阻断
- 实装方法
  - –断开结束之前一直阻断
  - –发送结束前一直阻断，之后在断开结束前另做处理，释放通信端点

socket接口的通信结束是使用close函数。通过close函数给socket加一个不能再进行发送接收的的标记，然后返回。socket开始进行断开，在断开结束前留有socket。ITRON TCP/IP API规范中通信的结束是通过tcp_cls_cep服务调用实现的。tcp_cls_cep断开TCP通信端点，等待TCP通信端子变为未使用状态，然后返回。

tcp_cls_cep 通信端点的关闭 【标准】

【C语言API】

ER ercd = tcp_cls_cep(ID cepid, TMO tmout);

【参数】

ID	cepid	TCP通信端点ID
TMO	tmout	超时指定

【特记事项】

- –发送未结束时，等待发送缓冲器中的数据发送结束，然后发送FIN并断开连接
- – 丢弃接收到的数据
- – 等待TCP通信端点变成未使用状态
- – 如果发生超时错误，就发送RST进行强制断开。此时也不会返回到原来的状态（例外的服务调用）

tcp_cls_cep进行TCP通信端点的关闭。详细的处理是：先在指定的ID的TCP通信端点还未结束通信时等待发送缓冲器中的数据发送结束，然后发送FIN并断开连接。而且此后接收到的数据要丢弃。由于是在等TCP通信端点变为未使用状态后再返回，所以从tcp_cls_cep返回后TCP通信端子可以马上进行再利用。

由于超时或tcp_can_cep取消了tcp_cls_cep的处理时，从指定的TCP通信端子发送RST来强制断开连接。不能立即发送RST时就只执行发送，如果这样也不行的话就省略RST的发送。不管怎样，如果从tcp_cls_cep返回就会出现TCP通信端点未使用状态。（指定了无阻断时，通过通知结束的回调进入未使用状态）发生超时时通过调用API使对象状态不发生变化，但tcp_cls_cep是此原则的一个例外。

通信端点与socket接口的文件描述符是不一样的，在TCP的连接状态完全结束前不能进行再利用。按照TCP/IP协议的规格，连接状态完全结束可能需要几分钟的时间。

错误代码为以下值：

E_OK	正常结束
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接）
E_WBLK	接收无阻断调用
E_TMOUT	轮询失败或者超时
E_RLWAI	取消处理、强制结束等待状态

tcp_sht_cep 数据发送的结束（半关闭） 【标准】

【C语言API】

ER ercd = tcp_sht_cep(ID cepid);

【参数】

ID	**cepid**	**TCP通信端点ID**

【特记事项】

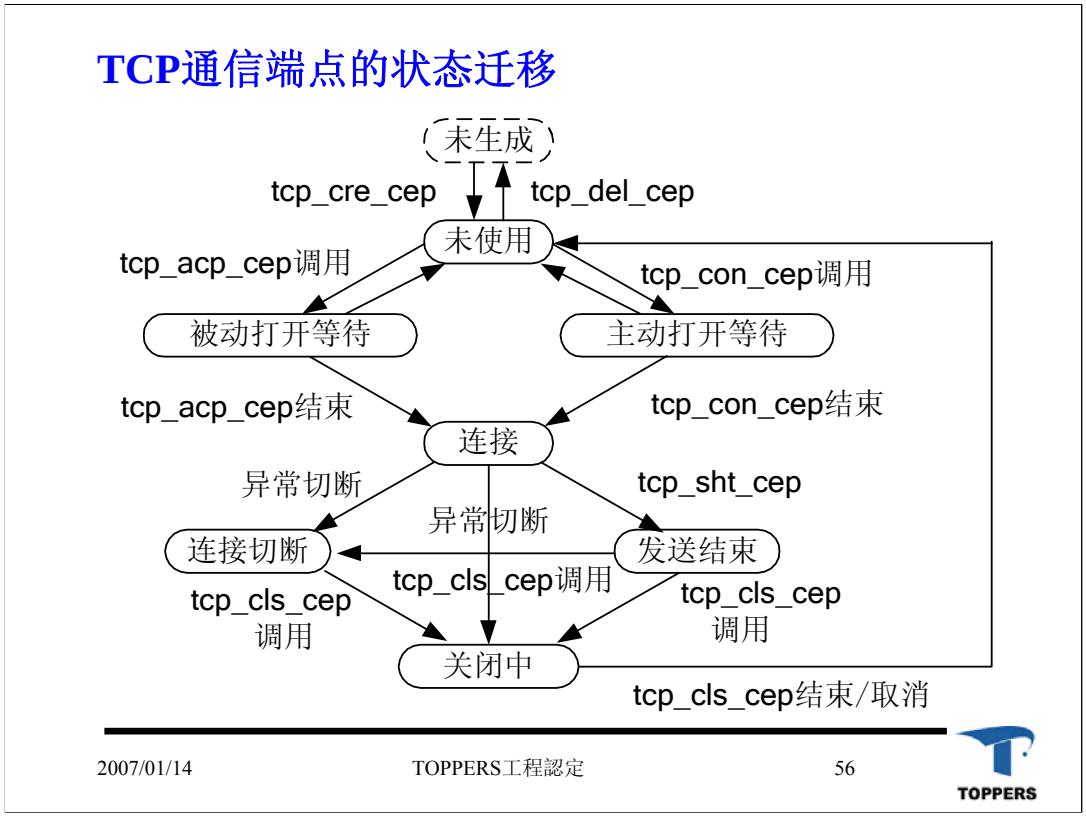
- **等到发送缓冲器中的数据发送完毕后发送FIN，开始进行连接**
- **实际上只是借助连接步骤，所以在此服务调用内不会有阻断情况**
- **等待TCP通信端点进入未使用状**
- **调用tcp_sht_cep后无法发送数据**
- **没有准备接收方的Shutdown**

tcp_sht_cep结束对指定的TCP通信端点的数据发送。具体来讲，在发送缓冲器中的数据发送结束后，发送FIN断开连接。因为tcp_sht_cep只进行断开，所以在API中不会进入等待状态。

调用tcp_sht_cep后，TCP通信端点进入发送结束状态，不能向其发送数据。如果要发送就会发生E_OBJ错误。可以接收数据。

错误代码为以下值：

正值	正常结束（放入到发送缓冲器中的数据的长度）
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（data、len、timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接）



TCP通信端点在API上有“未生成”“未使用”“等待被动打开”“等待主动打开”“连接”“发送结束”“连接断开”“关闭中”8种迁移状态。处于除“未生成”以外的7种状态中的任何一种状态时都叫“已生成”。处于除“未生成”“未使用”以外的6种状态中的任何一种状态时都叫“使用中”。处于除“连接”“发送结束”“连接断开”以外的5种状态中的任何一种状态时都叫“未连接”。

UDP的服务调用一览

- 通信端点的生成/删除
  - UDP_CRE_CEP　生成UDP通信端点（静态API）　标准
  - ucp_cre_cep　生成UDP通信端点　扩展
  - ucp_del_cep　删除UDP通信端点　扩展
- 包的发送接收
  - udp_snd_dat　UDP包的发送　标准
  - upd_rcv_dat　UDP包的接收　标准
- 其他服务调用
  - udp_can_cep　取消挂起处理　标准
  - udp_set_opt　设定UDP通信端点选项　扩展
  - ucp_get_opt　读取UDP通信端点选项　扩展
- 回调
  - 结束无阻断调用　标准
  - 只接收包（UDP）　标准

UDP（User Datagram Protocol）提供与TCP不同种类的End To End服务。UDP与TCP一样，即使没有建立连接也可以使用。而且可以从任意地址向多个地址发送数据。反方向的数据发送接收也能够实现。所以，没有必要像TCP一样创建接收口。

UDP的API中使用的通信端点有UDP通信端点（UDP Communication End Point，简称为cep）。无预先对UDP的通信端点设定对方的IP地址和端口号的功能。

发送到某个接口的广播地址的包对此接口进行广播传输。发送到255.255.255.255的包从所有支持广播传输方式的接口进行广播传输。

其他的服务调用

接收口・通信端点的删除

紧急数据的发送接收

- –紧急数据处理**out-of-band**的数据。根据实装的情况，也可以处理为**in-band**的数据
- –假定接收（**tcp_rcv_oob**）是在回调内进行调用

取消挂起中的处理

- –可以取消挂起中的处理（由于服务调用内的阻断、无阻断调用而未结束）

通信端点选项的设定/读取

- –设定/读取通信端点选项的服务调用
- –通信端点选项的使用方法取决于实装

TOPPERS

下面对TCP的其他服务调用进行说明。

・接收口・通信端点的删除

作为扩展API有删除动态API中生成的TCP接收口、TCP通信端点的服务调用。

・紧急数据的发送接收

ITRON TCP/IP API规范的标准是将紧急数据作为out-of-band数据进行处理。根据实装的情况，也可以通过设定TCP通信端点属性、TCP通信端点选项将其作为in-band的数据进行处理。假定紧急数据的接收服务调用tcp_rcv_oob是在紧急数据接收的回调例程内进行调用。

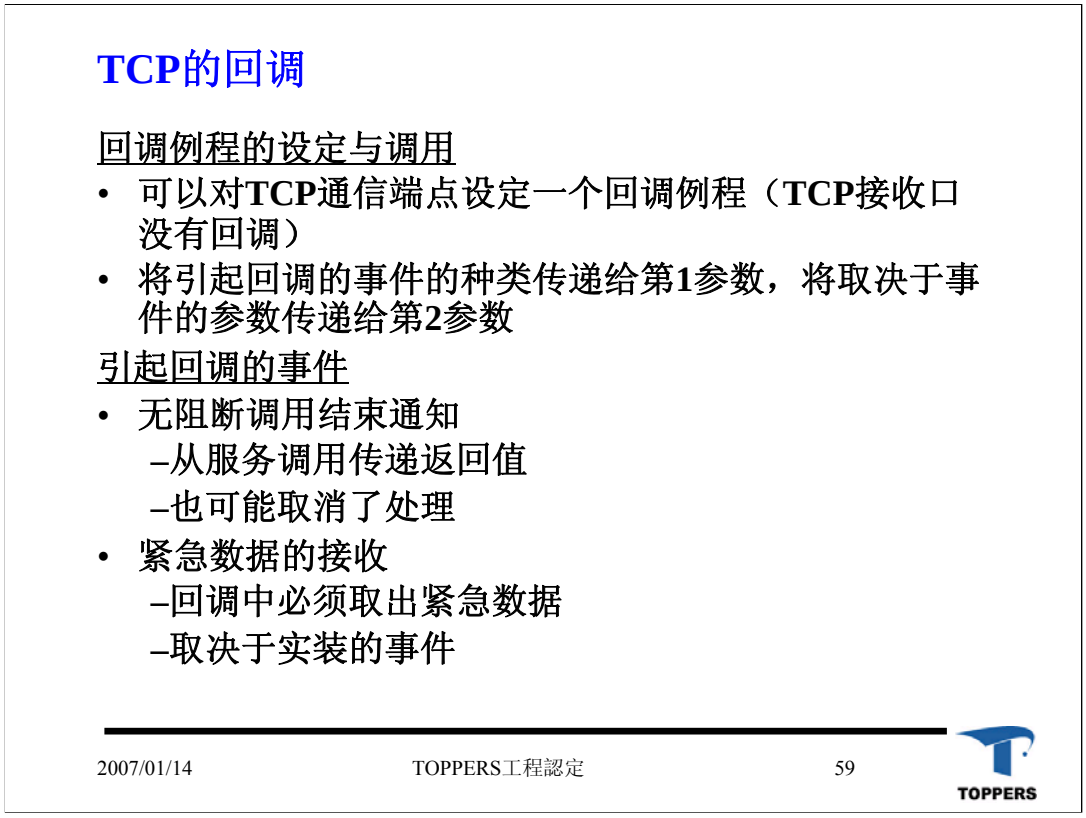
・取消挂起中的处理

tcp_can_cep取消挂起在指定TCP通信端点的指定的处理的执行。

被取消的API中返回E_RLWAI错误。取消了无阻断时，调用通知处理结束的回调例程。

・通信端点选项的设定／读取

有读取或写入TCP通信端点的选项的服务调用。TCP通信端点的选项的种类与功能取决于实装，在API规范中没有设定。



回调例程由应用程序定义，在取决于实装的context下执行。可以对TCP通信端点逐一定义。但TCP接收口没有回调例程。引起回调的事件种类作为参数传递给回调例程。

以下是回调例程的通用定义：

【C语言API】

ER ercd = callback(ID cepid, FNN fincd, VP p_parblk);

【共通参数】

ID	cepid	TCP通信端点ID
FN	fincd	事件的种类

【固有参数】

VP	p_parblk	事件中原有参数块的地址

【返回值】

取决于事件的种类。

TCP通信端点的回调事件有如下两种：

・无阻断调用的结束通知

在无阻断调用的处理已结束或已取消时进行调用。从API返回的返回值放在参数块中进行传递。从API返回的其他返回参数在调用API时保存到指定空间。

・紧急数据的接收

接收紧急数据时进行调用。在回调例程内需要使用tcp_rcv_oob取出紧急数据。没有取出时则丢弃数据。

省复制API的数据发送接收

- 假定窗口缓冲器的空间被连续获取（如果通过**tcp_cre_cep**提供窗口缓冲器空间就会如此）。这种情况下，在数据的发送接收时最少要复制一次。
- 直接在窗口缓冲器中读写的模型
- 实际上到底需要几次复制取决于使用的**LAN**控制器、协议栈的实装
  –根据条件也可能不需复制

对于TCP用API，支持通过省复制进行的数据发送接收步骤。该步骤以协议栈上的窗口缓冲器空间进行连续获取为前提。为了实现这一点，需要通过tcp_cre_cep提供窗口缓冲器空间。此时，发送接收API中至少要进行1次复制。省复制API的目的是提供可以直接读写窗口缓冲器的API。

在发送接收中到底要进行几次复制取决于LAN控制器、协议栈的实装。

tcp_get_buf 获取发送用缓冲器（省复制API）【标准】

【C语言API】

ER ercd = tcp_get_buf(ID cepid, VP *p_buf, TMO tmout);

【参数】

ID	**cepid**	**TCP通信端点ID**
TMO	**tmout**	**超时指定**

【返回参数】

VP	**buf**	**可用空间的首地址**
ER	**ercd**	**可以放入到发送缓冲器的数据长度/错误代码**

【特记事项】

- **即使发送缓冲器内只有1字节的空间也会返回**
- **返回连续的可用空间的长度**
- **如果连续调用就会返回相同的地址（长度发生变化）**
- **多个发送请求挂起时出现错误**

tcp_get_buf返回发送缓冲器中可以放入下一个要发送的数据的可用空间的首地址和连续的可用空间的长度。当发送缓冲器内没有可用空间时进入等待状态，直到出现可用空间。调用tcp_get_buf不会改变协议栈的内部信息。所以，即使连续调用tcp_get_buf也只会返回同一个空间。相反，如果调用tcp_snd_dat、tcp_snd_buf，协议栈的内部状态就会发生变化。如果调用这些API，之前调用的tcp_get_buf返回的信息就会变为无效。

错误代码为以下值：

正值	正常结束（可用空间的长度）
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（p_buf、timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接或发送结束tcp_snd_dat、tcp_get_buf挂起）
E_WBLK	接收无阻断调用
E_TMOUT	轮询失败或超时
E_RLWAI	取消处理、强制结束等待状态
E_CLS	断开TCP连接

tcp_snd_buf 缓冲器内的数据发送(省复制API) 【标准】

【C语言API】

ER ercd = tcp_snd_buf(ID cepid, INT len);

【参数】

ID	**cepid**	**TCP通信端点ID**
INT	**len**	**数据长度**

【特记事项】

- **发送通过tcp_get_buf获取的发送缓冲器内的数据（所以不需要传递首地址）**
- **实际上只是做发送的准备，所以在此服务调用内不会阻断**

发送通过tcp_get_buf从缓冲器内取出的长度为len的数据。

tcp_snd_buf只是做发送的准备，所以不会在API中等待。len比可以放入要发送的数据的连续可用空间的长度(调用tcp_snd_buf得到的值)长的时候，返回E_OBJ错误。

错误代码为以下值：

正值	正常结束
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（p_buf、timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接或发送结束，len过长）
E_CLS	断开TCP连接

tcp_rcv_buf 接收缓冲器的获取(省复制API) 【标准】

【C语言API】

ER ercd = tcp_rcv_buf(ID cepid, VP *p_buf, TMO tmout});

【参数】

ID	**cepid**	**TCP通信端点ID**
TMO	**tmout**	**超时指定**

【返回参数】

VP	**buf**	**接收数据的首地址**
ER	**ercd**	**接收数据的长度/错误代码**

【特记事项】

- **缓冲器中即使只有1字节的接收数据也会返回**
- **返回连续存储的接收数据的长度**
- **如果连续调出就会返回相同的地址（长度发生变化）**
- **多个接收要求挂起时出现错误**
- **如果由通信对方断开，返回值为0**

tcp_rcv_buf返回存放接收数据的缓冲器的首地址以及从此处连续进入的数据的长度。接收缓冲器为空时进入等待状态，直到接收数据。如果由对方正常结束连接接收缓冲器中就会没有了数据，此时会从API返回0。

调用tcp_rcv_buf协议栈的内部状态不会发生变化。所以，如果连续调用tcp_rcv_buf会返回相同的空间。相反，如果调用 tcp_rcv_dat、tcp_rel_buf，之前调用的tcp_rcv_buf返回的值将变为无效。

如果在有tcp_rcv_dat或tcp_rcv_buf挂起的同一个TCP通讯端点发行tcp_rcv_buf，就会出现E_OBJ错误。

错误代码为以下值：

正值	正常结束（接收数据的长度）
0	数据结束（正常断开连接）
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（p_buf、timout是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接或tcp_rcv_dat、tcp_rcv_buf挂起）
E_TMOUT	轮询失败或超时
E_RLWAI	取消处理、强制解除等待状态
E_CLS	断开TCP连接、接收缓冲器为空

tcp_rel_buf 接收缓冲器的释放(省复制API) 【標準】

【C语言API】

ER ercd = tcp_rel_buf(ID cepid, INT len);

【参数】

ID	**cepid**	**TCP通信端点ID**
INT	**len**	**数据长度**

【特记事项】

- **丢弃通过tcp_rcv_buf获取的接收缓冲器中的数据（所以不需要传递首地址）**
- **在此服务调用内不会阻断**

丢弃通过tcp_rcv_buf取出的缓冲器中长度为len的数据。在此API中不会进入等待状态。len比接收到的数据连续进入的数据的长度（tcp_rcv_buf返回的值）长的时候，会出现E_OBJ错误。原则上不管是什么样的条件只要是非法的值（如负值）就会出现E_OBJ错误。

错误代码为以下值：

正值	正常结束
E_ID	非法ID号
E_NOEXS	对象未生成
E_PAR	参数错误（len是非法的)
E_OBJ	对象状态错误（指定的TCP通信端点没有连接，len过长）

省复制API: 程序例子

```
/* 获取接收到的数据长以及其地址 */
while (( ercd1 = tcp_rcv_buf(cepid, &rbuf, TMO_FEVR)) > 0){
    /* 获取发送缓冲器 */
    ercd2 = tcp_get_buf(cepid, &sbuf, TMO_FEVR); /* 省略错误处理 */
    len = min(ercd1, ercd2);
    for ( i = 0; i < len; i++){
      if (isupper(rbuf[i]))
          sbuf[i] = tolower(rbuf[i]);
      else
          sbuf[i] = rbuf[i];
    }
    /* 发送数据 */
    ercd = tcp_snd_buf(cepid, len);          /* 省略错误处理 */
    /* 释放接收缓冲器 */
    ercd = tcp_rel_buf(cepid, len);          /* 省略错误处理 */
}
```


TOPPERS

省复制API的程序例子：

使用省复制API记述了对接收到的数据进行回送校验的程序。用While语句的条件tcp_rcv_buf取出接收数据的首地址和大小。值为0时正常断开，为负值时出现错误并跳出while语句。用下一个tcp_get_buf取出可以发送的缓冲器空间的首地址和大小。将发送空间大小和接收数据量中较小的设定为len，并将len大小的接收数据复制到发送空间缓冲器中。此时大写字符将会转换成小写字符。

通过tcp_snd_buf要求发送复制的数据，通过tcp_rel_buf丢弃复制的len大小的接收数据。

TINET
（TCP/IP协议栈）

1. TCP/IP基础
2. ITRON TCP/IP规范
3. TINET（TCP/IP堆栈）
4. TCP服务器端程序
5. TCP客户端程序
6. 总结

TINET的特征

由苫小牧高专 情报工学科开发的
ITRON TCP/IP API规范的小型TCP/IP协议栈

- **基于FreeBSD**
  - **–成熟的软件、其他系统的标准**
  - **–以BSD版权发布**
  - **–人力资源上的限制**
- **对应嵌入式系统的资源限制**
- **对应RTOS是TOPPERS/JSP Kernel**
- **API为ITRON TCP/IP API规范**

TOPPERS

TINET是依照苫小牧高等专科信息工程学专业开发的ITRON TCP/IP API规范的小型TCP/IP协议栈。TINET是基于FreeBSD的协议栈进行开发的。实装以TOPPERS/JSP Kernel为对象。所以，在包含TINET的软件在别的软件开发中不能使用的前提下进行再发布时，根据TOPPERS的许可、FreeBSD及FreeBSD的软件赠送者的许可规定，需要在再发布的文档（使用者手册等）上注明使用权等。但是我们知道，其实FreeBSD许可与其他的资源相比限制已经很少了。

FreeBSD的基础BSD UNIX的协议栈是成熟的软件，而且已经是很多系统的标准。

在实装方面考虑的事项

- 嵌入式系统的资源限制（仅TINET）
  - **IPv4为RAM约8KB、ROM约41KB**
  - **IPv6为RAM约9KB、ROM约59KB**
- **ITRON TCP/IP API 规范的必要性能**
  - 最少的复制次数
  - 排除动态内存管理
  - 非同步接口
  - 各API的错误信息的详细化
- 实际时间性的限制

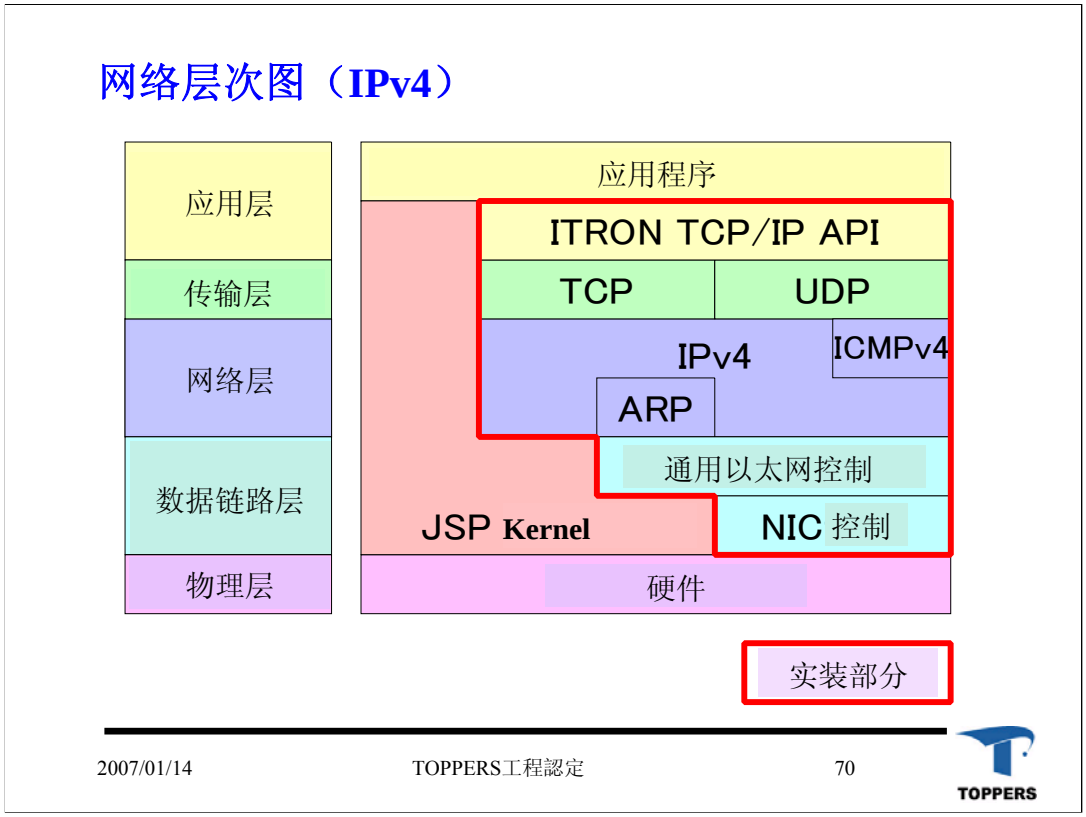
TOPPERS

TINET的开发是假定要运行在内存容量严格受限的嵌入式系统上运行来进行的。所以，实装的对象是单一链接的嵌入式系统。假设是最小的结构，那么在IPv4的实装中占RAM约8KB、占ROM约41KB，在IPv6的实装中占RAM约9KB、占ROM约59KB。关于IPv6，现在（2005年6月）ITRON TCP/IP API规范中没有将IPv6的规范进行标准化，而只是作为临时规范来进行使用的。

TINET通过开发基于ITRON TCP/IP API规范的应用程序，可以对应广泛的应用层。作为TCP的选项，只能用定义语句指定最大段长度MSS(Maximum Segment Size)。数据链路层的接口是通过取决于网络系统的接口来实现控制的。

TINET的实装目标目前（2005年6月）有以下两个：

- 秋月电子商务生产的AKI-H8/3069F LAN板对应以下的功能。
  - NE2000兼容NIC(REALTEK生产的RTL8019AS)
  - 内部回送功能
  - 使用串行接口的PPP
- 北斗电子生产的HSB7727ST(SH3)板对应以下的EtherNet。
  - SMSC生产的LAN91C111NIC



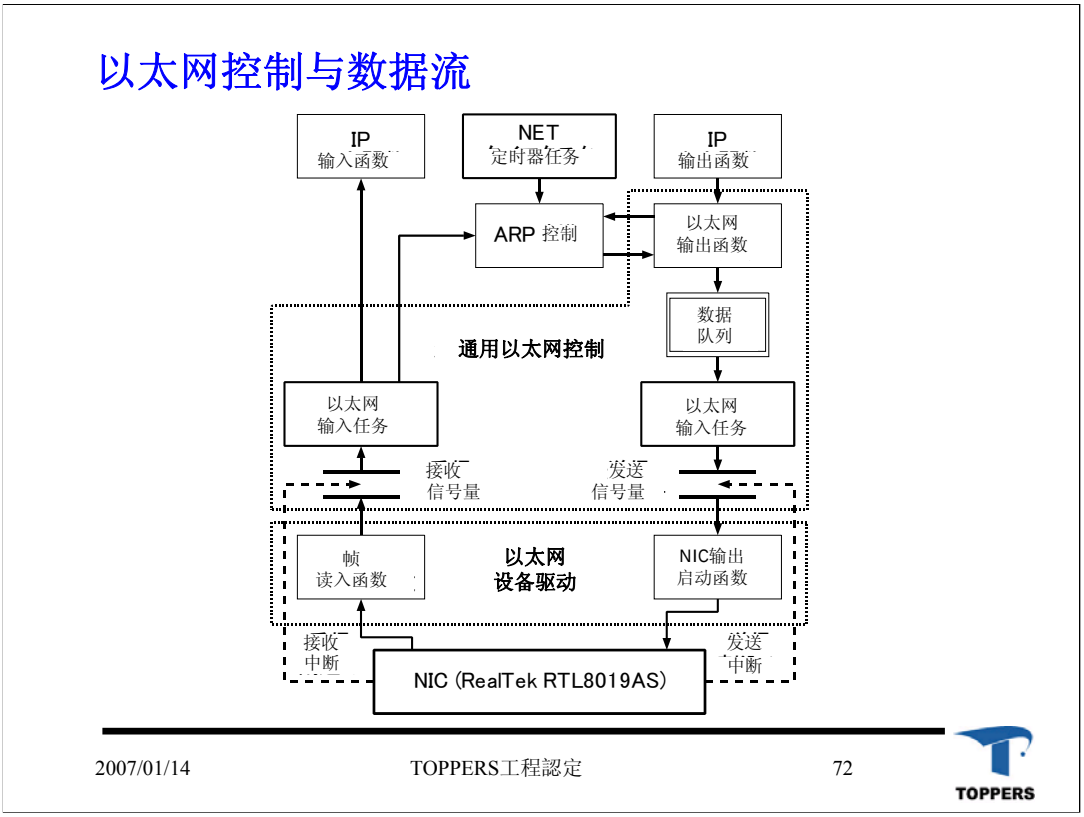
上图是对应网络层次图的TOPPERS/JSP以及TINET的层次图。传输层是TCP和UDP，与应用层的接口为ITRON TCP/IP API规范。网络层可以选择IPv4和IPv6。数据链路层进行不依赖于网络系统的通用以太网的控制。

网络缓冲器（net_buf）

- TCP/UDP、IP、存放以太网Header和数据的协议缓冲器
- 输出时使用TCP/UDP
- 输入时使用以太网设备驱动
- 相当于BSD的mbuf
- 固定长度的存储区
  - –事先保存下层的Header空间。
  - –中途不进行动态内存操作。
  - –Header长度为4个八位字节，1,514个八位字节的以太网帧所占的系统开销为0.4%。
  - –MTU（Maximum Transmisson Unit）
    - 网络的最大传输单位（在Ethernet上为1500）

TOPPERS

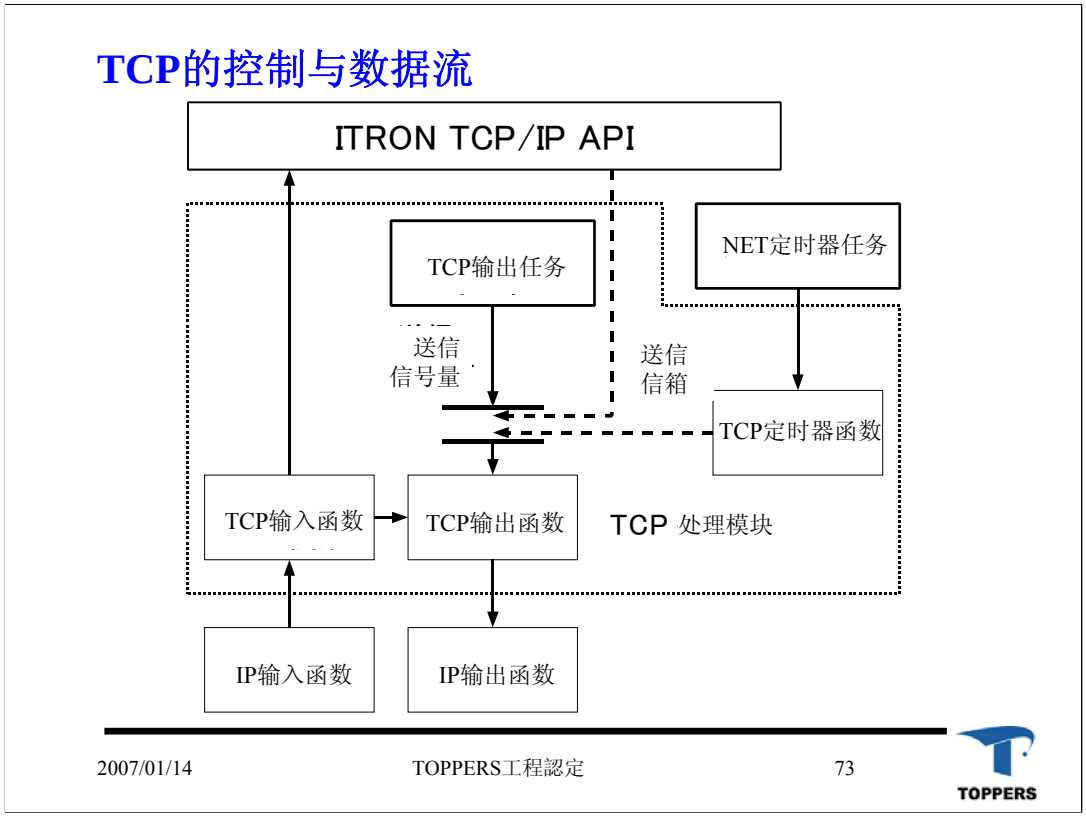
在TINET中，使用网络缓冲器(net_buf)来实现协议栈内各层次的数据传递。这是从ITRON 4.0规范的固定长度内存池中取出并进行分配。输出时使用TCP／UDP、输入时使用以太网驱动器来确保数据，而且两者都是在分配前确保数据。在协议栈内不进行网络缓冲器的动态内存操作。这些内存资源是在依赖于CPU的结构目录下的依赖于TINET的引用文件中进行设定的。



上图是从网络层到物理层的数据流程图。数据链路层由不依赖于NIC的通用Ethernet控制和依赖于NIC的Ethernet设备构成。此图是以通用Ethernet的数据流程为中心进行说明的。

接收了从Ethernet发来的信息时，在NIC上发生中断。此中断与以太网输入任务是同步的，而且这个任务从接收缓冲器中取出以太网帧。从以太网Header判断上层协议，并从IP输入函数向上层协议传递数据。

向Ethernet输出时，在网络缓冲器中设定EthernetHeader和IPDatagram，并从上层协议调用IP输出函数。以太网输出函数在网络缓冲器的Ethernet缓冲器上设定MAC地址。此时如果协议为IPv4，则通过ARP解决发送目的地的MAC地址。然后将网络缓冲器放入到数据队列中。以太网输出任务从数据队列中取出网络缓冲器，并与NIC中断句柄同步进行数据的写入。



在TCP控制中，按照ITRON TCP/IP API的步骤将来自于IP输入函数的网络缓冲器传递给通用程序，并依照ITRON TCP/IP API规范将接收到的网络发送数据传递给IP输出函数。

ITRON TCP/IP API 的实装

- API
  - ITRON TCP/IP API规范的标准功能
  - 暂定的ITRON TCP/IP（版本6） API规范的标准功能
- TCP
  - BSD 的通信功能
  - 最大段长度（MSS）选项
  - 省复制API
  - 无阻断调用
- UDP
  - 无阻断调用
- 静态API
  - TCP/UDP 通信端点的生成
  - TCP 接收口的生成

下面对按照TINET的ITRON TCP/IP API规范的实装进行说明。IPv4实装了ITRON TCP/IP API规范的标准功能。但IPv6只是实装了暂定的规范的标准功能。现在（2005年6月）Tron协会正在修订ITRON TCP/IP API规范，可能会与TINET的暂定的规范有所不同。

TCP支持BSD的通信功能、最大段长度的设定接口、ITRON TCP/IP API规范中指定的省复制API以及无阻断调用的发送接收接口。UDP支持无阻断调用的发送接收功能。静态API支持TCP/UDP的通信端点的生成和TCP接收口的生成。

TCP 的API

API 名	功　　能	NBLK*		省拷贝
		发送类	接收类	
TCP_CRE_REP	TCP收发窗口的生成			
TCP_CRE_CEP	TCP通信端点的生成			
tcp_acp_cep	等待连接要求		○	
tcp_con_cep	连接要求	○		
tcp_sht_cep	数据通信结束			
tcp_cls_cep	通信端点的关闭		○	
tcp_snd_dat	数据的发送	○		
tcp_rcv_dat	数据的接收		○	
tcp_get_buf	发送缓冲器的获取	○		○
tcp_snd_buf	缓冲器内数据的发送			○
tcp_rcv_buf	接收缓冲器的获取		○	○
tcp_rel_buf	接收缓冲器的释放			○
tcp_can_cep	处理的取消			

NBLK*: 无阻塞调用

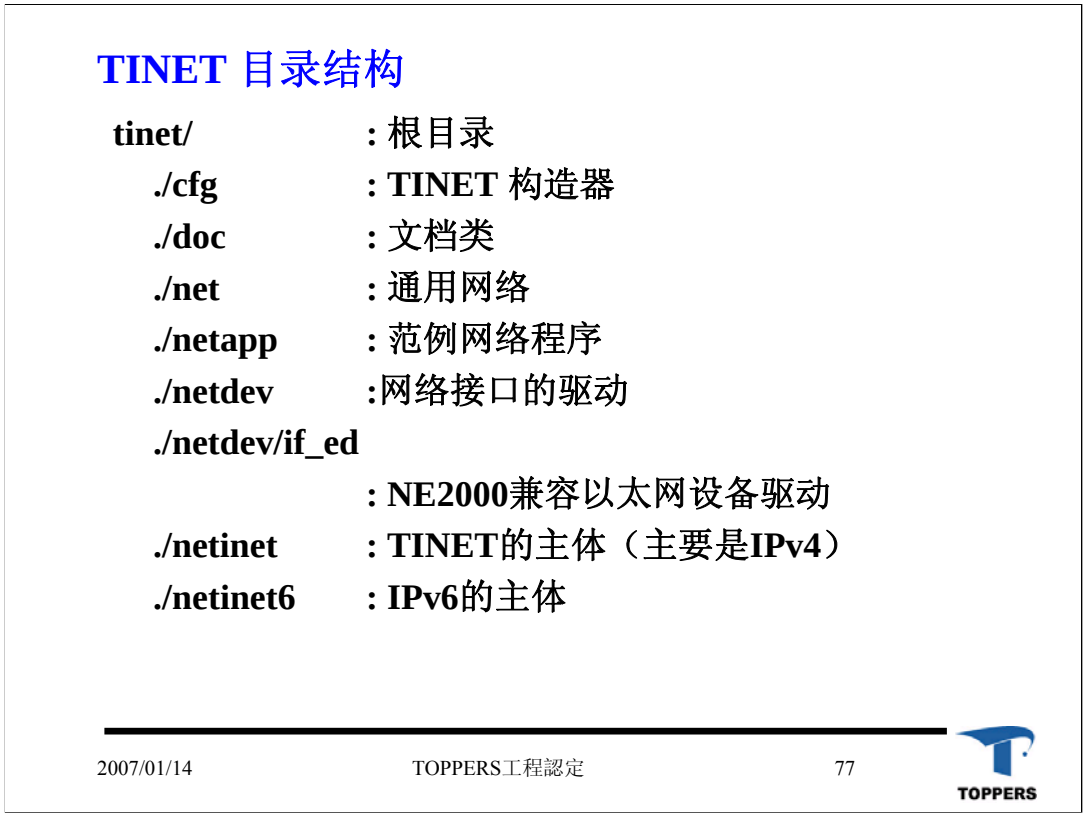
TOPPERS

支持ITRON TCP/IP API规范中的TCP的API如上记述。对应无阻断调用和省复制的API分别用○进行标注。

UDP 的API

API 名	功　　能	NBLK	
		发送类	接收类
UDP_CRE_CEP	UDP通信端点的生成		
udp_snd_dat	数据包的发送	○	
udp_rcv_dat	数据包的接收		○
udp_can_cep	处理的取消		
UDP包的接收	UDP包的接收		

支持ITRON TCP/IP API规范中的UDP的API如上记述。对应无阻断调用的API分别用○进行标注。



下面介绍一下TINET的目录结构。tinet目录在jsp目录下。./cfg目录存放解决ITRON TCP/IP API规范的静态API的TINET构造器的源代码。./doc目录存放TINET的文档类。./net目录存放不依赖于网络系统的通用网络控制相关的程序。./netapp目录存放Sample网络应用程序。./netdev目录存放Ethernet设备驱动中不依赖于JSP目标的程序文件。./netdev/if_ed目录存放NE2000兼容的RTL8019AS用的设备驱动。./netinet目录存放取决于IPv4的TINET的主体程序。./netinet6目录存放取决于IPv6的TINET的主体程序。

TINET 的文档（./doc）

- **tinet.txt**
  – 主菜单、API、应用程序构建方法
- **tinet_install.txt**
  – 安装手册、Sample应用程序的介绍
- **tinet_config.txt**
  – 结构说明、宏的解说等
- **tinet_defs.txt**
  –处理器、系统依赖的解说
- **tinet_sample.txt**
  – 样例程序的说明
- **tinet_chg.txt**
  – 变更履历

./doc目录中的文档内容如上所述。

配置/参数定义文件

- **config/$(CPU)/tinet_cpu_config.h**
  - 处理器依赖定义文件
- **config/$(CPU)/$(SYS)/tinet_sys_config.h**
  - 系统依赖定义文件
- **tinet/netdev/$(NIC)/tinet_nic_config.h**
  - 网络接口依赖定义文件
- **config/$(CPU)/tinet_cpu_defs.h**
  - 处理器依赖定义文件
- **tinet/netdev/$(NIC)/tinet_nic_defs.h**
  - 网络接口定义文件

下面介绍一下取决于实装的参数依赖文件。

取决于TOPPERS/JSP目标的文件有config/$(CPU)/$(SYS)中的JSP系统依赖的汇编函数文件sys_support.S、系统依赖的引用文件tinet_sys_config.h。为H8的时候，在sys_support.S中对设定RTL8019AS中断向量、初始化NIC硬件、许可或禁止H8中断实装依赖的中断的函数进行追加、变更。tinet_sys_config.h实现关于NIC和RTL8019AS的硬件控制的定义。

处理器依赖部分的网络设定是通过config/$(CPU)/tinet_cpu_config.h文件进行定义。

匹配等处理器依赖的设定是通过config/$(CPU)/tinet_cpu_defs.h文件进行定义。

网络应用程序依赖的定义是通过$(APP_DIR)/tinet_app_config.h实现。在此引用文件中指定板的IP地址等。

取决于网络接口的定义是通过tinet/netdev/$(NIC)/tinet_nic_config.h和tinet/netdev/$(NIC)/tinet_nic_defs.h实现。

应用程序文件

$(APP_DIR)/tinet_$(UNAME).cfg

- **TINET配置文件**
- **记述ITRON TCP/IP规范的静态API**
- **由TINET构造器进行处理**

$(APP_DIR)/tinet_app_confg.h

- **定义取决于应用程序的TINET结构参数的文件**

$(APP_DIR)/route_cfg.c

- **静态路径定义文件**

$(APP_DIR)/tinet_$(UNAME).cfg是TINET用的配置文件，$(UNAME)指定网络应用程序名称。在此文件中定义生成通讯端点、接收口等的静态API。此文件由make tinet进行创建，并由TINET构造器转换成C语言的程序。

$(APP_DIR)/route_cfg.c是定义静态路径常数的C语言程序。

tinet_app_config.h的例子

```
//IP 地址的设定
#define IPV4_ADDR_LOCAL                 ¥
    0xac198158              /* 172.25.129.88 */
#define IPV4_ADDR_LOCAL_MASK        ¥
    0xffffff00              /* 255.255.255.0 */
#define IPV4_ADDR_DEFAULT_GW        ¥
    0xac19818c              /* 172.25.129.140 */
```


以上是tinet_app_config.h文件的记述例子。

IPV4_ADDR_LOCAL以16进制指定IP地址。

IPV4_ADDR_LOCAL_MASK指定网络掩码值。

IPV4_ADDR_DEFAULT_GW指定默认网关的IP地址。

静态路径定义文件route_cfg.c的例子

```
T_IN4_RTENTRY
routing_tbl[NUM_ROUTE_ENTRY] = {
    /* default  gateway、间接发送 */
    { 0,  0,   IPV4_ADDR_DEFAULT_GW },
    /* 同一LAN内、直接发送  */
    { IPV4_ADDR_LOCAL & IPV4_ADDR_LOCAL_MASK,
      IPV4_ADDR_LOCAL_MASK,  0 },
    /* 对同一LAN内进行广播传输、直接发送 */
    { 0xffffffff,     0xffffffff,  0 },
};
```


route_cfg.c使用routing_tbl设定静态路径表。

TINET构造器生成文件

$(APP_DIR)/tinet_cfg.c

- **–生成对应TCP接收口、TCP通信端点以及UDP通信端点的结构体的文件。与应用程序、TINET一起进行编译并链接。**

$(APP_DIR)/tinet_kern.cfg

- **–生成TINET内部使用的Kernel对象的静态API的文件。由JSP的系统配置文件（标准是$（UNAME）.cfg）进行引用。**

$(APP_DIR)/tinet_id.h

- **–TCP接收口、TCP通信端点以及UDP通信端点的ID自动分配结果文件**

TINET构造器在应用目录下生成以下文件。

・$(APP_DIR)/tinet_cfg.c

用C语言的结构体数据记述TINET用配置文件中记述的TCP接收口、TCP通信端点以及UDP通信端点等的源文件。在创建时与应用文件等一起链接。

・$(APP_DIR)/tinet_kern.cfg

用静态API记述TINET内部使用的JSP Kernel的Kernel对象的配置文件。由JSP的配置文件进行引用，并由JSP的构造器转换成C语言文件。

・$(APP_DIR)/tinet_id.h

定义TCP接收口、TCP通信端点以及UDP通信端点的ID自动分配的值的引用文件。

文件tinet_kern.cfg的例子

```
CRE_SEM(SEM_TCP_CEP_LOCK1,{ TA_TPRI, 1, 1 });
CRE_SEM(SEM_TCP_CEP_SBUF_BUSY1,
            { TA_TPRI, 1, 1 });
CRE_SEM(SEM_TCP_CEP_RBUF_READY1,
            { TA_TPRI, 0, 1 });
CRE_FLG(FLG_TCP_CEP1,
            { TA_TFIFO|TA_WSGL,CEP_EVT_CLOSED });
```


在生成的tinet_kern.cfg文件中，定义了ITRON4.0规范的Kernel对象的静态API。

TCP服务器端程序设计

1. TCP/IP基础
2. ITRON TCP/IP规范
3. TINET（TCP/IP栈）
4. TCP服务器端程序
5. TCP客户端程序
6. 总结

TCP服务器程序设计

使用ITRON TCP/IP API规范
创建TCP服务器程序

- 使用TINET和TOPPERS/JSP Kernel
- 已完成的程序在程序目录的以下目录下
  - ./OBJ/AKIH8_3069F/
    - TCP_SRV :基本服务器
    - ECHO_SRV :回显服务器
    - DEV_SRV :设备服务器
    - MULTI_SRV :多任务服务器

使用TINET创建ITRON TCP/IP API。

关于此部分创建的TCP服务器程序，在./OBJ/AKIH8_3069F目录下创建了完成程序。实习后请作为参考。

TCP服务器程序设计：PC端的设定

将PC有线端的网络IP地址设为192.168.11.6、
网络掩码设为255.255.255.0

- 用以太网线连接电路板和PC
- 右击桌面上的“网上邻居”并选择“属性”
- 在打开的窗口中右击“本地连接”并选择“属性”
- 选择网络协议（TCP/IP）并点击属性
- 指定IP地址和子网掩码
  - IP地址　：192.168.11.6
  - 子网掩码　：255.255.255.0
- 通过控制台或用命令提示符运行ipconfig来确认设定

用交叉电缆连接AKIH8-3069F板和PC。PC端必须设定固定的IP地址。设定PC后，在Cygwin或DOS窗口中输入ipconfig命令来确认PC的IP地址。

TCP服务器程序设计

- 已完成的程序在程序目录的以下目录下
  - **./OBJ/AKIH8_3069F/**
    - **TCP_SRV：基本服务器**
    - **ECHO_SRV：回显服务器**
    - **DEV_SRV：设备服务器**
    - **MULTI_SRV：多任务服务器**
- **请复制TCP_SRV的内容并创建MY_SRV目录**
- **在各程序中将电路板的IP地址设为192.168.11.98。IP地址在tinet_app_config.h 的IPV4_ADDR_LOCAL宏中有设定。**

关于此部分创建的TCP服务器程序，在./OBJ/AKIH8_3069F目录下创建了完成程序。实习后请作为参考。

在TOPPERS/JSP-1.4.1的H8的实装中，为了使MAC驱动的中断设定有效，需要定义-DSUPPORT_ETHER并重新构建。但在1.4.2的实装中已经可以自动反映DEF_INH的设定了。以下是JSP-1.4.1教材的说明。

请将Libkernel中的Makefile的第101行如下改写。

COPTS := $(COPTS) → COPTS := $(COPTS) –DSUPPORT_ETHER

请在Cygwin中移到OBJ/AKIH8_3069F/libkernel目录下，并执行以下命令来重建Kernel程序库。

```
$ make clean
$ make libkernel.a
```

然后将TCP_SRV复制到AKIH8_3069F目录下，名称为MY_TCP。

```
$ cd ..                          ‘移到OBJ/AKIH8_3069F目录下
$ cp –r TCP_SRV MY_SRV           ‘根据各目录进行复制
```

板的IP地址由tinet_app_config.h文件中的IPV4_ADDR_LOCAL宏定义实现，子网掩码由IPV4_ADDR_LOCAL_MASK的定义实现。

TCP服务器程序设计

1. 基本服务器
2. 回显服务器
3. 设备操作服务器
4. 多任务服务器

基本服务器：TCP_SRV

- 如果执行，就会用**TCP**在端口号**“1234”**中等待客户端的连接
- 与客户端连接后，将客户端发送的数据输出到串行控制台
- 编译程序后确认运行

```
$ cd MY_SRV
$ make realclean
$ ls
Makefile    tcp_srv.c    tcp_srv.h   tinet_tcp_srv.cfg
route_cfg.c  tcp_srv.cfg  tinet_app_config.h
$ make depend
$ make
```

2007/01/14 TOPPERS工程認定 90

TOPPERS

基本服务器构成TCP服务器。在端口号“1234”中等待客户端的连接，在建立连接后将客户端的输入数据显示到串行控制台上。

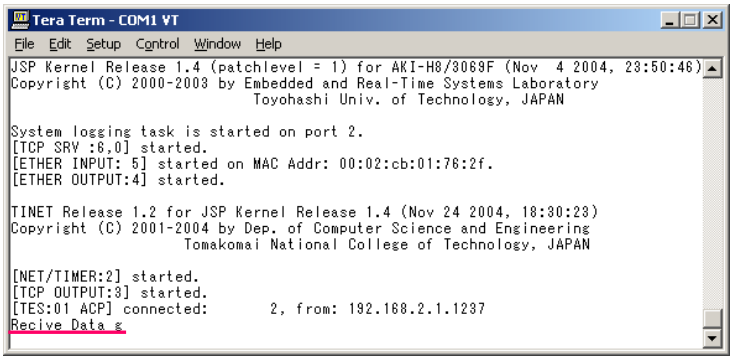
移动到MY_SRV目录下。构建基本服务器程序。

$ make tinet	‘在使用比TINET-1.3版本低的TINET时需要启动TINET的构造器 由于本教材使用的就是1.3版本，所以不需要此项操作。
$ make depend	‘生成依赖关系。JSP的结构
$ make	‘JSP的结构与编译、链接

启动teraTerm，确认串行电缆与以太网交叉电缆是否连接上，然后插上电源。如果teraTerm的ROM监视器上显示提示符，就输入ld命令，将构建的jsp.srec文件发送到板上。

基本服务器：连接

- 连接服务器时使用**telnet**
- **telnet**在开始菜单中的“运行”中运行
  telnet 192.168.11.98 1234
- 输入到**telnet**的字符被输出到串行控制台



2007/01/14 TOPPERS工程認定 91

TOPPERS

程序发送结束后，在TeraTerm 上用go命令运行基本服务器。在Cygwin或DOS窗口上用以下命令连接基本服务器。

$ telnet 192.168.11.98 1234

此后，用Enter键输入的字符会显示在TeraTerm上。

结束telnet时，在CTRL-]后通过Enter键输入显示以下提示符。

telnet>

如果在此提示符中输入quit[enter]，就会结束telnet。

telnet>quit

基本服务器：文件构成

- Makefile : Make文件
- route_cfg.c : 静态路由选择设定文件
- tinet_app_config.h : 配置定义文件
- tinet_tcp_srv.cfg : TINET结构
- tcp_srv.h : 用户程序Header
- tcp_srv.c : 用户程序源代码
- tcp_srv.cfg : JSP配置文件

TCP_SRV中的文件如上所述。

基本服务器：Make文件（Makefile）

- 与JSP的标准Makefile基本相同
- 在进行各种设定后引用TINET的Makefile.config

```
# 网络接口
NET_IF = ether

# 以太网・设备驱动的选择
NET_DEV = if_ed

# 网络层的选择
SUPPORT_INET4 = true

# 传输层的选择
SUPPORT_TCP = true

#
# TINET 的Makefile.config 的引用
#
include $(SRCDIR)/tinet/Makefile.config
```


TOPPERS

包含TINET的MakeFile与JSP的标准Makefile基本相同。在进行TINET用的各种设定后引用TINET用Makefile.config。然后进行TINET用应用程序的程序设定。

NET_IF选择网络接口。通过Loop、ppp、ether进行选择。

NET_DEV选择使用的设备。AKIH8_3069F板只有if_ed。

SUPPORT_INET4=true实现网络层选择IPV4。

SUPPORT_TCP=ture实现传输层选择TCP。可以同时设定SUPPORT_UDP。

基本服务器：静态路由选择设置文件（route_cfg.c）

- 作为静态路由选择信息，创建由tinet/netinet/in.h定义的T_IPV4EP类型的结构体数组
- 详细内容请参照TINET手册（tinet/doc/tinet.txt）的第6章
- 在只是默认网关的简单网络中，可以直接使用Sample应用程序route_cfg.c
- 静态路径表的格式

```
T_RT_ENTRY routing_tbl[NUM_ROUTE_ENTRY] = {
        { 0             0,              <gateway> },    /* 默认         */
        { <net>,        <mask>,         0         },    /* 直接发送     */
        { 0xffffffff,   0xffffffff,     0         },    /* 广播传输     */
};
```


静态路由选择设定文件(route_cfg.c)进行静态路由选择的定义。IPV4的路由入口是由./tinet/netinet/in.h文件进行定义的。静态路由入口是事先指定的路由选择信息。在只是默认网关的简单网络中，可以直接使用Sample应用程序的route_cfg.c（如下）。以下各定义请参照tinet_app_config.h。

```
T_IN4_RTENTRY routing_tbl[NUM_ROUTE_ENTRY] = {

        /*通过不同的LAN、default gateway 间接发送              */
        { 0,                    0,              IPV4_ADDR_DEFAULT_GW    },

        /*同一LAN内，直接发送                                   */
        { IPV4_ADDR_LOCAL &
          IPV4_ADDR_LOCAL_MASK,IPV4_ADDR_LOCAL_MASK,            0
                },

        /*向同一LAN内的广播传输方式、直接发送                   */
        { 0xffffffff,                           0xffffffff,     0
                },

        };
```

基本服务器：配置定义文件

- **tinet_app_config.h**
- 定义应用程序中的相关参数
  （请参照**tinet/doc/tinet_config.txt**）

```
/* 网络统计信息的获取 */
#define SUPPORT_MIB
/* IP地址 */
#define IPV4_ADDR_LOCAL             0xc0a8b262      /* 192.168.11.98    */
/* 子网掩码 */
#define IPV4_ADDR_LOCAL_MASK    0xffffff00          /* 255.255.255.0    */
/* 默认网关 */
#define IPV4_ADDR_DEFAULT_GW    0xc0a80b01          /* 192.168.11.1     */
/* 路由选择表的静态路由入口数 */
#define NUM_STATIC_ROUTE_ENTRY        3
/* 由改变路径(ICMP)产生的路由选择数。如果指定0则忽略路径改变 */
#define NUM_REDIRECT_ROUTE_ENTRY    0
```


配置定义文件（tinet_app_config.h）设定应用程序相关的参数。

SUPPORT_MIB使依照SNMP用管理信息库（MIB）的网络统计的获取有效。但TINET只提供依照管理信息库（MIB）的网络统计，它并不支持SNMP。

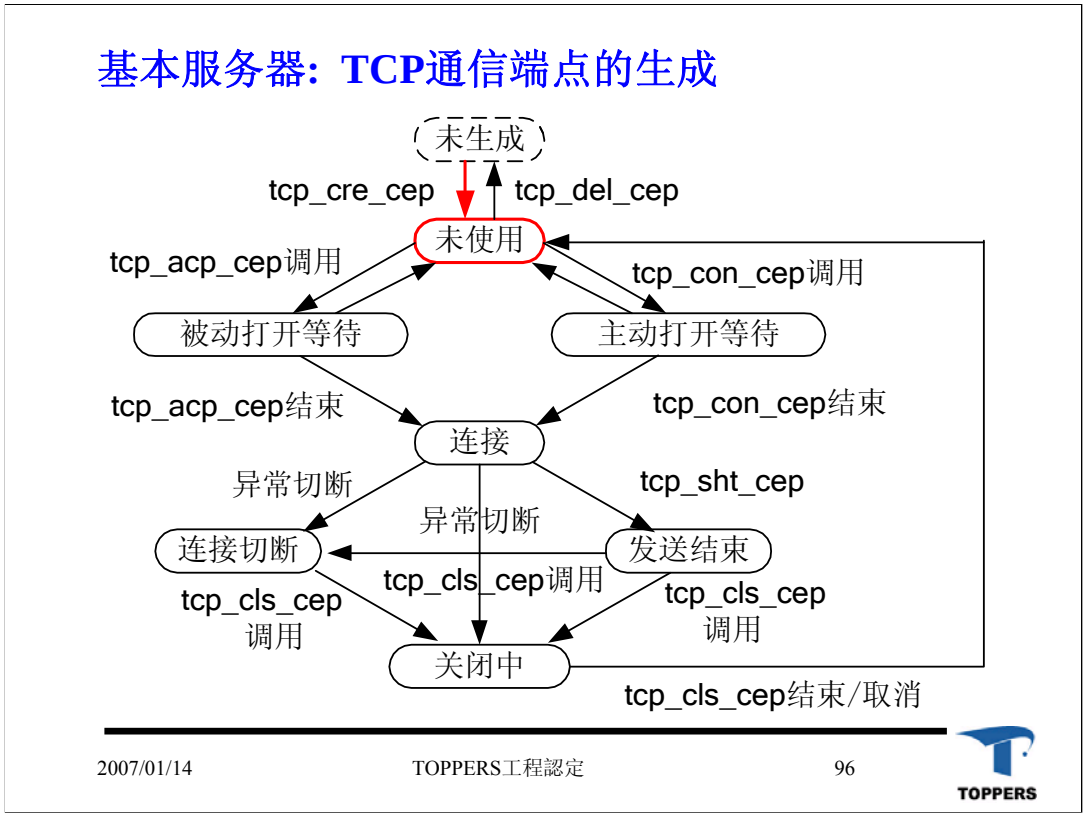
IPV4_ADDR_LOCAL指定自己的IP地址。但在使用PPP时，如果由对方进行分配就指定0。

IPV4_ADDR_LOCAL_MASK只有在网络接口为以太网时是有效的，它指定子网掩码。

IPV4_ADDR_DEFULT_GW只有在网络接口为以太网时是有效的，它指定默认网关。

NUM_STATIC_ROUTE_ENTRY指定路由选择表的静态路由入口数。

NUM_REDIRECT_ROUTE_ENTRY指定路线改变（ICMP）的路由入口数。为0时忽略路线的改变（ICMP）。



下面介绍一下TCP通信端点和TCP接收口的生成步骤。关于TINET，通信端点是由静态API生成的。由于是在编译、连接时生成对象的，所以无法用tcp_dep_cep来删除。

基本服务器：TINET结构(tinet_tcp_srv.cfg)

- **TCP接收口**
  - 只有服务器应用程序需要
  - 等待客户端的连接要求
  - 定义IP地址和端口号
  - 可由多个任务共同使用
- **生成TCP接收口的静态API**

```
TCP_CRE_REP(TCP接收口ID,
                {接收口属性, {IP地址,端口号}})
```

- **在基本服务器中创建一个接收口**

```
TCP_CRE_REP (TCP_SRV_REPID,{
                0, { IPV4_ADDRANY, REP_PORT } } )
```

等待接收对自己的所有IP地址的连接要求

1234

TOPPERS

只有TCP的服务器应用程序需要TCP接收口。它是等待客户端的连接要求的对象。服务器的IP地址和端口号是它的属性。可由多个任务共同使用。

在ITRON TCP/IP API规范中为以下内容：

【静态API】

TCP_CRE_REP(ID repid, {ATR repatr, {UP myipaddr, UH myportno)}});

【参数】

ID	repid	TCP接收口ID（自动分配）
ATR	repatr	TCP接收口属性
UP	myaddr	自己的IP地址
UH	myportno	端口号

将自己的IP地址指定为IPV4_ADDRANY（=0）时，要等待向自己所有IP地址发出的连接要求。

在基本服务器中，生成TCP接收口的静态API记述在TINET的配置文件（tinet_tcp_srv.cfg）内。

基本服务器：TINET结构(tinet_tcp_srv.cfg)

- 用于生成TCP通信端点的静态API

```
TCP_CRE_CEP(TCP通信端点ID, {
        通信端点属性,
        发送缓冲器,发送缓冲器容量,
        接收缓冲器,接收缓冲器容量,
        回调函数
})
```

- 在基本服务器中创建一个TCP通信端点

```
TCP_CRE_CEP (TCP_SRV_CEPID, {
        0,
        tcp_srv_cep_sbuf, TCP_SRV_CEP_SBUF_SIZE,
        tcp_srv_cep_rbuf, TCP_SRV_CEP_RBUF_SIZE,
        NULL
} )
```


TCP通信端点用专门用于生成它的静态API生成。关于TINET，自己必须确保用于发送与接收的缓冲器。在ITRON TCP/IP API规范中生成TCP通信端点的静态API如下所示：

【静态API】

TCP_CRE_CEP(ID cepid, { ATR cepatr, VP sbuf, INT sbufsiz, VP rbuf, INT rbufsz,
FP callback});

【参数】

ID	cepid	TCP通信端点ID（自动分配）
ATR	cepatr	TCP通信端点属性
VP	sbuf	发信用窗口缓冲器的首地址
INT	sbufsz	发信用窗口缓冲器的大小
VP	rbuf	接收用窗口缓冲器的首地址
INT	rbufsz	接收用窗口缓冲器的大小
FP	callback	回调例程的地址

在基本服务器中使用一个TCP通信端点。

基本服务器：TINET结构(tinet_tcp_srv.cfg)

- 发送接收缓冲器（**TCP窗口缓冲器**）
  - 由应用程序提供
- **tcp_srv.h**

```
#define TCP_SRV_CEP_SBUF_SIZE       (1024*4)
#define TCP_SRV_CEP_RBUF_SIZE       (1024*4)
```

- **tcp_srv.c**

```
UB tcp_srv_cep_sbuf[TCP_SRV_CEP_SBUF_SIZE];
UB tcp_srv_cep_rbuf[TCP_SRV_CEP_RBUF_SIZE];
```


TCP通信端点使用的发送接收缓冲器需要由应用程序来提供。在基本服务器中，tcp_srv.h引用文件定义了窗口缓冲器的大小。发送接收缓冲器空间由用户程序的tcp_srv.c提供。

基本服务器：运行TINET构造器

- 用make tinet运行TINET的构造器
  - 通过预处理程序将tinet_tcp_srv.cfg转换为tmpfile9
  - 对生成的tmpfile9运行构造器（tinet_cfg）

```
$ cd MY_TCP
$ make realclean
$ ls
Makefile        tcp_srv.c      tcp_srv.h              tinet_tcp_srv.cfg
route_cfg.c     tcp_srv.cfg    tinet_app_config.h
$ make depnd
h8300-hms-gcc -E -I. -I../../../include  -I../../../config/h8/akih8_3069f –I../../../config/h8 –I../
../../tinet/netdev/if_ed -I../../../tinet –DCPU_CLOCK=200000000 –DLABEL_ASM –DVEC
TOR_SIZE=64 –DUSE_PING –DUSE_NETAPP_SUBR -DUNDEF_TCP_CFG_PASSIV
E_OPEN –DUNDEF_TCP_NON_BLOCKING -DUNDEF_UDP_CFG_NON_BLOCKING
 -DSUPPORT_INET4 –DSUPPORT_ETHER  -DSUPPORT_TCP –DTCP_CFG_LIBRAR
Y –DUDP_CFG_LIBRARY –DTNCT_MONITOR -x c-header tinet_tcp_srv.cfg > tmpfile9
../../../tinet/cfg/tinet_cfg -s tmpfile9
rm -f tmpfile9
  :
$ls
Makefile            kernel_id.h      tcp_srv.cfg          tinet_id.h
Makefile.depend     kernel_obj.dat   tcp_srv.h            tinet_kern.cfg
Kernel_cfg.c        route_cfg.c      tinet_app_config.h   tinet_tcp_srv.cfg
Kernel_chk.c        tcp_srv.c        tinet_cfg.c          vector.S
```


TOPPERS

上图是构建基本服务器时的命令及显示。TINET的构造器用“make tinet”命令运行。首先通过预处理程序转换TINET的配置文件tinet_tcp_srv.cfg，将其写入到tmpfile9中。然后对tmpfile9运行构造器，生成以下文件。

tinet_id.h	TINET的自动分配ID的文件
tinet_cfg.c	TCP接收口和TCP通信端点等空间设定
tinet_kern.cfg	TINET中使用的Kernel对象的配置文件

基本服务器：TINET构造器生成文件（1/2）

- tinet_id.h

–定义了TINET的对象的ID自动分配结果

```
#ifndef _TINET_ID_H_
#define _TINET_ID_H_
#define TCP_SRV_CEPID                1
#define TCP_SRV_REPID                1
#endif   /* of _TINET_ID_H_*/
```

- tinet_kern.cfg

–由于用静的API生成接收口、通信端点，而在TINET内部使用的Kernel对象

–由被Kernel配置文件（tcp_srv.cfg）引用的TINET配置文件（./tinet/tinet.cfg）进行引用

```
CRE_SEM(SEM_TCP_CEP_LOCK1,{ TA_TPRI, 1, 1 });
CRE_FLG(FLG_TCP_CEP_EST1,{ TA_TFIFO|TA_WSGL, CEP_EVT_CLOSED });
CRE_FLG(FLG_TCP_CEP_SND1,{ TA_TFIFO|TA_WSGL, CEP_EVT_SWBUF_READY });
CRE_FLG(FLG_TCP_CEP_RCV1,{ TA_TFIFO|TA_WSGL, 0 });
```


TOPPERS

在自动生成的tinet_id中，对TCP接收口ID（TCP_SRV_CEPID）和TCP通信端点（TCP_SRV_REPID）都分配了1。

tinet_kern.cfg中定义了在TINET内部使用的Kernel对象的静态API。此配置文件被用户程序的配置文件（tcp_srv.cfg）进行引用，并通过JSP的构造器生成Kernel对象。

基本服务器中使用3个信号量和一个事件标志。

基本服务器：TINET构造器生成文件（2/2）

- tinet_cfg.c
  - –通过静态API的生成接收口、通信端点的生成，而在TINET内部使用的结构体

```
...
#define TNUM_TCP_CREPID          1
const ID tmax_tcp_crepid = (TMIN_TCP_CREPID + TNUM_TCP_CREPID - 1);
T_TCP_CREP tcp_crep[TNUM_TCP_CREPID] = {
        {0,{ IPV4_ADDRANY, 1234 },  },
};

#define TNUM_TCP_CCEPID          1
const ID tmax_tcp_ccepid = (TMIN_TCP_CCEPID + TNUM_TCP_CCEPID - 1);
T_TCP_CCEP tcp_ccep[TNUM_TCP_CCEPID] = {
        {0,
         tcp_srv_cep_sbuf,
         ( 1024 * 4 ),
          .....},
};
```


tinet_cfg.c提供基本服务器中使用的TCP接收口和TCP通信端点的结构体空间。

TCP接收口的结构体T_TCP_CREP和TCP通信端点的结构体T_TCP_CCEP定义在tinet/netinet/tcp_var.h中。

基本服务器：用户程序

- 配置文件

– 引用TINET配置文件

```
#include "../../tinet/tinet.cfg"
```

– 生成一个任务

```
CRE_TSK(TCP_SRV_TASK, {
        TA_HLNG|TA_ACT, 0,
        tcp_srv_task,
        TCP_SRV_MAIN_PRIORITY,
        TCP_SRV_STACK_SIZE, NULL});
```

- **TINET标准引用文件**

```
#include "tinet_id.h"
#include <netinet/in.h>
#include <netinet/in_itron.h>
```


基本服务器程序由以下3个文件构成：

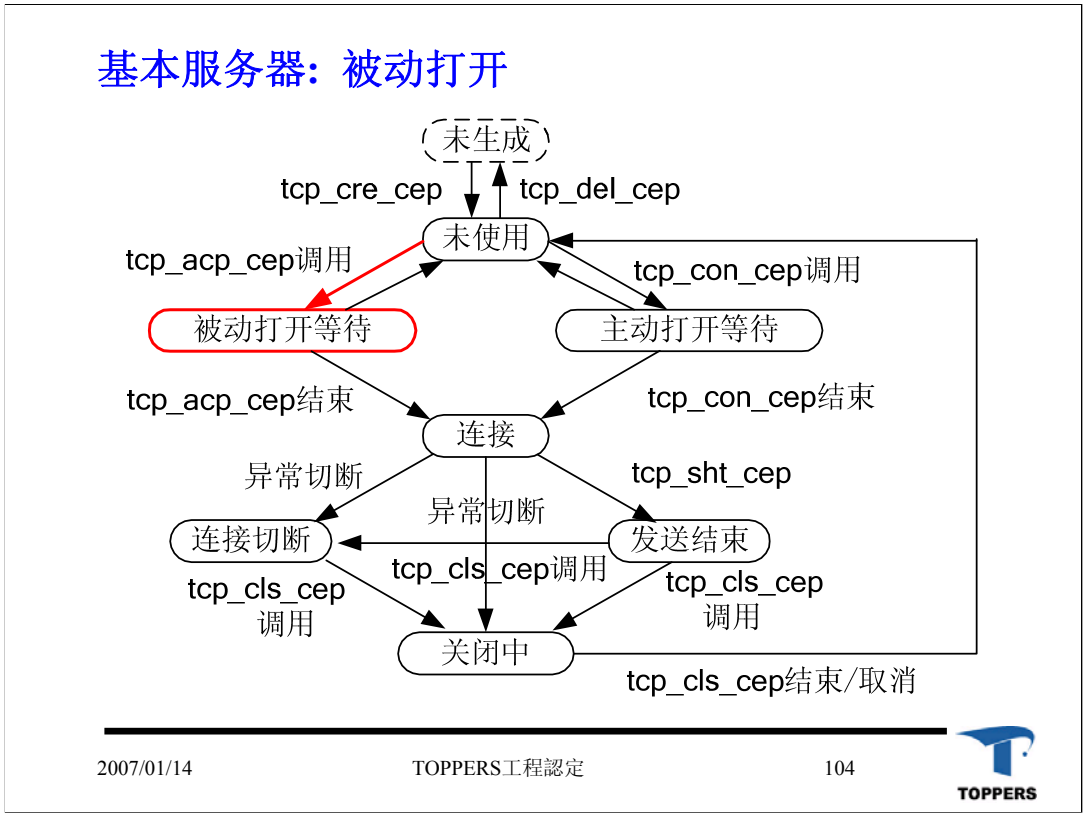
・tcp_srv.cfg　配置文件

・tcp_srv.h　引用文件

・tcp_srv.c　基本服务器主体的C语言文件

配置文件（tcp_srv.cfg）需要引用TINET的构造器生成的Kernel对象用的配置文件（tinet.cfg）。并定义生成执行基本服务器功能的任务（程序记录在tcp_srv.c中）的静态API。

如上所述，用户程序（tcp_srv.c）需要引用TINET用的标准引用文件。



介绍一下被动打开。以后的程序会在用户程序（tcp_srv.c）中进行记述。

基本服务器：被动打开

- 被动打开API

```
tcp_acp_cep(
        TCP通信端点ID, TCP接收口ID,
        IP地址/端口地址结构体,
        超时)
```

- 基本服务器

```
T_IPEP dst;

tcp_acp_cep(
        TCP_SRV_CEPID, TCP_SRV_REPID,
        &dst,
        TMO_FEVR)
```


以下是tcp_srv.c的第44行到第64行。通过第59行的tcp_acp_cep服务调用进行被动打开。如果实现客户端的连接要求，就会返回E_OK并移到ip2str行。发生错误不返回E_OK时，再次执行tcp_acp_cep服务调用进行被动打开。

```
void
tcp_srv_task(VP_INT exinf)
{
        static UB addr[sizeof("0123:4567:89ab:cdef:0123:4567:89ab:cdef")];
        ID      tskid;
        T_IPEPdst;
        ER      error = E_OK;
        char    c;
        int rlen;
        SYSTIM                  now;

        get_tid(&tskid);
        syslog(LOG_NOTICE, "[TCP SRV :%d,%d] started.", tskid, (INT)exinf);
        while(TRUE){
                if (tcp_acp_cep(TCP_SRV_CEPID, TCP_SRV_REPID, &dst, TMO_FEVR) != E_OK){
                        syslog(LOG_NOTICE, "[TES:%02d ACP] error: %s", TCP_SRV_CEPID,
itron_strerror(error));
                        continue;
                }
                ip2str(addr, &dst.ipaddr);
                get_tim(&now);
```

基本服务器：数据的接收

- 数据接收API

```
tcp_rcv_dat(
        TCP通信端点ID、
        缓冲器、 缓冲器容量、
        超时)
```

- 返回值
  - 接收字符数
  - 为0时，由通信对方断开连接
  - 负值时出现错误

以下是tcp_srv.c的第63行到第80行。第63行到第66行显示确定接收的客户端的IP地址和时间。通过第69行的tcp_rcv_dat服务调用接收客户端的数据。如果返回值为0，则是客户端断开连接时的返回值。如果是负值，则是产生错误。这两种情况下可以通过break语句跳出while语句并移到err_fin。如果返回值比0大，则是接收的数据量。此时可以通过syslog语句实现控制台的显示，并再次进入tcp_rcv_dat服务调用等待下一个数据。

```
        ip2str(addr, &dst.ipaddr);
        get_tim(&now);

        syslog(LOG_NOTICE, "[TES:%02d ACP] connected:  %6d, from: %s.%d", TCP_SRV_CEPID, now /
SYSTIM_HZ, addr, dst.portno);

        while (TRUE) {
          if ((rlen = tcp_rcv_dat(TCP_SRV_CEPID, rbuffer, BUF_SIZE - 1, TMO_FEVR)) <= 0) {
              if (rlen != E_OK)
                  syslog(LOG_NOTICE, "[TES:%02d RCV] error: %s,%d", TCP_SRV_CEPID,
itron_strerror(rlen));
              break;
          }
          rbuffer[rlen] = '¥0';
          syslog(LOG_NOTICE, "Recive Data %s", rbuffer);
        }

err_fin:
        if ((error = tcp_cls_cep(TCP_SRV_CEPID, TMO_FEVR)) != E_OK)
```

基本服务器：数据的接收

- 基本服务器
  –接收字符数保存在**rlen**中，接收数据保存在**rbuffer**中

```
rlen = tcp_rcv_dat(
            TCP_SRV_CEPID,
            rbuffer, BUF_SIZE - 1,
            TMO_FEVR)
```


以下是tcp_rcv_dat服务调用的ITRON TCP/IP API规范。
此服务调用是标准（非省复制API）的接收服务调用。

【C语言API】

ER ercd = tcp_rcv_dat(ID cepid, INT len, TMO tmout);

【参数】

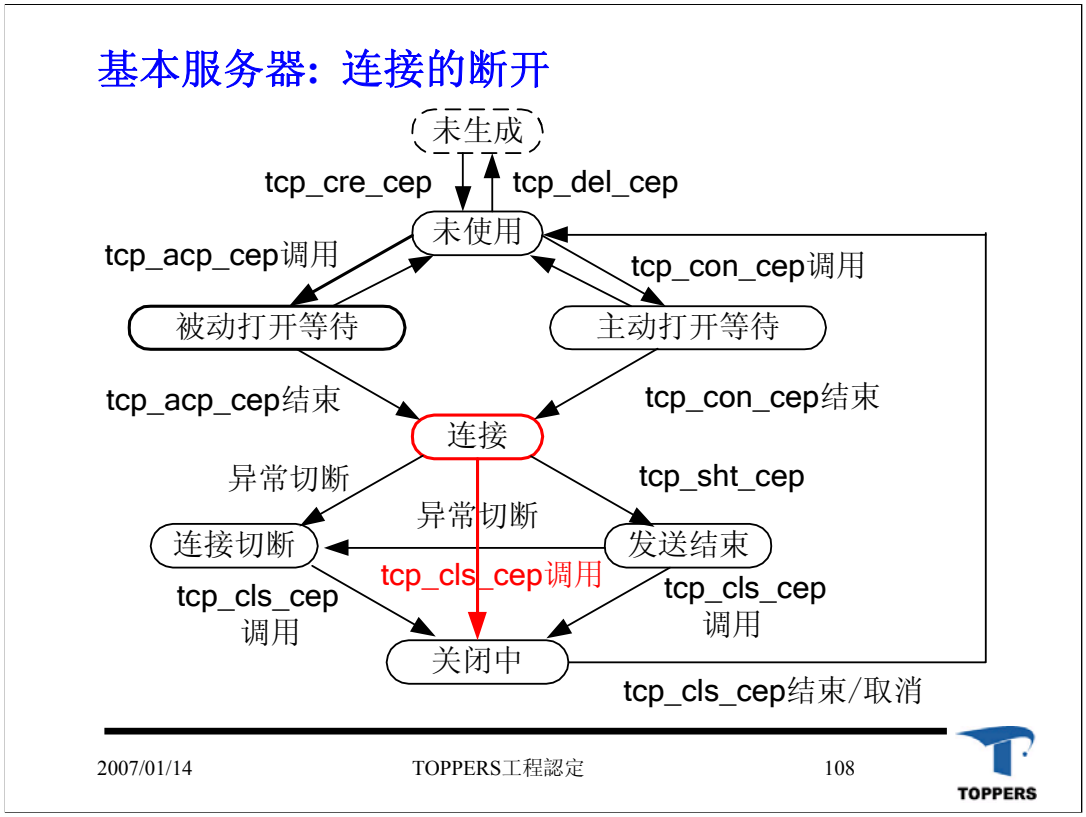
ID	cepid	TCP通信端点ID
VP	data	存放接收数据的空间的首地址
INT	len	要接收的数据的长度
TMO	tmout	超时指定

【返回参数】

ER	ercd	取出的数据的长度/错误代码

【错误代码】

正值	正常结束（取出的数据的长度）
0	数据结束（正常断开连接）
负值	错误代码


基本服务器：连接的断开
未生成
tcp_cre_cep
tcp_del_cep
未使用
tcp_acp_cep调用
tcp_con_cep调用
被动打开等待
主动打开等待
tcp_acp_cep结束
tcp_con_cep结束
连接
异常切断
tcp_sht_cep
异常切断
连接切断
发送结束
tcp_cls_cep调用
tcp_cls_cep
调用
tcp_cls_cep
调用
关闭中
tcp_cls_cep结束/取消
2007/01/14
TOPPERS工程認定
108
TOPPERS

说明一下连接的断开。

基本服务器：断开连接

- 断开连接**API**

```
tcp_cls_cep(TCP通信端点ID, 超时)
```

- 基本服务器

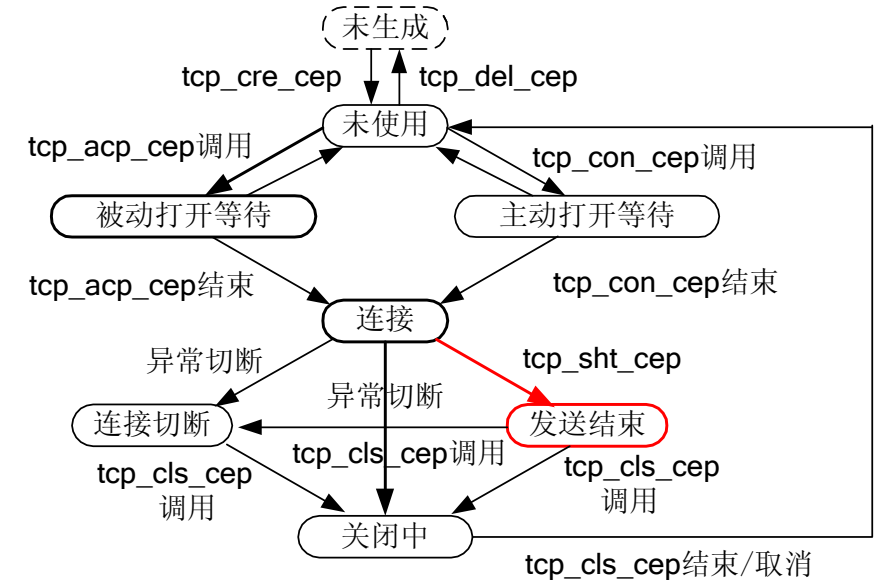
```
tcp_cls_cep(TCP_SRV_CEPID, TMO_FEVR)
```


以下是tcp_srv.c的第78行到第87行。断开连接使用tcp_cls_cep服务调用。在基本服务器中，客户端正常断开连接或发生错误时断开连接。执行tcp_cls_cep服务调用后，显示当前时间并等待被动打开。

```
        err_fin:
                if ((error = tcp_cls_cep(TCP_SRV_CEPID, TMO_FEVR)) != E_OK)
                syslog(LOG_NOTICE, "[TES:%02d CLS] error: %s", TCP_SRV_CEPID,
itron_strerror(error));

                get_tim(&now);
                syslog(LOG_NOTICE, "[TES:%02d FIN] finished:   %6d",
                                                TCP_SRV_CEPID, now / SYSTIM_HZ);
        }
}
```

基本服务器：发送结束通知



基本服务器上没有安装发送结束通知。发送结束通知通过tcp_shut_cep服务调用实现的。

基本服务器：tcp_sht_cep 与tcp_cls_cep 的不同

- tcp_cls_cep
  –关闭通信端点，关闭后就不能再使用通信端点
- tcp_sht_cep
  –结束数据发送，结束后就不能再发送数据，但可以接收数据
  –用于通知对方发送已结束
- 用于结束通信的客户端・服务器间的协调
  –服务器无法判断客户端的请求是否结束，所以不能断开连接
  –客户端知道请求已结束，但不知道服务器端的数据是否已全部到达
  –客户端为通知服务器请求已结束，使用tcp_sht_cep

通过tcp_cls_cep服务调用关闭通信端点，关闭后将不能再使用通信端点。

使用tcp_cls_cep服务调用会在TCP通信端点变成未使用状态后返回，所以返回后可以再次使用TCP通信端点。通过tcp_sht_cep服务调用结束数据发送，结束后将不能再发送数据。tcp_sht_cep只做断开的准备，在API内并不会变为等待状态。此服务调用用于通知对方发送已结束。

服务器程序无法判断客户端是否结束了要求，所以不能主动断开连接。客户端知道自己的要求已结束，但无法判断服务器的数据是否已全部到达。因此，客户端为了通知服务器要求已结束，使用tcp_sht_cep服务调用。

TCP服务器程序设计

1. 基本服务器
2. 回显服务器
3. 设备操作服务器
4. 多任务服务器

回显服务器：ECHO_SRV

- 修改基本服务器，创建将接收到的字符加1并返回的回显服务器
- 需要给基本服务器添加以下处理
  - 接收字符的加1
  - 数据的发送
- 请修改之前创建的MY_SRV并创建回显服务器

修改基本服务器并创建回显服务器。

回显服务器将接收到的客户端的数据字符加1并返回给客户端。

仅tcp_srv.c。修改基本服务器的程序，修改为：将接收到的数据字符加1并写入发送缓冲器，然后发送。

回显服务器：发送数据

- 发送数据API

```
tcp_snd_dat(
        TCP通信端点ID,
        缓冲器, 发送大小,
        超时)
```

- 返回值
  - 发送字符数
  - 负值时出现错误
- 发送缓冲器
  - 确保用于创建发送字符串的缓冲器

```
static UB       sbuffer[BUF_SIZE];
```

2007/01/14 TOPPERS工程認定 114

TOPPERS

发送数据使用tcp_snd_dat服务调用。以下是ITRON TCP/IP API规范。

【C语言API】

ER ercd = tcp_snd_dat(ID cepid, VP data, INT len, TMO tmout);

【参数】

ID	cepid	TCP通信端点ID
VP	data	发送数据的首地址
INT	len	要发送的数据的长度
TMO	tmout	超时时间

【返回参数】

ER	ercd	放入发送缓冲器的数据的长度/错误代码

【错误代码】

正值	正常结束（放入到发送缓冲器中的大小）
负值	错误代码

在tcp_srv.c中追加用于创建发送字符串的缓冲器。

回显服务器：数据发送

- 接收到的字符串的处理

```
for(i = 0; i < rlen; i++){
    if(rbuffer[i] >= ' ')
        sbuffer[i] = rbuffer[i] + 1;
    else
        sbuffer[i] = rbuffer[i];
}
```

- 发送处理

```
char *psdata;
....
slen_remain = rlen;
psdata = sbuffer;
do{
        if ((slen = tcp_snd_dat(TCP_SRV_CEPID,
                                psdata, slen_remain, TMO_FEVR)) < 0) {
                syslog(LOG_NOTICE, "[TES:%02d SND]  .....);
                goto err_fin;
        }
        slen_remain -= slen;
        psdata += slen;
}while(slen_ remain > 0);
```


TOPPERS

给接收到的字符加1。为了使控制代码保持不变，将rbuffer[i]比空格代码大的情况作为对象复制到发送缓冲器 (sbuffer)上。发送时使用tcp_snd_dat服务调用。将发送数据量设定为slen_remain，将tcp_snd_data的返回值放到slen中。当slen大于等于0时，从slen_rmain中减去slen，计算还没有发送的数据量。只按照已发送的数据来前进psdata的指针。slen_remain大于0时，由于还有没发送完的数据，所以再次使用tcp_snd_dat服务调用进行发送。

TCP服务器程序设计

1. 基本服务器
2. 回显服务器
3. 设备操作服务器
4. 多任务服务器

TCP客户端程序

用ITRON TCP/IP API规范
创建TCP客户端程序

- 已完成的程序在以下目录中
  - **./OBJ/AKIH8_3069F/**
    - **TCP_CLIENT ： 基本客户端**
    - **DIP_MONITOR ： DIP状态通知客户端**
- 在Windows下运行的TCP服务器在以下目录中
  - **./environment/tcp_serVer/tcp_serVer.exe**

在本章中，创建将板作为用户、将PC作为服务器的系统。TCP客户端的完成程序在./OBJ/AKIH8_3069F中的TCP_CLEINT和DIP_MONITOR两个目录下。PC上需要Windows用的服务器程序。此程序作为服务器进行启动，并将客户端发送的数据显示到控制台上。

设备操作服务器：DEV_SRV

- 修改回显服务器，创建操作设备的服务器
- 接收数据
  - 为0时LED1、2同时熄灭
  - 为1时LED1亮灯、LED2熄灭
  - 为2时LED1熄灭、 LED2亮灯
  - 为3时LED1、2同时亮灯
  - 为其他值时忽略
- 接收到数据后用4个字符的字符串返回DIP开关的状态
  - 如：为ON、 OFF、 ON 、OFF时返回1010
- 设备的操作使用第一天创建的device.c
- 不要忘记进行硬件的初始化！

2007/01/14 TOPPERS工程認定 118

TOPPERS

修改回显服务器，创建设备操作服务器。如果不能顺利创建回显服务器，请参考正确的例子重新创建。

```
$ cd ..                          ‘移到./OBJ/AKIH8_3068F目录下
$ rm –rf MY_SRV                  ‘删除MY_SRV
$ cp –r ECHO_SRV MY_SRV          ‘复制ECHO_SRV
```

改造的规范：

1）从telnet发送0到3的数字，服务器接收数据

为0时LED1和LED2熄灭

为1时LED1亮灯、LED2熄灭

为2时LED1熄灭、LED2亮灯

为3时LED1和LED2亮灯

2）LED处理后读入DIP-SW的状态，根据ON、OFF状态进行操作。

返回对应DIP-SW1到4的4个字符的数字（0或1）。

DIP-SW为ON时返回1，为OFF时返回0。

关于设备的操作，从./OBJ/AKIH8_3069F/device复制第一天创建的device.c来进行使用。

设备操作服务器：初始化例程

- 初始化例程
  - 设备的初始化

```
void
device_init(VP_INT exinf){
    initial_switch();    /* 开关的初始化 */
    initial_led();       /* LED的初始化 */
}
```

- 作为初始化例程进行调用

```
ATT_INI({TA_HLNG, 0, device_init});
```

- **Makefile的编辑**
  - 将device.c添加到编译对象中

```
UTASK_COBJS = $(UNAME).o device.o
```

2007/01/14 TOPPERS工程認定 119

TOPPERS

将设备的初始化例程device_init()追加到tcp_srv.c中，将device_init的原型声明追加到tcp_srv.h中。

为了作为初始化例程进行调用，在tcp_srv.cfg中追加ATT_INI静态API。

将device.o添加到Makefile的UTASK_COBJS中，并添加到编译对象中。

设备操作服务器：LED操作函数

- 接收到的字符作为参数来操作**LED**

```
void
led_control(char c){
        switch(c){
          case '0':
              set_led(LED1,OFF);
              set_led(LED2,OFF);
              break;
          case '1':
              set_led(LED1,ON);
              set_led(LED2,OFF);
              break;
             .......
          default:
              break;
        }
}
```


用接收到的字符将操作LED的函数led_control();添加到tcp_srv.c中。参数是接收到的字符串的第一个字符。

设备操作服务器：获取DIP状态

- 获取DIP开关的状态并转换成字符串

```
int dip_monitor(char *buffer){
        if (get_switch(SW1) == ON) {
            buffer[0] = '1';
        }else{
            buffer[0] = '0';
        }
        if (get_switch(SW2)  == ON) {
            buffer[1] = '1';
        }else{
            buffer[1] = '0';
        }
        .....
         buffer[4] = '¥n';
         buffer[5] = '¥r';
        buffer[6] = '¥0';
        syslog(LOG_NOTICE, "Dip state is %s", buffer);
        return 6;
}
```


TOPPERS

在tcp_srv.c中添加获取DIP开关的状态并转换成字符串的函数dip_monitor();。参数是指向字符的指针，它将DIP-SW的状态写入到字符串空间中并通过syslog语句将内容显示到控制台上。作为返回值，返回作成的字符串的长度。

设备操作服务器：数据的发送接收

- 接收到数据后操作**LED**，获取**DIP**的状态并发送给客户端

```
rbuffer[rlen] = '¥0';
syslog(LOG_NOTICE, "Recive Data %s", rbuffer);
/* LED的操作 */
led_control(rbuffer[0]);
/* DIP状态的获取 */
slen_remain = dip_monitor(sbuffer);
psdata = sbuffer;
while(slen_remain > 0){
    if ((slen = tcp_snd_dat(TCP_SRV_CEPID, psdata,
                                        slen_remain, TMO_FEVR)) < 0) {
        syslog(LOG_NOTICE, "[TES:%02d SND] .....);
        goto err_fin;
    }
    slen_remain -= slen;
    psdata += slen;
}
```


修改tcp_srv.c的tcp_srv_tsk任务。

TCP服务器程序设计

1. 基本服务器
2. 回显服务器
3. 设备操作服务器
4. 多任务服务器

多任务服务器：MUL_SRV

- 基于设备操作服务器进行修改，修改为在一个接收口接收两个客户端的连接请求
- 可以在一个TCP接收口等待多个任务
- 按各个任务生成TCP通信端点
- 生成两个任务，代码是共享的
- 在任务中，将个别信息放入到数组中，并从索引中获取扩展信息（exinf）（参照sample1）
- 提供两个作为全局变量的缓冲器等
- 关于LED的操作，进不进行互斥控制都可以

修改设备操作服务器，创建可以对应两个客户端的多任务服务器。修改方法是添加在一个TCP接收口中以两个任务对应两个TCP通信端点的处理。

修改tinet_tcp_srv.cfg、tcp_srv.c和tcp_srv.h，生成两个TCP通信端点。修改tcp_srv.cfg和tcp_srv.c，生成两个tcp_srv_tsk。将发送接收缓冲器作为数组进行双重化，用两个任务实现TCP通信端点与发送接收缓冲器使用不同的任务。

关于LED的操作，即使不进行排他控制也是可以的。

多任务服务器： TINET结构(tinet_tcp_srv.cfg)

- 增加一个**TCP**通信端点
  - 分别提供缓冲器

```
TCP_CRE_CEP (TCP_SRV_CEPID1, {
        0,
        tcp_srv_cep_sbuf1, TCP_SRV_CEP_SBUF_SIZE,
        tcp_srv_cep_rbuf1, TCP_SRV_CEP_RBUF_SIZE,
        NULL
        } );
```

```
TCP_CRE_CEP (TCP_SRV_CEPID2, {
        0,
        tcp_srv_cep_sbuf2, TCP_SRV_CEP_SBUF_SIZE,
        tcp_srv_cep_rbuf2, TCP_SRV_CEP_RBUF_SIZE,
        NULL
        } );
```


修改tinet_tcp_srv.cfg，追加一个TCP通信端点。此时为了区别各通信缓冲器，对TCP_SRV_CEPID1用的发送接收缓冲器进行如下修改。

发送窗口缓冲器：tcp_srv_cep_sbuf1

接收窗口缓冲器：tcp_srv_cep_rbuf1

TCP_SRV_CEPID2用的发送接收缓冲器为以下设定。除此之外，与TCP_SRV_CEPID1相同。

发送窗口缓冲器：tcp_srv_cep_sbuf2

接收窗口缓冲器：tcp_srv_cep_rbuf2

多任务服务器：确保缓冲器

- 发送接收缓冲器（**TCP窗口缓冲器**）：**tcp_srv.c**
  – 确保两个

```
UB tcp_srv_cep_sbuf1[TCP_SRV_CEP_SBUF_SIZE];
UB tcp_srv_cep_rbuf1[TCP_SRV_CEP_RBUF_SIZE];
UB tcp_srv_cep_sbuf2[TCP_SRV_CEP_SBUF_SIZE];
UB tcp_srv_cep_rbuf2[TCP_SRV_CEP_RBUF_SIZE];
```

- 任务中使用的发送接收缓冲器
  – 确保两个

```
static UB   rbuffer[2][BUF_SIZE];
static UB   sbuffer[2][BUF_SIZE];
```


将TCP通信端点使用的发送接收窗口缓冲器追加到tcp_sev.c中。同样地，将发送接收窗口缓冲器的原型声明追加到tcp_srv.h中。

对tcp_srv.c中的两个服务器任务使用的发送接收缓冲器进行数组化。

多任务服务器：TCP通信端点的信息获取

- 提供全局变量用来保存**TCP**通信端点的**ID**
- 从索引中获取传递给任务的参数（**exifnf**）信息

```
int tcp_cep_id[] = {TCP_SRV_CEPID1, TCP_SRV_CEPID2};
```

- 生成任务
  - 生成两个

```
CRE_TSK(TCP_SRV_TASK1, {
    TA_HLNG|TA_ACT, 0,
    tcp_srv_task,
    TCP_SRV_MAIN_PRIORITY,
    TCP_SRV_STACK_SIZE,
    NULL
    });
```

```
CRE_TSK(TCP_SRV_TASK2, {
    TA_HLNG|TA_ACT, 1,
    tcp_srv_task,
    TCP_SRV_MAIN_PRIORITY,
    TCP_SRV_STACK_SIZE,
    NULL
    });
```


将传递给两个任务的tcp_srv_task的参数设定为0、1。为了实现各任务分别获取TCP通信端点，要提供以TCP通信端点ID为初始数据的全局变量的数组。

多任务服务器：程序

- **TCP通信端点ID在任务启动时通过exinf获取**

```
INT  cepid;
cepid = tcp_cep_id[(INT)exinf];
```

```
tcp_acp_cep(cepid, TCP_SRV_REPID, &dst, TMO_FEVR)
```

- **缓冲器也通过exinf指定**

```
tcp_rcv_dat(cepid, rbuffer[(INT)exinf], BUF_SIZE - 1, TMO_FEVR)
```

```
/* LED的操作 */
led_control(rbuffer[(INT)exinf][0]);
/* DIP状态的获取 */
slen_remain = dip_monitor(sbuffer[(INT)exinf]);
```


关于tcp_srv_tsk函数，在任务启动时由exinf通过全局数组在两个任务内分别将TCP通信端点ID放到cepid中。而且，发送接收缓冲器也是通过exinf的数组分别指定缓冲器并进行被动打开、接收和发送。

TCP服务器端程序设计

1. TCP/IP基础
2. ITRON TCP/IP规范
3. TINET（TCP/IP栈）
4. TCP服务器端程序
5. TCP客户端程序
6. 总结

TCP客户端程序

用ITRON TCP/IP API规范
创建TCP客户端程序

- **已完成的程序在以下目录中**
  - **./OBJ/AKIH8_3069F/**
    - **TCP_CLIENT ： 基本客户**
    - **DIP_MONITOR ： DIP状态通知客户端**
- **在Windows下运行的TCP服务器在以下目录中**
  - **./environment/tcp_serVer/tcp_serVer.exe**

在本章中，创建将板作为客户端、将PC作为服务器的系统。TCP客户端的完成程序在./OBJ/AKIH8_3069F中的TCP_CLEINT和DIP_MONITOR两个目录下。PC上需要Windows用的服务器程序。此程序作为服务器进行启动，并将客户端发送的数据显示到控制台上。

TCP客户端程序设计

1. 基本客户端
2. **DIP状态通知客户端**

基本客户端： TCP_CLIENT

- 执行后就开始等待串行控制台的按键输入，按键输入后尝试连接到IP地址为192.168.11.6、端口号为1234的服务器
- 成功连接后读入串行控制台的输入信息，直到输入回车
- 将读入的信息发送给服务器
- 如果输入Ctrl-C或“Q”，就会向服务器发送结束通知（“_end_”）并等待服务器发出的结束通知
- 在由服务器断开连接并收到服务器的结束通知后关闭端口

TOPPERS

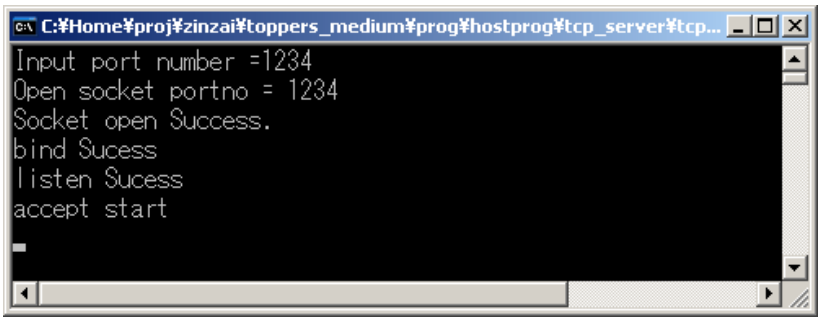
TCP_CLIENT程序将板作为客户端访问PC。如果程序启动，就会显示“[TCP CLIENT] Push any key to connent”并进入键等待状态。通过键输入对服务器进行主动打开。如果与服务器建立了连接，就会将输入的字符串发送给服务器。服务器显示字符串。

输入Ctrl-C或“Q”来结束客户端程序，这样就会向服务器发送字符串“_end_”，客户端等待接收“_end_”。

如果服务器接收到“_end_”，就会将“_end_”返回给客户端，并停止运行然后关闭。客户端接收到“_end_”后关闭TCP通信端点。

基本客户：服务器

- 运行程序目录/environment/tcp_serVer/tcp_serVer.exe
- 开始运行后要输入端口号，输入“1234”后回车
- 如果成功打开端口，就会显示“accpet start”



TOPPERS

在PC上运行服务器程序。请双击/environment/tcp_serVer/目录下的运行程序开始运行。运行后会显示“Input port num =”。请在此处输入“1234”并按下Enter键。通过以下步骤进入被动打开状态。

```
socket();    ‘生成socket
bind();      ‘将socket分配给端口号
listen();    ‘进入可以连接到已进行了分配的端口号的状态
accpet();    ‘等待接收连接
```

建立客户端的连接后会显示“accpet start”并进入接收等待状态。

基本客户端：构建

- 编译程序，确认运行

```
$ cp –r TCP_CLIENT MY_CLIENT
$ cd MY_CLIENT
$ make realclean
$ ls
Makefile    tcp_client.c   tcp_client.h  tinet_tcp_client.cfg
route_cfg.c  tcp_client.cfg  tinet_app_config.h
$ make depend
$ make
```

- 连接到服务器，将控制台上输入的数据发送给服务器，并确认服务器程序发送的数据是否显示

在./OBJ/AKIH8_3069F/目录下，用cp命令将TCP客户端程序目录下的TCP_CLIENT复制到MY_CLIENT上。移到MY_CLIENT目录下，以与服务器程序相同的步骤生成TCP客户端程序。

通过TeraTerm将生成的jsp.srec下载到板上并运行，请按步骤确认是否能连接及发送数据。

基本客户端：文件构成

- **Makefile** ：**Make文件**
- **route_cfg.c** ：**静态路由选择设定文件**
- **tinet_app_config.h** ：**配置定义文件**
- **tinet_tcp_clinet.cfg** ：**TINET结构**
- **tcp_client.h** ：**用户程序Header**
- **tcp_client.c** ：**用户程序源代码**
- **tcp_client.cfg** ：**JSP配置文件**

以上是TCP_CLIENT中的文件。

基本客户端: TINET结构（tinet_tcp_client.cfg）

- 因为是客户端应用程序，所以不用创建TCP接收口，只创建一个TCP通信端点

```
TCP_CRE_CEP(TCP通信端点ID, {
        通信端点属性,
        发送缓冲器, 发送缓冲器容量,
        接收缓冲器, 接收缓冲器容量,
        回调函数
})
```

```
TCP_CRE_CEP (TCP_CLIENT_CEPID, {
        0,
        tcp_client_cep_sbuf, TCP_CLIENT_CEP_SBUF_SIZE,
        tcp_client_cep_rbuf, TCP_CLIENT_CEP_RBUF_SIZE,
        NULL
} );
```

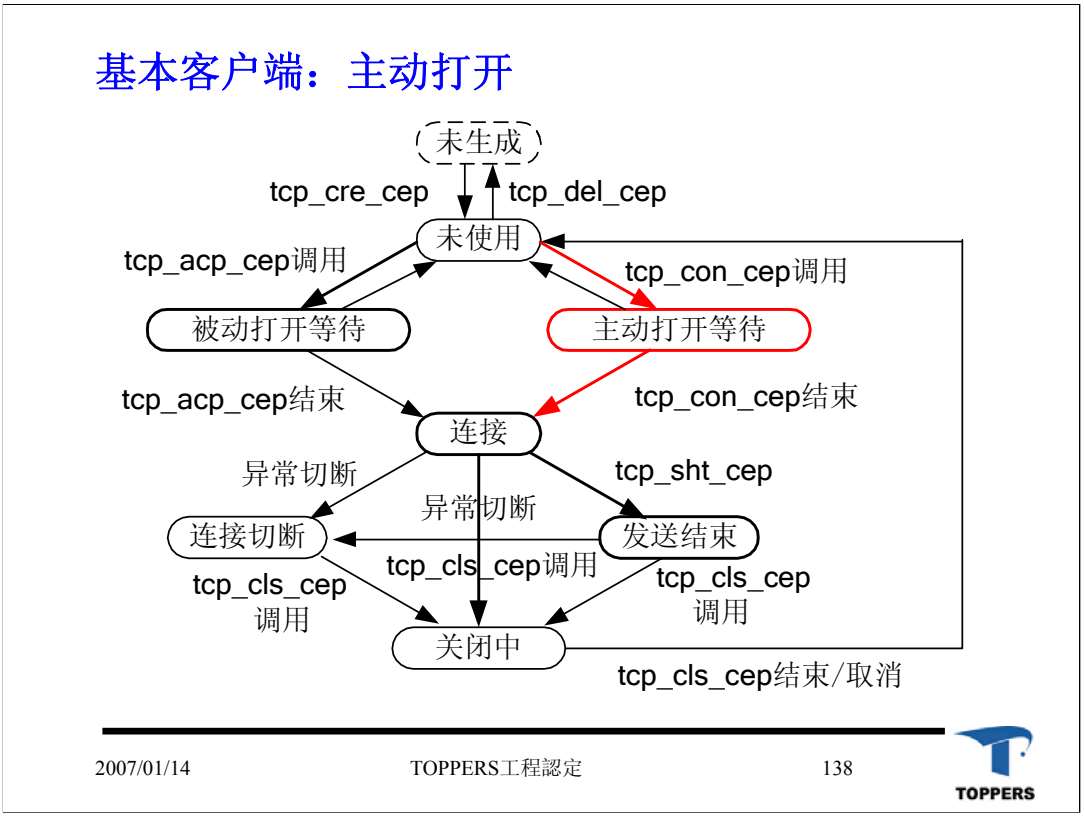

下面介绍一下TINET配置文件（tinet_tcp_client.cfg）。TCP客户端程序与TCP服务器程序不同，它不需要进行接收等待。因此，tinet_tcp_client.cfg也不需要生成TCP接收口，只需创建TCP通信端点。生成的设定与TCP服务器基本相同。

基本客户端：JSP配置文件

- 生成一个任务

```
CRE_TSK(TCP_CLIENT_TASK, {
        TA_HLNG|TA_ACT,
        0,
        tcp_client_task,
        TCP_CLIENT_MAIN_PRIORITY,
        TCP_CLIENT_STACK_SIZE,
        NULL
});
```


JSP配置文件（tcp_client.cfg）中记述了用于生成运行客户端程序的任务（tcp_client_task）的静态API。



介绍主动打开等待的步骤。以后会以tcp_client.c程序为对象进行说明。

基本客户端：主动打开

- 主动打开API

```
tcp_con_cep(
        TCP通信端点ID,
        自己的IP地址/端口地址结构体,
        对方的IP地址/端口地址结构体,
        超时)
```

- 基本客户

```
T_IPV4EP    dst;
dst.ipaddr = DEST_ADDR;
dst.portno = DEST_PORT;
tcp_con_cep(TCP_CLIENT_CEPID,
            (T_IPV4EP*)NADR, &dst,
             TMO_FEVR)
```

IP及端口号取决于协议栈

2007/01/14 TOPPERS工程認定 139

TOPPERS

通过tcp_con_cep服务调用进行主动打开。以下是tcp_client.c的第70行到第86行。通过第80行的tcp_con_cep服务调用进行主动打开。如果与服务器建立了连接，就会返回E_OK并移到第85行的syslog。产生错误没有返回E_OK时，再次执行tcp_con_cep服务调用来进行主动打开。

```
while (TRUE) {
        /*
         * 连接到端点
         */
        do{
                syslog(LOG_NOTICE, "[TCP CLIENT] Push any key to connect");
                syscall(serial_rea_dat(TASK_PORTID, &c, 1));
        syslog(LOG_NOTICE, "[TCP CLIENT CEP] connected to %s:%d", addr, dst.portno);
                if (error = tcp_con_cep(TCP_CLIENT_CEPID, (T_IPV4EP*)NADR, &dst,
TMO_FEVR) != E_OK) {
                        syslog(LOG_NOTICE, "[TCP CLIENT CEP] error: %s",
itron_strerror(error));
                }
        }while(error != E_OK);
        syslog(LOG_NOTICE, "[TCP CLIENT CEP] connection establish.");
```

基本客户端：断开连接

- 先断开连接的一方必须等待2MSL
- 在TINET中，由于与TCP通信端点的连接是1对1对应的，所以TCP通信端点等待2MSL
- 在Windowns下连接本身就要等待2MSL，socket消失
- 所以这次要从Windows侧断开连接

作为TCP的基本功能，要求建立连接的一方必须等待2MSL。MSL（Max Segment Lifetime）是最大段生存时间，它表示包在网络中能够停留的最长时间。换句话说，超过了MSL包还没有到达时，可以认为包由于某种原因丢失了。也就是说，为了断开，如果发行FIN后作为往返待机2MSL，就可以判断为已经没有接收数据。

一般2MSL约为120秒，但根据实装的情况多数都是更短的时间。如果是TINET，好像是用TCP_TVAL_KEEP_COUNT(8)×TCP_TVAL_KEEP_INTERVAL（7.5秒）设定为60秒。

在TCP客户端，由于是Windows下的服务器来断开连接，所以服务器端要等待2MSL。在Windows的socWin接口中，通过accpet();建立连接时生成通信用的socket。断开连接时此socket等待2MSL后消失。

基本客户端：断开连接

- 断开连接前发送结束包

```
#define END_STAR "_end_"
if (c == '¥003' || c == 'Q'){
      if ((slen = tcp_snd_dat(TCP_CLIENT_CEPID,
                              END_STAR, 5, TMO_FEVR)) < 0) {
         syslog(LOG_NOTICE, "[TCP CLIENT SND] ......);
            goto err_fin;
      }}
```

- 等待从服务器端发来结束包
- 如果接收到结束包就关闭端口

```
tcp_cls_cep(TCP_CLIENT_CEPID, TMO_FEVR)
```


TOPPERS

为了对断开进行说明，下面记述了tcp_client.c的第87行到第106行。如果从控制台输入Ctrl-C或“Q”，就会使用tcp_snd_dat服务调用将END_STAR(=“_end_”)发送到服务器（第97行的判定）。等待接收服务器发送的“_end_”，使用tcp_cls_cep服务调用关闭TCP通信端点。

```
        /*
         * 数据的发送接收
         */
        while (TRUE) {
                /*
                 * 读入字符串，直到输入Enter
                 */
                slen = 0;
                do{
              syscall(serial_rea_dat(TASK_PORTID, &c, 1));
              if (c == '¥003' || c == 'Q'){
                        /* 发送结束通知 */
                        if ((slen = tcp_snd_dat(TCP_CLIENT_CEPID, END_STAR, 5, TMO_FEVR)) < 0)
{
                                syslog(LOG_NOTICE, "[TCP CLIENT SND] can not send end data",
                                itron_strerror(slen));
                        }
                        goto err_fin;
              }
              sbuffer[slen++] = c;
                }while(c != '¥r');
```

TCP客户端程序设计

1. 基本客户端
2. DIP状态通知客户端

DIP状态通知客户端：DIP_MONITOR

- 依据基本客户端，创建连接到服务器后周期性发送DIP开关状态的程序
- 服务器端的程序与基本客户端使用相同的程序
- 关于DIP开关的状态，在主任务（tcp_client_task）以外再创建一个DIP状态通知任务（dip_monitor_task），通过此任务周期性发送DIP开关的状态
- 主任务连接到服务器后启动DIP状态通知任务，在串行控制台输入Ctrl-C或Q后结束DIP状态通知任务，并断开与服务器的连接
- 不要忘记进行硬件的初始化！

修改基本客户端程序，创建DIP状态通知客户端程序。服务器端的程序与基本客户端使用相同的程序，无变更。

DIP状态通知客户端由两个任务构成。主任务（tcp_clent_task）连接到服务器后启动另一个任务DIP状态通知任务（dip_monitor_task），并等待控制台输入Ctr l -C或“Q”。在输入后结束DIP状态通知任务并要求断开与服务器的连接。

如果启动DIP状态通知任务，就会周期性读取DIP-SW的状态，并将其转换成表示OFF、ON状态的“0”或“1”的4个字符串发送到服务器。周期为2秒。

DIP状态通知客户端：JSP配置文件

- 在中止状态下新生成**DIP**状态通知任务

```
CRE_TSK(DIP_MONITOR_TASK, {
        TA_HLNG,
        0,
        dip_monitor_task,
        DIP_MONITOR_PRIORITY,
        DIP_MONITOR_STACK_SIZE,
        NULL
});
```


在JSP的配置文件中追加DIP状态通知任务（dip_monitor_task）的生成API。

DIP状态通知客户端：DIP状态通知任务

```
void dip_monitor_task(VP_INT exinf){
      int   slen;
      syslog(LOG_NOTICE, "[DIP MONITOR] started.");
      while(TRUE) {
            if (get_switch(SW1) == ON) {
                  sbuffer[0] = '1';
            }else{
                  sbuffer[0] = '0';
            }
            .......
            if ((slen = tcp_snd_dat(TCP_CLIENT_CEPID,
                                    &sbuffer, 4, TMO_FEVR)) < 0) {
            syslog(LOG_NOTICE, "[TCP CLIENT SND] ......);
            }
            dly_tsk(DIP_MONITOR_INTERVAL);
      }
}
```


在tcp_client.c中追加dip_monitor_task();函数。

DIP状态通知客户端: 主任务

- 连接到服务器后，使DIP状态通知任务进入执行状态

```
syscall(act_tsk(DIP_MONITOR_TASK));
```

- 如果接收到串行控制台的Ctrl-C或Q，就结束DIP状态通知任务

```
do{
    syscall(serial_rea_dat(TASK_PORTID, &c, 1));
}while(c != '¥003' && c != 'Q');
syslog(LOG_NOTICE,"[TCP CLIENT] terminate dip monitor task");
syscall(ter_tsk(DIP_MONITOR_TASK));
```


在tcp_client.c的主任务（tcp_client_task）中追加连接到服务器后使DIP状态通知任务进入执行状态的act_tsk服务调用。并追加串行控制台输入Ctr l -C或“Q”时结束DIP状态通知任务的ter_tsk服务调用。

删除不需要的数据与发送。

尝试创建碗面定时器的TOPPERS/JSP版。

由于在此开发环境下无法使用调试器，所以尝试使用任务监视器、系统LOG的转储功能进行调试。

第1天的总结

学习以下几项：

- **AKI-H8/3069F与TOPPERS/JSP Kernel的构建方法**
- 外围设备的程序设计方法
- **ITRON规范**
- 使用**ITRON API**进行程序设计
  - 任务相关的**API**
  - 同步附属功能
- 碗面定时器
  - 设计
  - 编码

第2天的总结

- 学习以下几项：
  - **TCP/IP概要**
  - **ITRON TCP/IP规范**
  - **TINET**
  - **TCP服务器程序设计**
  - **TCP客户端程序设计**

教材的编写

参考资料

- 「TCP/IPによるネットワーク構築 Vol.I 原理・プロトコル・アーキテクチャー」
  Douglas Comer著/村井純・楠木博之 訳

- 「平成16年度 マイコン応用研修 TINETによる TCP/IP プログラミング」
  苫小牧工業高等専門 情報工学科

*参考资料中文名（意译）

《TCP/IP网络构建Vol.I原理・协议・架构》
Douglas Comer著/村井纯・楠木博之译

《平成16年度微处理器应用研修TINET的TCP/IP程序设计》
苫小牧工业高等专门 情报工学科　阿部司