TCP/IP LEAN

SECOND EDITION

Web Servers for Embedded Systems

- Understand the inner workings of TCP/IP
- Implement dynamic content generation and client/server data transfer capabilities

Jeremy Bentham

CMPBooks

TCP/IP Lean

Web Servers for Embedded Systems

Second Edition

Jeremy Bentham

CMP Books
Lawrence, Kansas 66046

CMP Books
CMP Media LLC
1601 West 23rd Street, Suite 200
Lawrence, Kansas 66046
USA
www.cmpbooks.com

Acquisitions Editor:	Robert Ward
Managing Editor:	Michelle O'Neal
Editor:	Rita Sooby
Layout production:	Kris Peaslee
Cover art:	Robert Ward
Cover design:	Damien Castaneda

Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, California 94710
1-800-788-3123
www.pgw.com

ISBN: 1-57820-108-X

CMP*Books*

Table of Contents

Preface xi
The Lean Plan xi
Embedded Systems xii
The Hardware xiii
The Network xiii
The Operating System xiv
The Development Environment xiv
The Software xv
Acknowledgments xv

Chapter 1 Introduction 1
The Lean Plan 1
Getting Started 2
Software Introduction 5
Network Hardware 5
Device Drivers 8
Configuration File Format 14
Process Timer 14
State Machines 17
Buffering 21
Coding Conventions 29

Chapter 2 Introduction to Protocols: SCRATCHP 31
Overview 31
Protocol 32
SCRATCHP Services 34
Logical Connections 36
Packet Format. 38
Addressing 42
Protocol Identification 43
Reception and Transmission 46
Implementation. 49
Summary 68

Chapter 3 Network Addressing and Debugging 71
Overview 71
Internetworks 71
IP Addresses 74
Address Resolution 75
ARP Scanner. 77
Using ARPSCAN for Network Debugging 84
Ethernet 2 89
IEEE 802.3 Networks 90
Summary 93

Chapter 4 The Network Interface: IP and ICMP 95
Overview 95
TCP/IP Stack. 95
Internet Control Message Protocol 110
Ping Implementation. 112
Router Implementation. 122
Summary 131

Chapter 5 User Datagram Protocol: UDP 135
Overview 135
Ports and Sockets 135
Datagram Format 138
UDP Checksum. 140
UDP Utility. 142
Summary 152

Chapter 6 Transmission Control Protocol: TCP 155
Overview 155
TCP Concepts 156
TCP Implementation 169
TCP Application — Telnet 188
Telnet Implementation 190
Using Telnet 199
Conclusion 203

Chapter 7 Hypertext Transfer Protocol: HTTP 207
Overview 207
HTTP GET Method 207
Simple Web Server 211
Introducing HTML 217
State Machine Implementation 226
Summary 235

Chapter 8 Embedded Gateway Interface: EGI. 237
Overview 237
Interactive Displays 237
Standard CGI interface 244
EGI Implementation 249
Summary 267

Chapter 9 Miniature Web Server Design 269
Overview 269
Microcontroller Software Development 270
Hardware 270
Development Environment 274
Software Techniques 275
Web Server Protocols 278
Summary 290

Chapter 10 TCP/IP on a PICmicro® Microcontroller. . 291
Overview 291
Peripherals 291
Block Diagram 294
Circuit Diagram 294
Low-Level Software 296

SLIP and IP Drivers. 303
ICMP 319
TCP 321
Summary 329

Chapter 11 PWEB: Miniature Web Server for the PICmicro® 331

Overview 331
Web Server 331
ROM File System 336
Using the PWEB Server 349
Dynamic Content 351
Dynamic Web Pages 355
Summary 367

Chapter 12 ChipWeb — Miniature Ethernet Web Server 369

Overview 369
Hardware 370
Ethernet Driver 375
LCD Driver 383
Other Drivers 386
Protocols 386
Protocol Debugging 398
User Interface 398
Configuration 404
Conclusion 409

Chapter 13 Point-to-Point Protocol: PPP 411

Overview 411
Design of PPP 412
Protocol Components 415
Sample PPP Negotiation 420
PPP Implementation 426
Summary 433

Chapter 14 UDP Clients, Servers, and Fast Data Transfer 435
Overview 435
Client–Server Networking 435
Peer-to-Peer Networking 437
Beyond the Web Server. 438
Buffer Enhancements 438
IP and ICMP Processing 445
UDP Servers 448
UDP Time Client 451
High-Speed Data Transfer 457
Hardware 458
Software 461
Summary 467

Chapter 15 Dynamic Host Configuration Protocol: DHCP. 471
Overview 471
DHCP Methodology 472
Sample Transaction 477
DHCP Implementation. 481
Summary 487

Chapter 16 TCP Clients, SMTP, and POP3 Email 489
Overview 489
TCP Client Techniques. 490
TCP Client Implementation 494
SMTP Email Client. 502
POP3 Email Client 509
Summary 515

Appendix A Configuration Notes 517
Network Configuration 517
Addressing 519
Testing the Network. 519
Windows SLIP Configuration. 520

Appendix B Resources 523
Publications 523
Hardware 524
Software 524

Appendix C Software on the CD-ROM 527
ARPSCAN 528
DATAGRAM 529
NETMON 529
PICmicro® Software 530
PING 530
ROUTER 531
SCRATCHP 531
TELNET 532
WEBROM 532
WEBSERVE 533
WEB_EGI 533

Appendix D PICmicro®-Specific Issues 535
Compiler Support 535

Function Index 541

Stucture Index 545

Index 547

What's on the CD-ROM? 576

2

Chapter 2

Introduction to Protocols: SCRATCHP

Overview

In this chapter, I start by looking at what a protocol is, then I show how it can be implemented in software. I'll examine

- the definition of a protocol,
- the standard way of describing a protocol,
- the client–server model,
- modal and modeless clients, and
- logical connections — open, close, and data transfer

Because this is a hands-on book, I'll illustrate these points by creating a protocol from scratch (called SCRATCHP) and writing a utility that allows you to exercise the protocol. While implementing the protocol, you'll have an opportunity to explore the following areas.

- Storage of Ethernet and SLIP frames
- Ethernet addressing
- Protocol identification
- Byte swapping
- Low-level packet transmission and reception

You'll end up with a stand-alone utility that can be used to exercise the protocol that's been created, or it can be used as a base for implementing another protocol. The foundations will have been laid for the TCP/IP protocols to come.

Protocol

For two computers to communicate, they must speak the same language. A communication language framework is generally called a *protocol*. The name is derived from the framework employed by diplomats when attempting to communicate across cultural boundaries. Two computers may employ different processors, languages, and operating systems, but if they both use a common protocol, then they can communicate.

Protocols don't just enable communications, they also restrict them. Neither party may stray outside the bounds of protocol without facing incomprehension or rejection. So a protocol doesn't just define how communication may occur but also provides a framework for the information that is communicated. But how can any one protocol encompass all the variety of present-day computer communications? It can't, so you need a family of protocols, each of which is designed for a specific task. As with a software project, you need a tree structure, with the simpler network-oriented tasks at the bottom and the higher user-oriented tasks at the top. Such a structure is often called a protocol *stack*, though this leads to confusion with the last in, first out (LIFO, push/pop, or call/return) stack data storage mechanism also used by programmers.

Here, the term stack refers to the way protocol components are stacked on top of each other to give the desired functionality. If I want to transfer a file, I might take a standard file transfer protocol and stack it on top of a communications protocol. The communications protocol wouldn't understand about files — it simply moves bocks of data around. Conversely, the file transfer protocol wouldn't understand about networks — it simply converts files into blocks of data. Combine the two, and you have network file transfer capability.

The separation into protocol layers doesn't necessarily make for easier reading. Older protocol specifications used to simply record the pattern of "bytes on the wire" for achieving a given result (and also, if you're lucky, a smattering of timing information). In a layered world, a protocol specification must tie down the upper and lower application programming interfaces (APIs) and the operations to be performed on them. Any relation to bytes on the wire (i.e., actual visible work) is purely coincidental. There is the danger that the APIs may become operating system–specific, so vendor-independent standardization is very important.

Standardization

The international community, in its wisdom, decided to standardize on the number of protocol layers in a stack, and the International Standards Organization (ISO) Open Systems Interconnection (OSI) model was born (Figure 2.1). Their layers are listed below from top down.

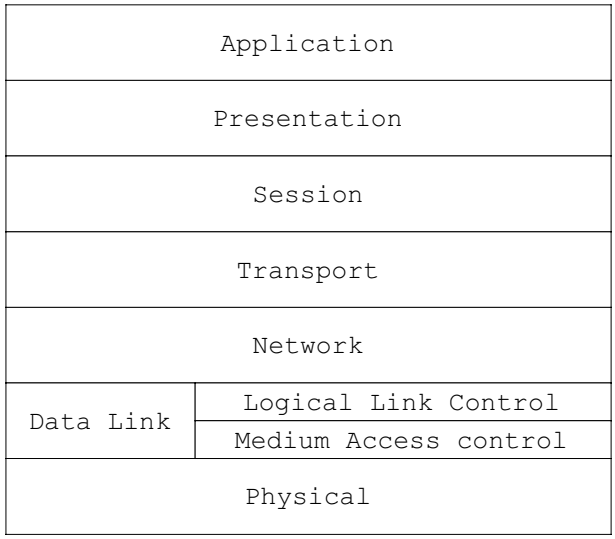
7. ApplicationUser interface
6. PresentationData formatting
5. SessionLogical connections
4. TransportError-free communication
3. NetworkNetwork addressing

2. Data linkTransmission and reception

1. PhysicalNetwork hardware

For local area networks (LANs), the data link layer is further subdivided into a component called medium access control (MAC), which resides within the network hardware, and the software-based logical link control (LLC), which provides a uniform software interface (packet driver interface) to the higher levels.

Figure 2.1 OSI seven-layer model.

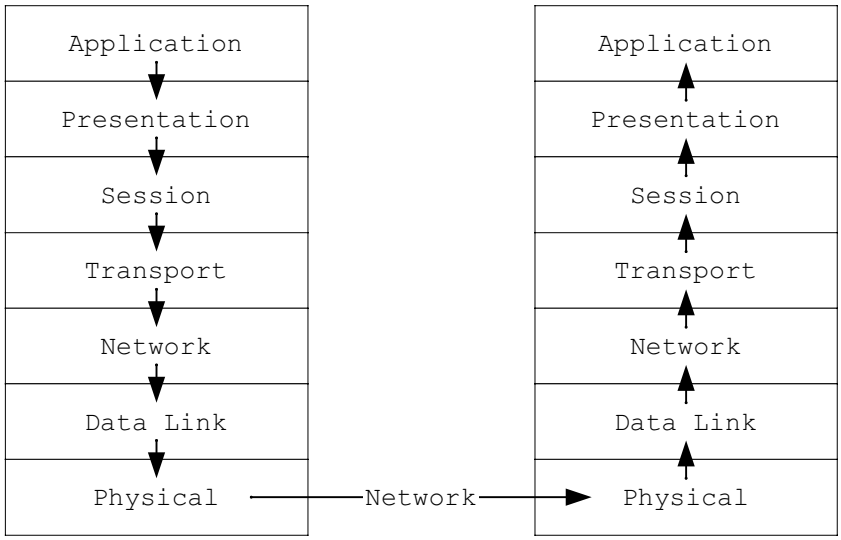


When two applications are communicating over a network, it can be useful to think in terms of the data entering at the top of one protocol stack then traveling downward on that machine, across to the other machine at the physical layer, and back up the second stack (Figure 2.2).

Of course, not all data will originate at application level: resolving addresses requires communication between network layers, and maintenance of a connection requires session-to-session communications. The user is generally unaware of these until he or she happens to see a diagnostic log of all packet transfers; then the reaction is one of amazement that a simple transfer between applications can generate so much traffic. As with a duck crossing a pond, the smooth visible motion belies the furious paddling underneath.

The TCP/IP family of protocols predates the ISO standardization effort, so it does not fit comfortably within the model. Also, the higher layers are remarkably difficult to standardize because they must encompass the totality of network applications. Confronted by the remarkable growth of the Internet, this overall ISO standardization effort has been completely sidelined, though the seven-layer terminology and the lower level standards are still in widespread use. Although I'll implement SCRATCHP as a single protocol, the seven-layer model does provide important pointers on how your software might be structured.

Figure 2.2 Application-to-application transfer.



SCRATCHP Services

Just as an operating system offers the user a range of commands, so SCRATCHP will offer the network user a range of services (i.e., remotely accessible functions). The usual TCP/IP approach is to create a separate specification for each service, but to save time, I'll combine several services into the one protocol. I'll start with a minimum of these, but the protocol must permit the addition of services at a later date. A preliminary list of services is

1. `IDENT` (ID resolution)
2. `ECHO` (connection diagnostic)
3. `DIR` (file directory)
4. `GET` (file transfer: read)
5. `PUT` (file transfer: write)

The `ident` service is used for converting computer IDs into addresses and is explained later. The `echo` service allows simple diagnostic tests to be performed. It duplicates incoming data and returns it to the sender. In this way, you can check response times and error rates. File transfer is a fundamental requirement of computer networking and is useful for examining bulk data transfer techniques. I have provided simple `dir`, `get`, and `put` functions.

Client–Server Model

A useful piece of terminology would be to refer to one machine (the requester of the service) as a *client* and the other (the provider of the service) as the *server*. In reality, you might as well write the software so that every machine has the potential to become a client or a server and use keyboard or network commands to determine which mode should be activate at any time.

The `ident` service is used to identify potential servers, so it must be as simple as possible — one packet transmitted and one packet received. The command is sent as a string, followed by an optional argument string. If a server responds, it returns a copy of the command string to confirm which command it is responding to.

A potential problem is that the response is indistinguishable from the command, so it may be interpreted as another command, generating another response, and so on ad infinitum. There are two approaches to solving this problem: one modal, the other modeless.

Modal Client

Every time a client issues a command, it could go into some sort of command mode, so it knows the next communication it receives is going to be a response to that command. This mode would typically be stored as a state variable. The transaction would be:

1. client goes into command mode and
2. client sends command to server; then,

either

3. client receives response and goes back into normal mode

or

3. client receives no response, times out, and goes back into normal mode.

There are two risks with this approach.

1. The client time-out occurs while the response is still in transit, so it is no longer in command mode when it arrives.
2. While in command mode, the client receives an unexpected packet from another node, which it can't handle because it is in the wrong mode.

Modal techniques are frequently used in simple point-to-point serial links, but they must be used with care in networking, where it is impossible to anticipate what will happen next.

Modeless Client

If you want to keep your client as modeless (i.e., stateless) as possible, you must include more information in the data packet that is transmitted. Instead of sending pure data and storing the command mode *internally*, the transmitted packet must contain an indication that says, "I am a command packet." The server's reply packet must then have a different indication that says, "I am a response packet." By expressing the information *externally*, the client doesn't have to store it internally, and debugging is made easier because you can determine the client's intentions by examining the packets it has sent, rather than having to pry on its internal data.

It is interesting to note that one of the key factors in the success of the World Wide Web has been that the upper protocol layers are stateless. At any one time a Web server may be handling hundreds of clients, and in the course of a day it may handle millions. If it had to keep detailed information on each, there would be a major storage problem. A simple Web server stores no information about any user. Contrast this with a typical multiuser system, where a large number of settings and preferences are stored in an individual user's account.

So, keep the `ident` command stateless for simplicity, but what about the file transfer commands? If you're going to handle bulk data transfers, it is hard to keep the machines completely stateless. If nothing else, they have to remember which files they have opened and

why. Ideally, the network would be treated as a simple *pipe*, through which data would flow (or stream).

For this, you really need to establish a *logical connection*, or bidirectional *data pipe* between the client and server: anything fed in one end of the pipe will emerge unaltered at the other end. This is an important concept, which is much used in networking.

Logical Connections

From the earliest days, networks have usually been used for the purpose of establishing logical connections between two computers. When you use a browser to contact a Web site, you are setting up one or more logical connections between your client and their server. The Web pages and graphics are then fed down these connections, like water down a pipe, until the client has all the necessary data to display the page.

Logical connections are reliable. To maintain the connection, the protocol software has to keep track of all packets sent and received and have a retry strategy to cover any packets that go astray. Unfortunately, this reliability comes at a price: writing the protocol software for opening, maintaining, and closing logical connections is a nontrivial task, involving the creation of state machines in both client and server and an exchange of signals between them to ensure the state machines remain in sync.

You may spot an apparent contradiction with my previous assertion that Web client–server communications are stateless. Clearly they must keep state information about each other for the duration of a transfer. That's why I was careful to say their *applications* are stateless; the lower levels are continuously making and breaking connections, with all the state tracking that entails.

Opening and Closing a Connection

In a simplified protocol, clients have to initiate all actions, so they will request a service that requires the establishment of a connection. The host can then agree to the establishment of a connection by acknowledging the request, ignoring the request if it disagrees, or setting an error flag (it may have insufficient resources to support another connection).

Closure of the connection may be initiated by either party. In a file transfer, it will normally be the sender of the data who closes the connection after the data is transferred; although, the recipient may also do this if it can't handle the data any more (e.g., its disk is full).

Data Flow in a Connection

For the duration of the connection, data may flow bidirectionally between the two parties. Both sides need to keep track of the amount of data sent and received to ensure no data has been skipped or duplicated. Commonly used techniques to do this are listed below.

Lock-step One packet is sent, and the sender waits until an acknowledgment is received before sending another.

Block sequencing Each packet contains one data block, with a sequential number (sequence number). The recipient may acknowledge receipt of each block, using its sequence

number, or wait until a few have been received then acknowledge them all, using the sequence number of the latest block.

Byte sequencing This is similar to block sequencing, but the sequence number reflects the byte count, rather than the block count.

Figure 2.3 Sequencing methods.

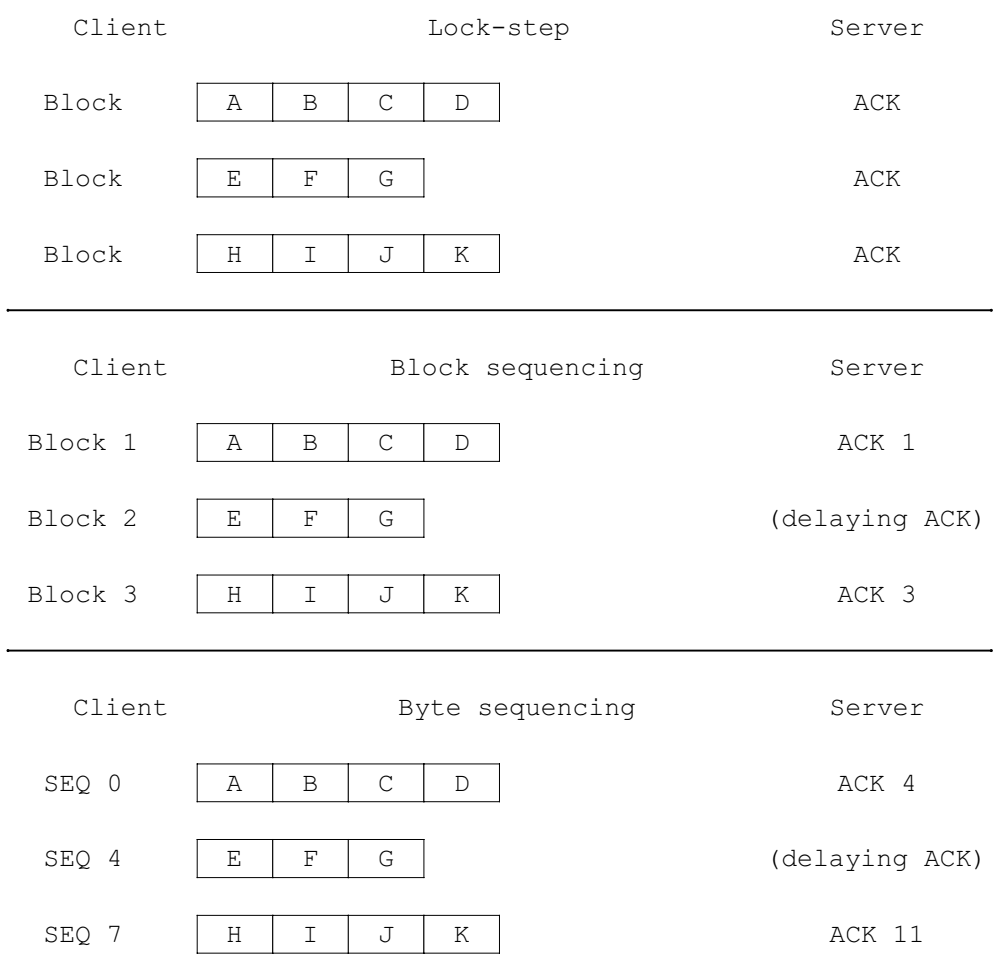


Figure 2.3 shows the client–server interactions, assuming the client is sending 11 bytes in three blocks to the server. The lock-step method doesn't need to identify each block individually, since only one can be in transit at any one time (in railway parlance: one engine in steam). The sequencing methods differ in that they identify a block using either an incrementing block number or the total number of bytes sent prior to the current block. The acknowledgment reflects either the latest block received or the latest byte received (i.e., the sequence number plus the byte count).

For simplicity, I have shown the first block as having a byte count of zero. In reality, it is better to start with a pseudorandom base value, which is negotiated at the start of the transaction and is subsequently increased to reflect the actual byte count transferred. The value is typically stored as a 32-bit `LWORD` and is allowed to wrap around past zero when it gets too large, on the assumption that there won't be several gigabytes of data in transit for any one transaction at any one time.

Note that the lock-step method has built-in flow control: the sender cannot out-pace the receiver because the receiver will only acknowledge if it has spare buffer space for the next data block. Flow control can be added to the other techniques by the simple expedient of placing a limit on the maximum number of blocks (or bytes of data) that can be in transit and unacknowledged. When the sender exceeds this "window," it must stop transmitting data until it receives an acknowledgment.

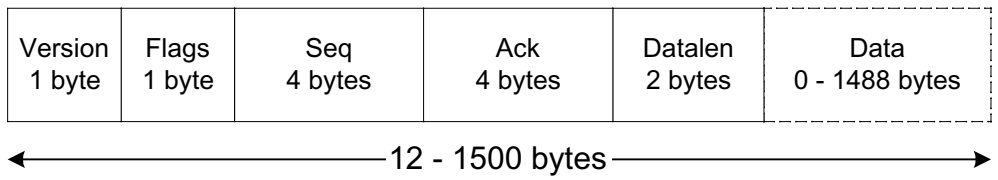
There is little to choose between the two sequencing methods. Block acknowledgment is used in the ISO link layer LLC and is slightly easier to implement than byte sequencing, provided a fixed block size is used. TCP has a variable block size (a "sliding window"), so it employs a byte-sequencing method. This is what I'll use for SCRATCHP.

Packet Format

Having decided on the basic structure of transactions, I can define a packet format to suit (Figure 2.4). Because of my minimalist approach, there is relatively little in it.

- protocol version (one byte)
- flags (one bit each)
  - command
  - response
  - start connection
  - connected
  - stop connection
  - error
- sequence and acknowledgment numbers (four bytes each)
- data length (two bytes)

Figure 2.4 SCRATCHP packet format.



The *protocol version* is a useful way of retaining compatibility as SCRATCHP evolves. It can be checked by any recipient to ensure that it is equipped to decode this version of the protocol and to give the user sensible error messages if there is a problem (e.g., "This utility does not support SCRATCHP version 5").

The *data length* field may seem redundant, since the underlying network protocol should provide an overall length value from which the data length could be derived. Unfortunately, the Ethernet frame length will include any padding applied to undersized frames, so it won't always give the correct answer.

```
/* Flag values */
#define FLAG_CMD        0x01    /* Data area contains command */
#define FLAG_RESP       0x02    /* Data area contains response */
#define FLAG_START      0x04    /* Request to start connection */
#define FLAG_CONN       0x08    /* Connected; sequenced transfer in use */
#define FLAG_STOP       0x10    /* Stop connection */
#define FLAG_ERR        0x20    /* Error; abandon connection */

/* SCRATCHP packet header */
typedef struct {
    BYTE ver;                   /* Protocol version number */
    BYTE flags;                 /* Flag bits */
    LWORD seq;                  /* Sequence value */
    LWORD ack;                  /* acknowledgment value */
    WORD dlen;                  /* Length of following data */
} SCRATCHPHDR;

/* SCRATCHP packet */
#define SCRATCHPDLEN 994
typedef struct {
    SCRATCHPHDR h;              /* Header */
    BYTE data[SCRATCHPDLEN];    /* Data (or null-terminated cmd/resp string) */
} SCRATCHPKT;
```

In common with most other Ethernet protocols, the integer values will be sent with the most significant byte first. The SCRATCHP data array is dimensioned at 994 bytes, which allows it to fit within the 1,500-byte Ethernet or 1,006-byte SLIP data area.

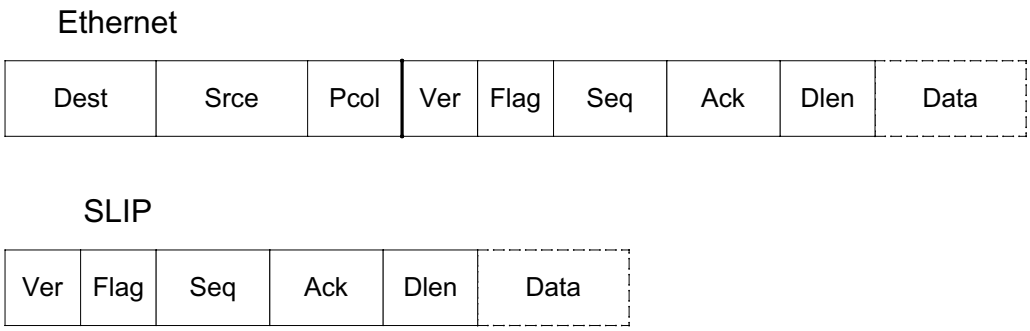
Internal Storage

Having fixed the external appearance of the SCRATCHP packet, I need to decide the internal storage format. You will recall that my network drivers work on a generic frame format, which has a two-byte frame type (which will identify whether it is an Ethernet or SLIP packet and maybe provide a system-specific handle for the network adaptor), followed by a block of data up to the maximum Ethernet frame size.

```
typedef struct {
    GENHDR g;                   /* General-pupose frame header */
    BYTE buff[MAXGEN];          /* Frame itself (2 frames if fragmented) */
} GENFRAME;
```

The SCRATCHP packet will be contained within the data area of an Ethernet or SLIP packet (Figure 2.5).

Figure 2.5 Ethernet and SLIP packets.



Because Ethernet and SLIP packets have different header lengths (14 bytes and zero bytes), you need a standard way of determining where the network header ends and the SCRATCHP packet starts. A function can do this by checking the packet type and indexing into the packet data area accordingly.

```
/* Get pointer to the data area of the given frame */
void *getframe_datap(GENFRAME *gfp)
{
    return(&gfp->buff[dtype_hdrlen(gfp->g.dtype)]);
}
/* Return frame header length, given driver type */
WORD dtype_hdrlen(WORD dtype)
{
    return(dtype&DTYPE_ETHER ? sizeof(ETHERHDR) : 0);
}
```

Note that a pointer to the frame *data* area also points to the SCRATCHP *header*, and a pointer to the SCRATCHP *data* area may also point to a command *header*. In this nested world, one packet's data is generally another packet's header, so the term "data" must always be qualified by the context in which it appears.

There are other awkward differences between Ethernet and SLIP: the former has a source address, which will be useful when sending a reply, and a protocol-type identifier, which is discussed later. Any functions attempting to access these features need to check the packet type first.

```
/* Get pointer to the source address of the given frame, 0 if none */
BYTE *getframe_srcep(GENFRAME *gfp)
{
    ETHERHDR *ehp;
    BYTE *srce=0;

    if (gfp->g.dtype & DTYPE_ETHER)         /* Only Ethernet has address */
    {
        ehp = (ETHERHDR *)gfp->buff;
        srce = ehp->srce;
    }
    return(srce);
}
/* Copy the source MAC addr of the given frame; use broadcast if no addr */
BYTE *getframe_srce(GENFRAME *gfp, BYTE *buff)
{
    BYTE *p;

    p = getframe_srcep(gfp);
    if (p)
        memcpy(buff, p, MACLEN);
    else
        memcpy(buff, bcast, MACLEN);
    return(p);
}
/* Get pointer to the destination address of the given frame, 0 if none */
BYTE *getframe_destp(GENFRAME *gfp)
{
    ETHERHDR *ehp;
    BYTE *dest=0;

    if (gfp->g.dtype & DTYPE_ETHER)         /* Only Ethernet has address */
    {
        ehp = (ETHERHDR *)gfp->buff;
        dest = ehp->dest;
    }
    return(dest);
}
```

```
/* Copy destination MAC addr of the given frame; use broadcast if no addr */
BYTE *getframe_dest(GENFRAME *gfp, BYTE *buff)
{
    BYTE *p;

    p = getframe_destp(gfp);
    if (p)
        memcpy(buff, p, MACLEN);
    else
        memcpy(buff, bcast, MACLEN);
    return(p);
}
/* Get the protocol for the given frame; if unknown , return 0 */
WORD getframe_pcol(GENFRAME *gfp)
{
    ETHERHDR *ehp;
    WORD pcol=0;

    if (gfp->g.dtype & DTYPE_ETHER)         /* Only Ethernet has protocol */
    {
        ehp = (ETHERHDR *)gfp->buff;
        pcol = ehp->ptype;
    }
    return(pcol);
}
```

Using these functions, you can safely access the address and protocol fields on all packets, even though SLIP frames don't possess them. This avoids the necessity for frame-specific features in the SCRATCHP code layer, since all frames can be treated equally.

Addressing

I have already talked about the client contacting the server, but I have given no indication as to how this is achieved. How are the client and server identified so that they can contact each other? Of course, the server can simply respond to the address of any client that contacts it, but there is still the burden on the client to make the initial contact, and to do that, it needs some way of addressing the host, since there might be multiple hosts on the network.

Each Ethernet card has a unique six-byte *physical address*, so the client could use that. But imagine the complaints from the users if they have to type a 12-digit hexadecimal number every time they want to contact a new host. Also, the number would be highly specific to that item of hardware. If the network card failed and had to be replaced, the number would change, even though the computer still seemed to be the same from the user's point of view.

It is far better to assign each computer on the network a *logical address* then invent some scheme to map the logical address onto the physical address of the Ethernet card. For convenience, I will refer to the logical address as the Ident (ID) of the computer and the physical address as the *address*. The logical-to-physical mapping process is called *address resolution*.

What is an ID, and where does it come from? An ID can be numeric (`123`) or a null-terminated string (`fileserver`). I'll use the latter format for maximum flexibility. It must either be permanently burned into the software when it is created (a nuisance, since all nodes on the network would have to run different copies of the software) or read when the software is loaded — either from the command line or from a configuration file. Either way, it is essential that each computer on the network acquires a unique ID.

To resolve an ID into an address, the client must broadcast the ID on the network as an invitation for the designated server to respond. The server responds, giving its physical address, which the client stores and uses for all subsequent communications.

Figure 2.6 Sample `ident` transactions.

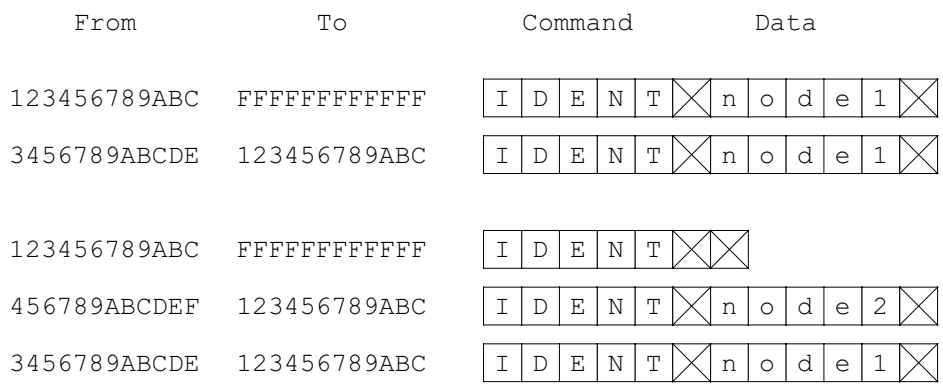


Figure 2.6 shows a client broadcasting an identification request for the machine `node1`, using two null-terminated strings in the SCRATCHP data area — the null character is indicated by a strikethrough of the box. The client receives a reply containing a duplicate of the request, with the all-important node address, which will be used for subsequent communications. The second transaction illustrates the use of a null `ident` string to identify all nodes on the (hopefully very small) network. Two responses are obtained in a pseudorandom order. There is no knowing which node will answer first.

Protocol Identification

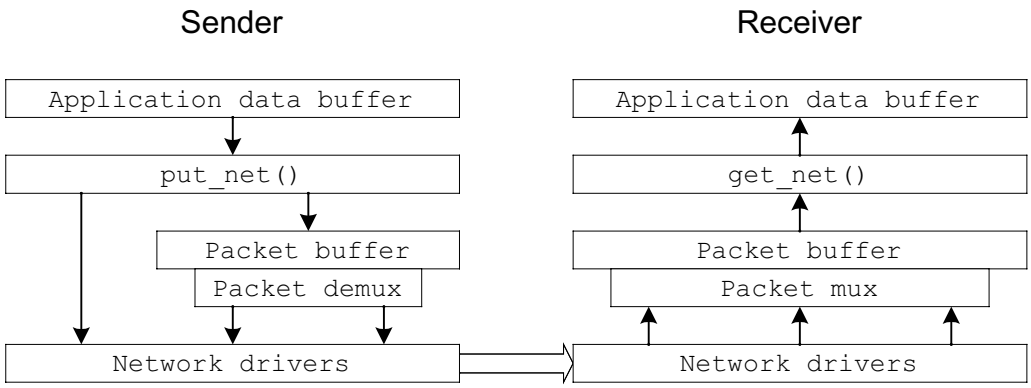
Ethernet is capable of carrying several protocols at the same time without the risk of confusion over which data belongs to which protocol. It achieves that by tagging each frame with a 16-bit protocol type, which uniquely identifies that protocol; for example, Internet Protocol (IP) has a hexadecimal value of `800h`. If SCRATCHP was intended to coexist with other protocols, you would need to obtain an official protocol identifier from the Institution of Electrical and Electronic Engineers (IEEE). At the time of writing, this cost $5,000; however, SCRATCHP should only be run on a "scratch" network, so you can use any identifier you

like. Prudence dictates you should pick a high number that is out of the range of currently assigned protocols, so the hexadecimal value FEEBh is used.

Multiplexing and Buffering

The software that gathers transmit packets from a variety of senders is a *multiplexer* (mux, for short), and the corresponding software that accepts received packets and dispatches them to the appropriate recipient is called a *demultiplexer* (demux, for short).

Figure 2.7 Data flow between nodes.



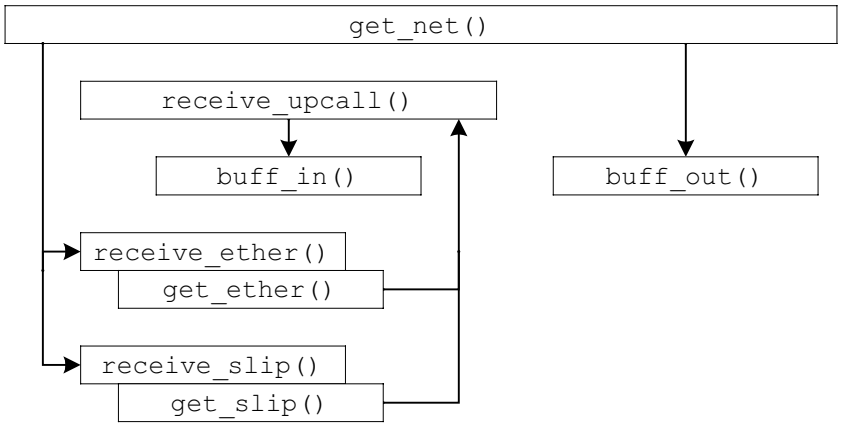
The mux/demux operation (Figure 2.7) is automatically performed by the network driver layer. Submitting a packet to put_net() automatically routes it to the appropriate network driver, possibly via a (polygonal, as described in the previous chapter) packet buffer, if the interface doesn't have its own Transmit buffer. All received packets are stored in a similar polygonal incoming packet buffer.

Control flow, as shown in Figure 2.8, is more convoluted since there must be some provision for polling the network interfaces, as they may be interrupt-driven.

The receive_ether() and receive_slip() functions take the place of Ethernet and serial interrupt handlers, in that they are called from get_net(), call get_ether() or get_slip() for each packet received, then do an up-call to save the packet, which in turn uses the standard circular buffer input routine (Figure 2.8). Having done that, get_net() calls the buffer output routine to fetch any stored packets.

If interrupts are available, the two receive_ functions are redundant, and the interrupt handlers call the get_ functions directly.

Figure 2.8 Control flow for packet reception.



Byte Swapping

SCRATCHP will normally run on a little endian (least significant byte first) PC architecture, so it was tempting to use this storage method for the two-byte values in the SCRATCHP packets. However, most Ethernet protocols use big endian (most significant byte first) storage, and I wanted to explore byte-swapping issues, so I made SCRATCHP little endian. If you happen to run my software on a little endian machine, then the byte-swapping stage must be skipped (preferably using conditional compilation), but the underlying software structure remains the same.

I have seen protocol software that is liberally sprinkled with byte swap functions, which is a nightmare to debug because you're never quite sure whether a value is in its swapped or unswapped state. To avoid this, you have to have a byte-swapping philosophy and stick rigidly to it. My philosophy is that byte swapping is the *last* action to be performed when *sending* a packet and the *first* action to be performed when *receiving* a packet.

This means that a transmit packet, that has been byte swapped is only fit for transmission: it may not be used for other purposes such as diagnostic printouts because the printout function won't display the swapped values correctly. After transmission, a transmit packet must be discarded because it is useless; on the relatively rare occasions a retransmission is required, the packet can easily be rebuilt from the original data. This approach also helps to minimize the storage requirements and forces you to think clearly about a retry strategy, rather than relying on resending old packets that happen to be around. This rigorous approach is perhaps slightly too dogmatic and inflexible for a simple protocol such as SCRATCHP, but it prepares the ground for the more complex protocols to come.

Reception and Transmission

When a packet is received, do the necessary testing and byte swapping then forward it to `do_scratchp()` for action.

```
/* Demultiplex incoming packets */
int get_pkts(GENFRAME *nfp)
{
    int rxlen, txlen=0;

    if ((rxlen=get_frame(nfp)) > 0)         /* If any packet received.. */
    {
        if (is_scratchp(nfp, rxlen))        /* If SCRATCHP.. */
        {
            swap_scratchp(nfp);                 /* ..do byte-swaps.. */
            txlen = do_scratchp(nfp, rxlen, 0); /* ..action it.. */
        }
    }                                       /* ..and maybe return a response */
    return(txlen);                          /* (using the same pkt buffer) */
}
```

To economize on storage, `do_scratchp()` reuses the Receive buffer as a Transmit buffer to hold any response it wants to make and simply returns a transmit length value, or `0` if no response has been generated.

```
/* Check Ethernet frame, given frame pointer & length, return non-0 if OK */
int is_ether(GENFRAME *gfp, int len)
{
    int dlen=0;

    if (gfp && (gfp->g.dtype & DTYPE_ETHER) && len>=sizeof(ETHERHDR))
    {
        dlen = len - sizeof(ETHERHDR);
        swap_ether(gfp);
    }
    return(dlen);
}
/* Make a frame, given data length. Return length of complete frame
** If Ethernet, set dest addr & protocol type; if SLIP, ignore these */
int make_frame(GENFRAME *gfp, BYTE dest[], WORD pcol, WORD dlen)
{
    ETHERHDR *ehp;
```

```
    if (gfp->g.dtype & DTYPE_ETHER)
    {
        ehp = (ETHERHDR *)gfp->buff;
        ehp->ptype = pcol;
        memcpy(ehp->dest, dest, MACLEN);
        swap_ether(gfp);
        dlen += sizeof(ETHERHDR);
    }
    return(dlen);
}
/* Byte-swap an Ethernet frame, return header length */
void swap_ether(GENFRAME *gfp)
{
    ETHERFRAME *efp;

    efp = (ETHERFRAME *)gfp->buff;
    efp->h.ptype = swapw(efp->h.ptype);
}
/* Check SLIP frame, return non-zero if OK */
int is_slip(GENFRAME *gfp, int len)
{
    return((gfp->g.dtype & DTYPE_SLIP) && len>0);
}
/* Check for SCRATCHP, given frame pointer & length */
int is_scratchp(GENFRAME *nfp, int len)
{
    WORD pcol;
                                            /* SLIP has no protocol field.. */
    pcol = getframe_pcol(nfp);             /* ..so assume 0 value is correct */
    return((pcol==0 || pcol==PCOL_SCRATCHP) && len>=sizeof(SCRATCHPHDR));
}

/* Byte-swap an SCRATCHP packet, return header length */
int swap_scratchp(GENFRAME *nfp)
{
    SCRATCHPKT *sp;

    sp = getframe_datap(nfp);
    sp->h.dlen = swapw(sp->h.dlen);
```

```
    sp->h.seq = swapl(sp->h.seq);
    sp->h.ack = swapl(sp->h.ack);
    return(sizeof(SCRATCHPHDR));
}
```

Transmission is a fill-in-the-blanks exercise, followed by the necessary byte swaps.

```
/* Make a SCRATCHP packet given command, flags and string data */
int make_scratchpds(GENFRAME *nfp, BYTE *dest, char *cmd,
                    BYTE flags, char *str)
{
    return(make_scratchp(nfp, dest, cmd, flags, str, strlen(str)+1));
}

/* Make a SCRATCHP packet given command, flags and data */
int make_scratchp(GENFRAME *nfp, BYTE *dest, char *cmd, BYTE flags,
                  void *data, int dlen)
{
    SCRATCHPKT *sp;
    ETHERHDR *ehp;
    int cmdlen=0;

    sp = (SCRATCHPKT *)getframe_datap(&genframe);
    sp->h.ver = SCRATCHPVER;                /* Fill in the blanks.. */
    sp->h.flags = flags;
    sp->h.seq = txbuff.trial;               /* Direct seq/ack mapping.. */
    sp->h.ack = rxbuff.in;                 /* ..to my circ buffer pointers! */
    if (cmd)
    {
        strcpy((char *)sp->data, cmd);      /* Copy command string */
        cmdlen = strlen(cmd) + 1;
    }
    sp->h.dlen = cmdlen + dlen;             /* Add command to data length */
    if (dlen && data)                       /* Copy data */
        memcpy(&sp->data[cmdlen], data, dlen);
    if (nfp->g.dtype & DTYPE_ETHER)
    {
        ehp = (ETHERHDR *)nfp->buff;
        ehp->ptype = PCOL_SCRATCHP;         /* Fill in more blanks */
        memcpy(ehp->dest, dest, MACLEN);
    }
```

```
    diaghdrs[diagidx] = sp->h;              /* Copy hdr into diagnostic log */
    diaghdrs[diagidx].ver = DIAG_TX;
    diagidx = (diagidx + 1) % NDIAGS;
    return(sp->h.dlen+sizeof(SCRATCHPHDR)); /* Return length incl header */
}

/* Transmit a SCRATCHP packet. given length incl. SCRATCHP header */
int put_scratchp(GENFRAME *nfp, WORD txlen)
{
    int len=0;

    if (txlen >= sizeof(SCRATCHPHDR))       /* Check for min length */
    {
        if (pktdebug)
        {
            printf ("Tx ");
            disp_scratchp(nfp);
            printf("   ");
        }
        swap_scratchp(nfp);                 /* Byte-swap SCRATCHP header */
        if (is_ether(nfp, txlen+sizeof(ETHERHDR)))
            txlen += sizeof(ETHERHDR);
        txcount++;
        len = put_net(nfp, txlen);          /* Transmit packet */
    }
    return(len);
}
```

Implementation

If you have read the first chapter, you'll not be surprised that I'm about to embark on a states-and-signals exercise. The software receives the following signals.

- User (keystrokes)
- Network (packets)
- Timer (time-outs)
- Null (idle)

When it receives one of these, it may take any or none of the following actions.

- Change state
- Send a packet
- Update user display

I'll start with the simplest command, `ident`, which is completely stateless.

`ident` Command

When the user presses the `I` key, a broadcast Ident packet is emitted. If any responses are received, the software displays them as part of its normal idle-state network polling.

First, I have a main loop that translates the key press into a signal.

```
GENFRAME *nfp;
WORD txlen;
...
nfp = &genframe;                              /* Open net driver.. */
nfp->ftype = frametype = open_net(netcfg);  /* ..get frame type */
...
int i, keysig, connsig, sstep=0;

while (cmdkey != 'Q')               /* Main command loop.. */
{
    txlen = keysig = connsig = 0;
    if (sstep || kbhit())           /* If single-step or keypress..*/
    {
        k = getch();                /* ..get key */
        if (sstep)
            timeout(&errtimer, 0);  /* If single-step, refresh timer */
        cmdkey = toupper(k);        /* Decode keystrokes.. */
        switch (cmdkey)             /* ..and generate signals */
        {
        case 'I':                   /* 'I': broadcast ident */
            if (connstate != STATE_CONNECTED)
            printf("Broadcast ident request\n");
            keysig = SIG_USER_IDENT;
            break;
        }
    }
}
connsig = do_apps(&rxbuff, &txbuff, keysig);
txlen = do_scratchp(nfp, 0, connsig);
put_scratchp(nfp, txlen);       /* Transmit packet (if any) */
```

The user key press is translated into a key signal, `SIG_USER_IDENT`. This signal is bounced straight through the application code, `do_apps()`, without change (more on this function later). It is then sent to the main SCRATCHP state machine, `do_scratchp()`, to be translated into a network packet.

```
int do_scratchp(GENFRAME *nfp, int rxlen, int sig)
{
    ...
    if (connstate == STATE_IDLE)           /* If idle state.. */
    {
        timeout(&errtimer, 0);             /* Refresh timer */
        switch (sig)                       /* Check signals */
        {
        case SIG_USER_IDENT:                /* User IDENT request? */
            txlen = make_scratchpds(nfp, bcast, CMD_IDENT, FLAG_CMD, "");
            break;
        ...
        }
    }
    ...
}
```

The packet (in the buffer indicated by network frame pointer `nfp`) is then transmitted by `put_scratchp()`.

```
txlen = make_scratchpds(nfp, bcast, CMD_IDENT, FLAG_CMD, "");
put_scratchp(nfp, txlen);
...
```

So what happens when you press the `I` key? With a bit of luck, your first packet is sent on the network. If you're fortunate enough to possess a protocol analyzer (which captures and displays all network traffic), you might see a display similar to this.

```
Packet #1
  Packet Length:64
  Ethernet Header
  Destination:  FF:FF:FF:FF:FF:FF  Ethernet Broadcast
  Source:       00:C0:26:B0:0A:93  Rack2
  Protocol Type:0xFEEB
  Packet Data:
............iden  01 01 00 00 00 00 00 00 00 00 00 07 69 64 65 6E
t...............  74 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
..............    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This is the actual byte stream on the network. The analyzer doesn't understand the packet contents, so some manual decoding is necessary. The six-byte Ethernet addresses are unique to each adaptor, so yours should not be the same as mine! The analyzer has identified the node name as `Rack2`, which, not coincidentally, is the same ID name as in the SCRATCHP configuration file.

The actual data is below the 64-byte minimum frame size, so there is a significant amount of padding. You can see the protocol version number (`01`) followed by the command flag (`01`). Skipping the four-byte sequence and acknowledgment numbers, there is a length value of seven

(most significant byte first). The `ident` string is only six bytes, including a null terminator, so one extra null character is significant, indicating that this is a wildcard search for all nodes.

Such a broadcast would be inadvisable on a network of any size, since I'd get a flood of responses, but I'll assume I have only two other nodes on the network, named `vale` and `sun`, to get the responses.

```
Packet #2
  Packet Length:64
  Ethernet Header
  Destination:  00:C0:26:B0:0A:93  Rack2
  Source:       00:20:18:3A:ED:64  Sun  Protocol Type:0xFEEB
  Packet Data:
............iden  01 02 00 00 00 00 00 00 00 00 0A 69 64 65 6E
t.sun...........  74 00 73 75 6E 00 00 00 00 00 00 00 00 00 00
..............    00 00 00 00 00 00 00 00 00 00 00 00 00 00

Packet #3
  Packet Length:64
  Ethernet Header
  Destination:  00:C0:26:B0:0A:93  Rack2
  Source:       00:50:04:F7:7C:CA  Vale
  Protocol Type:0xFEEB
  Packet Data:
............iden  01 02 00 00 00 00 00 00 00 00 0B 69 64 65 6E
t.vale..........  74 00 76 61 6C 65 00 00 00 00 00 00 00 00 00
..............    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The order in which these responses arrive is not significant, since both are transmitting at more or less the same time.

The responses are received and decoded and displayed by the SCRATCHP application.

```
ident 'sun' address 00:20:18:3a:ed:64
ident 'vale' address 00:50:04:f7:7c:ca
```

In a real application, the Ident-to-address mapping would be stored (cached) for reuse later. I will just display the addresses and discard them.

```
int do_scratchp(GENFRAME *nfp, int rxlen, int sig)
{
    if (rxlen)                              /* If packet received.. */
    {
        rxflags = sp->h.flags;              /* Decode command & data areas */
        if (rxflags&FLAG_CMD || rxflags&FLAG_RESP)
            crlen = strlen((char *)sp->data) + 1;
        dlen = sp->h.dlen - crlen;          /* Actual data is after command */
        if (rxflags & FLAG_ERR)             /* Convert flags into signals */
            sig = SIG_ERR;
        ...
        else if (rxflags & FLAG_CMD)
```

```
            sig = SIG_CMD;
        else if (rxflags & FLAG_RESP)
            sig = SIG_RESP;
        ...
    }
    if (connstate == STATE_IDLE)           /* If idle state.. */
    {
        timeout(&errtimer, 0);             /* Refresh timer */
        switch (sig)                       /* Check signals */
        {
        case SIG_CMD:                      /* Command signal? */
            if (!strcmp((char *)sp->data, CMD_IDENT))
            {                              /* IDENT cmd with my ID or null? */
                if (dlen<2 || !strncmp((char *)&sp->data[crlen], locid, dlen))
                {                          /* Respond to sender */
                    txlen = make_scratchp(nfp, getframe_srcep(nfp), CMD_IDENT,
                                          FLAG_RESP, locid, strlen(locid)+1);
                }
            }
            break;

        case SIG_RESP:                     /* Response signal? */
            if (!strcmp((char *)sp->data, CMD_IDENT))
            {                              /* IDENT response? */
                printf("Ident '%s'", (char *)&sp->data[crlen]);
                if ((p=getframe_srcep(nfp)) !=0 )
                {
                    printf(" address ");
                    pr6byt(p);
                }
                printf("\n");
            }
            break;
        ...
        }
    }
}
```

If a packet is received (`rxlen` is non-zero), then a signal is raised. Because I am the command originator, I'm interested in the response signal, `SIG_RESP`, which simply prints the `ident` name and address.

Note that the same function also handles the case where I have *received* a command; that is, I am the host being queried. In this case, a `SIG_CMD` is raised, and I must respond by putting my (local) `ident` string, `locid`, in the response.

It may seem strange placing the client and server code side-by-side in the same function, and this can make the code slightly more difficult to read, since these are two mutually exclusive execution strands. However, the commonality of the support code (e.g., packet composition and decomposition) and the vital necessity of keeping any modifications to the client and server in sync does favor this approach, even at the expense of some confusion over identity ("… so, is this a client, or server, or what?").

Connection

The bulk of the services require a logical connection between the two machines. I can put off the creation of state and signal tables no longer (Table 2.1).

Table 2.1 State and signal table.

Signals	States				
	IDLE	IDENT	OPEN	CONNECTED	CLOSE
CMD	Send RESP			Send to app.	
RESP	Check RESP	Send START <OPEN>		Send to app.	
START	Send CONN <CONNECTED>		Send CONN <CONNECTED>	Send CONN	
CONN	Send ERR		Send CONN <CONNECTED>	Send to app.	
STOP	Send ERR		Send STOP <IDLE>	Send STOP <IDLE>	<IDLE>
ERR			<IDLE>	Send STOP <IDLE>	Send STOP <IDLE>
timeout		Resend IDENT	Resend START	Resend data	Resend STOP
fail		<IDLE>	Send ERR <IDLE>	Send ERR <IDLE>	<IDLE>
open	Send IDENT <IDENT>				
close			Send END <IDLE>	Send STOP <CLOSE>	

Connection State Machine

The state changes have been marked with angle brackets, so `<IDENT>` indicates a change to the `IDENT` state. Network signals (from received packets) are in uppercase, whereas user and system signals (key presses and time-outs) are in lowercase. The `fail` signal is raised after several successive time-outs (i.e., the retry count has been exceeded).

Opening and Closing a Connection

To assist you in reading these tables, here's a sample connection sequence for a client. That is, the node requests the connection starting from the `IDLE` state.

1. receive `open` signal from user; send `ident` command; go to `IDENT` state
2. receive `ident` response; send `start`; go to `OPEN` state
3. receive `conn`; send `conn`; go to `CONNECTED` state

The client also shoulders the burden of handling connection errors. Each step is retried on time-out.

1. no `ident` response; resend `ident` command
2. no `conn` response; resend `start` command

The sequence for the server, starting from `IDLE`, is much simpler.

1. receive `ident` command; send response; no state change
2. receive `start`; send `conn`; go to `CONNECTED` state

However, you must exercise a small amount of caution when assuming that the client is responsible for all error handling. Imagine that the server's `conn` response is corrupted; the server then thinks it is connected, but the client doesn't realize this, so it resends a `start` signal. Although already connected, the server must accept this error condition (the duplicate `start` packet) and resend the `conn`.

There are two closure sequences: abrupt, in the event of an error, or slightly more graceful under normal conditions.

The graceful closure involves the exchange of `stop` signals, whereas the abrupt closure is the unilateral sending of an error packet. A potential problem with the latter is that the error packet may go astray, then one side would think the connection was still open, while the other thought it was closed. The only remedy for this situation is that, sooner or later, the open side would send a data packet to the closed side and receive an error packet in response, thus closing the connection.

The state machine software is simply a large set of nested conditionals, with entries for each state–signal combination that requires an action.

```
int do_scratchp(GENFRAME *nfp, int rxlen, int sig)
{
    ...
    if (connstate == STATE_IDLE)            /* If idle state.. */
    {
        timeout(&errtimer, 0);              /* Refresh timer */
        switch (sig)                        /* Check signals */
        {
        case SIG_USER_IDENT:                 /* User IDENT request? */
            txlen = make_scratchpds(nfp, bcast, CMD_IDENT, FLAG_CMD, "");
            break;
```

```
        case SIG_USER_OPEN:                 /* User OPEN request? */
            txlen = make_scratchpds(nfp, bcast, CMD_IDENT, FLAG_CMD, remid);
            buff_setall(&txbuff, 1);        /* My distinctive SEQ value */
            newconnstate(STATE_IDENT);      /* Start ident cycle */
            break;

        case SIG_CMD:                       /* Command signal? */
            ...
            break;

        case SIG_RESP:                      /* Response signal? */
            ..,
            break;

        case SIG_START:                     /* START signal? */
            getframe_srce(nfp, remaddr);
            buff_setall(&txbuff, 0x8001);   /* My distinctive SEQ value */
            txack = sp->h.seq;              /* My ack is his SEQ */
            buff_setall(&rxbuff, txack);
            *remid = 0;                     /* Clear remote ID */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_CONN, 0, 0);
            newconnstate(STATE_CONNECTED);  /* Go connected */
            break;

        case SIG_CONN:                      /* CONNECTED or STOP signal? */
        case SIG_STOP:
           txlen = make_scratchp(nfp, getframe_srcep(nfp), 0, FLAG_ERR, 0, 0);
            break;                          /* Send error */
        }
    }
    else if (connstate == STATE_IDENT)      /* If in identification cycle.. */
    {
        switch (sig)                        /* Check signals */
        {
        case SIG_RESP:                      /* Got IDENT response? */
            if (!strcmp((char *)sp->data, CMD_IDENT) && dlen<=IDLEN)
            {
                if (!remid[0] || !strcmp((char *)&sp->data[crlen], remid))
                {                           /* Get remote addr and ID */
                    getframe_srce(nfp, remaddr);
```

```
                strcpy(remid, (char *)&sp->data[crlen]);
                txlen = make_scratchp(nfp, remaddr, 0, FLAG_START, 0, 0);
                newconnstate(STATE_OPEN);
            }                           /* Open up the connection */
        }
        break;

    case SIG_ERR:                       /* Error response? */
        newconnstate(STATE_IDLE);       /* Go idle */
        break;

    case SIG_TIMEOUT:                   /* Timeout on response? */
        n = strlen(remid) + 1;          /* Resend IDENT command */
        txlen = make_scratchp(nfp, bcast, CMD_IDENT, FLAG_CMD, remid, n);
        break;

    case SIG_FAIL:                      /* Failed? */
        newconnstate(STATE_IDLE);       /* Go idle */
        break;
    }
}
else if (connstate == STATE_OPEN)       /* If I requested a connection.. */
{
    switch (sig)                        /* Check signals */
    {
    case SIG_START:
    case SIG_CONN:                      /*  Response OK? */
        buff_setall(&rxbuff, sp->h.seq);
        txlen = make_scratchp(nfp, remaddr, 0, FLAG_CONN, 0, 0);
        newconnstate(STATE_CONNECTED);  /* Send connect, go connected */
        break;

    case SIG_STOP:                      /* Stop already? */
        txlen = make_scratchp(nfp, remaddr, 0, FLAG_STOP, 0, 0);
        newconnstate(STATE_IDLE);       /* Send stop, go idle */
        break;

    case SIG_ERR:                       /* Error response? */
        newconnstate(STATE_IDLE);
        break;                          /* Go idle */
```

```
        case SIG_TIMEOUT:                   /* Timeout on response? */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_START, 0, 0);
            break;                          /* Resend request */

        case SIG_FAIL:                      /* Failed? */
            newconnstate(STATE_IDLE);       /* Go idle */
            break;

        }
    }
    else if (connstate == STATE_CONNECTED)  /* If connected.. */
    {
        switch (sig)                        /* Check signals */
        {
        case SIG_START:                     /* Duplicate START? */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_CONN, 0, 0);
            break;                          /* Still connected */

        case SIG_TIMEOUT:                   /* Timeout on acknowledge? */
            buff_retry(&txbuff, buff_trylen(&txbuff));
                                            /* Rewind data O/P buffer */
            /* Fall through to normal connect.. */
        case SIG_CONN:                      /* If newly connected.. */
        case SIG_NULL:                      /* ..or still connected.. */
            ...
            break;

        case SIG_USER_CLOSE:                /* User closing connection? */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_STOP, 0, 0);
            newconnstate(STATE_CLOSE);      /* Send stop command, go close */
            break;

        case SIG_STOP:                      /* STOP command? */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_STOP, 0, 0);
            newconnstate(STATE_IDLE);       /* Send ack, go idle */
            break;
```

```
        case SIG_ERR:                       /* Error command? */
            newconnstate(STATE_IDLE);       /* Go idle */
            break;

        case SIG_FAIL:                      /* Application failed? */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_ERR, 0, 0);
            newconnstate(STATE_IDLE);       /* Send stop command, go idle */
            break;

        }
    }
    else if (connstate == STATE_CLOSE)      /* If I'm closing connection.. */
    {
        switch (sig)                        /* Check signals */
        {
        case SIG_STOP:                      /* Stop or error command? */
        case SIG_ERR:
            newconnstate(STATE_IDLE);       /* Go idle */
            break;

        case SIG_TIMEOUT:                   /* Timeout on response? */
            txlen = make_scratchp(nfp, remaddr, 0, FLAG_STOP, 0, 0);
            break;                          /* Resend stop command */
        }
    }
    return(txlen);
}
```

The state changes are handled by `newconnstate()`, which allows a simple diagnostic printout if the appropriate debug option is enabled. It also refreshes the time-out timer, on the assumption that no time-out is required if the system is constantly changing state (or re-entering the same state).

```
/* Do a connection state transition, refresh timer, do diagnostic printout */
void newconnstate(int state)
{
    if (state!=connstate)
    {
        if (statedebug)
            printf("connstate %s\n", connstates[state]);
```

```
        if (state != STATE_CONNECTED)
            newappstate(APP_IDLE);          /* If not connected, stop app. */
    }
    connstate = state;
    errcount = 0;
    timeout(&errtimer, 0);                  /* Refresh timeout timer */
}
```

Maintaining a Connection

A connection supports the transfer of data between the two systems. The software must

- send and receive data, keeping in sync with the other node,
- reject duplicate data,
- resend lost data,
- avoid sending too much data to the other node, and
- avoid sending too little data in each packet.

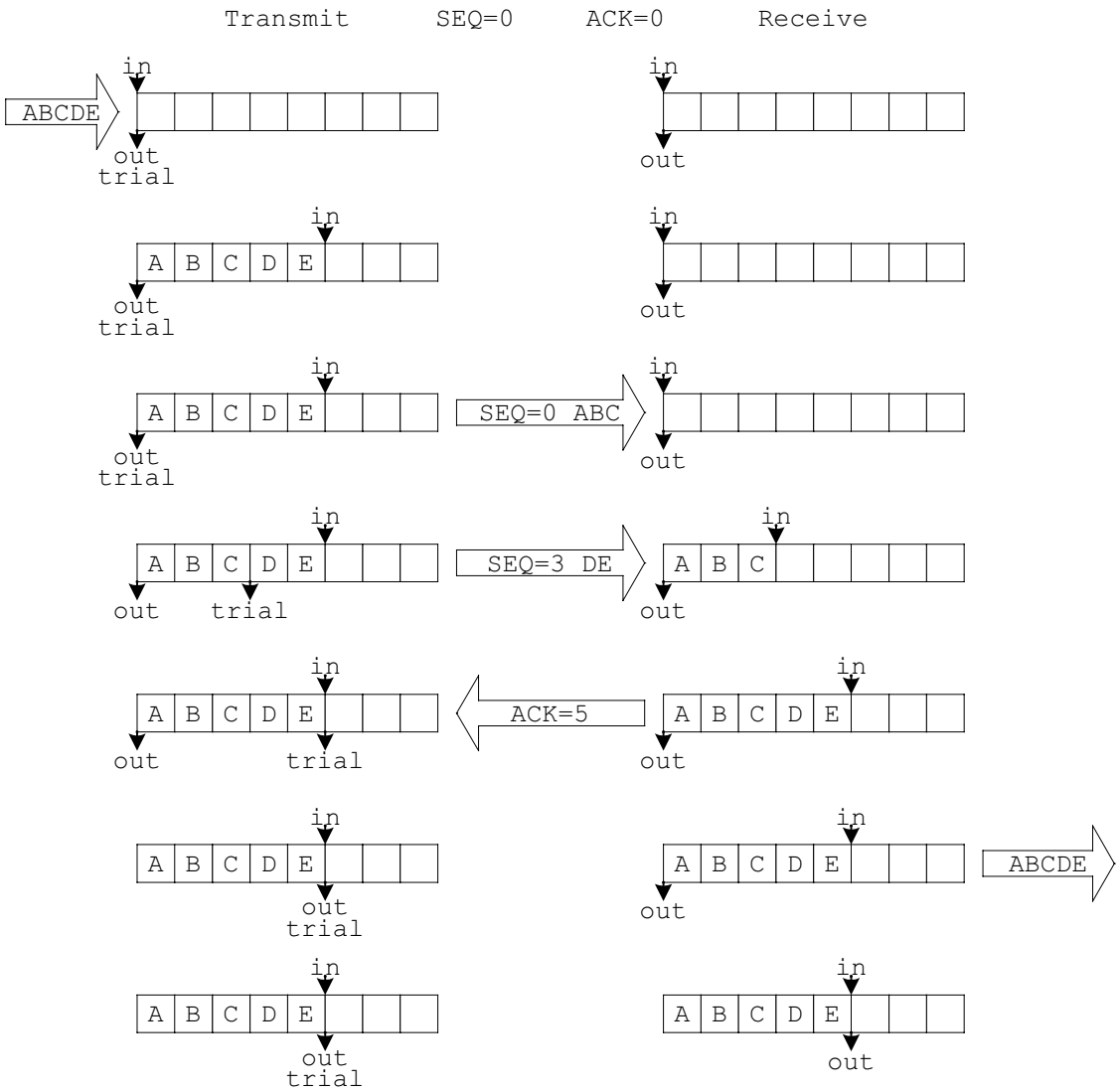
To address the first point, imagine that both nodes have circular buffers of data, and you are simply trying to keep the circular buffer pointers in sync. The circular buffer pointers have 32-bit values (even though the buffer size doesn't warrant it) to allow a simple mapping onto the sequence and acknowledgment values. What is this mapping? Imagine a data block in traveling from one application into the transmit circular buffer, across the network, into the receive circular buffer, and into another application.

Figure 2.9 shows the data `ABCDE` in transit, on the assumption that it had to be transmitted over the network in two blocks, and a single acknowledgment was generated for both blocks.

It can be seen that the sequence pointer for the transfer is equivalent to the sender's `trial` pointer, whereas the acknowledgment value is equivalent to the sender's `in` pointer. This accounts for the following code in the routine used to create SCRATCHP packets.

```
/* Make a SCRATCHP packet given command, flags and data */
int make_scratchp(GENFRAME *nfp, BYTE *dest, char *cmd, BYTE flags,
                  void *data, int dlen)
{
    SCRATCHPKT *sp;
    ...
    sp = (SCRATCHPKT *)getframe_datap(&genframe);
    ...
    sp->h.seq = txbuff.trial;             /* Direct seq/ack mapping.. */
    sp->h.ack = rxbuff.in;                /* ..to my circ buffer pointers! */
    ...
}
```

Figure 2.9 Data flow through a connection.



There are many ways to structure the connection code. The hardest job is to keep a clear indication of how it reaches its decisions as to whether to accept incoming packet data and whether to send data, acknowledgments, or both. First, I present the code for the receive decisions.

```
int do_scratchp(GENFRAME *nfp, int rxlen, int sig)
{
    ...
    LWORD oldrx, rxw, acked=0;
```

```
    ...
            /* Check received packet */
            if (rxlen > 0)                  /* Received packet? */
            {
               newconnstate(connstate);    /* Refresh timeout timer */

               /* Rx seq shows how much of his data he thinks I've received */
               oldrx = rxbuff.in - sp->h.seq; /* Check for his repeat data */
               if (oldrx == 0)                   /* Accept up-to-date data */
                   buff_in(&rxbuff, &sp->data[crlen], dlen);
               else if (oldrx <= WINDOWSIZE)  /* Respond to repeat data.. */
                   tx = 1;                  /* ..with forced (repeat) ack */
               else                          /* Reject out-of-window data */
                   errstr = "invalid SEQ";

               /* Rx ack shows how much of my data he's actually received */
               acked = sp->h.ack - txbuff.out; /* Check amount acked */
               if (acked <= buff_trylen(&txbuff))
                   buff_out(&txbuff, 0, (WORD)acked);  /* My Tx data acked */
               else if (acked > WINDOWSIZE)
                   errstr = "invalid ACK";
               rxw = rxbuff.in - txack;        /* Check Rx window.. */
               if (rxw >= WINDOWSIZE/2)       /* ..force Tx ack if 1/2 full */
                   tx = 1;
               if (errstr)                    /* If error, close connection */
               {
                   printf("Protocol error: %s\n", errstr);
                   txlen = make_scratchp(nfp, remaddr, 0, FLAG_ERR, 0, 0);
                   newconnstate(STATE_IDLE);
               }
            }
    ...
}
```

Usually, the incoming sequence value will equal the Receive buffer `in` value, so the incoming data block can be accepted. If it is not, but it is still within the data window size, then the block is probably a duplicate of a previous one and may be ignored (although the most likely reason for the duplicate is that the latest acknowledgment has gone astray, so it's best to retransmit it). If the incoming data block is outside the data window, then it can't be a duplicate, so an error is flagged.

A similar test is applied to the incoming acknowledgment value. This must be within the data window to be meaningful. If it is outside, it is an error condition.

The decision to transmit is contingent on having data to transmit or a pressing need to send an acknowledgment. It is tempting to generate an acknowledgment for every incoming packet, but this would significantly increase network traffic and the workload of the sender and receiver. Instead, wait until the data window is half full, the sender has duplicated a packet, or you have data to send (don't forget that every one of the data transmissions always has an acknowledgment field). This is hardly an optimal strategy, but it serves reasonably well.

```
            /* Check whether a transmission is needed */
            txw = WINDOWSIZE - buff_trylen(&txbuff);/* Check Tx window space */
            trylen = minw(buff_untriedlen(&txbuff), /* ..size of data avail */
                          minw(SCRATCHPDLEN, txw)); /* ..and max packet len */
            if (trylen>0 || sig==SIG_TIMEOUT || tx) /* If >0, or timeout.. */
            {                                       /* ..or forced Tx.. */
                txlen = make_scratchp(nfp, remaddr, 0, FLAG_CONN, 0, trylen);
                buff_try(&txbuff, sp->data, trylen);/* ..do a transmission */
                txack = rxbuff.in;
            }
            if (buff_trylen(&txbuff) == 0)  /* If all data acked.. */
                newconnstate(connstate);    /* refresh timer (so no timeout) */

            break;
```

The Applications

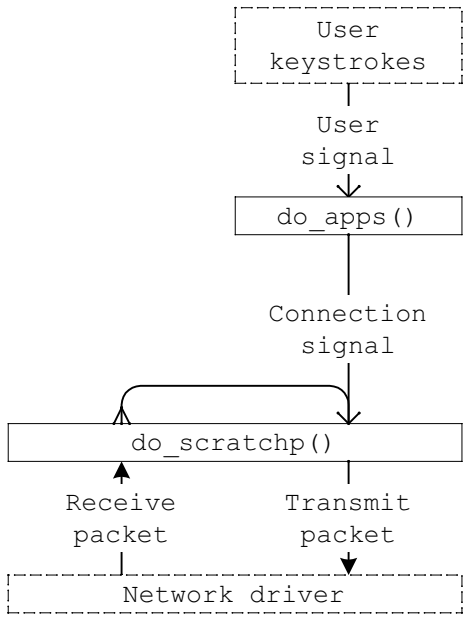
Now that all the hard work of creating, maintaining, and destroying connections is done, there is the relatively simple job of creating application code for

- `ECHO` (connection diagnostic),
- `DIR`ectory of files,
- `GET` (file transfer: read), and
- `PUT` (file transfer: write).

To isolate them from the vagaries of the network, these applications preside over two circular buffers: a Receive buffer that is automatically filled by incoming network data and a Transmit buffer that is automatically emptied into outgoing network packets. As far as the applications are concerned, data transfers are *reliable*. The only error they may see is a catastrophic failure of the connection. All other errors are handled by the lower levels.

To also isolate the applications from the vagaries of the user, they receive predigested user actions in the form of signals. They can also emit signals to the lower layers; for example, to close a connection if the user requests it (Figure 2.10).

Figure 2.10 Application and connection signals.



There is an inherent symmetry between the sending and receiving of files over the connection; to exploit this, I have a *sender* state and a *receiver* state, where a `put` command makes the client a sender and the server a receiver, and the `get` command does the converse.

```
/* Do application-specific tasks, given I/P and O/P buffers, and user signal
** Return a connection signal value, 0 if no signal */
int do_apps(CBUFF *rxb, CBUFF *txb, int usersig)
{
    WORD len;
    BYTE lenb;
    int connsig=0;
    char cmd[CMDLEN+1];

    if (sigdebug && usersig && usersig>=USER_SIGS)
        printf("Signal %s ", signames[usersig]);
    connsig = usersig;                      /* Send signal to connection */
    if (connstate != STATE_CONNECTED)       /* If not connected.. */
        ;                                   /* Do nothing! */
    else if (appstate == APP_IDLE)          /* If application is idle.. */
    {
        if (usersig == SIG_USER_DIR)        /* User requested directory? */
        {                                   /* Send command */
```

```
            buff_in(txb, (BYTE *)CMD_DIR, sizeof(CMD_DIR));
        }
        else if (usersig == SIG_USER_GET)   /* User 'GET' command? */
        {
            filelen = 0;                    /* Open file */
            if ((fhandle = fopen(filename, "wb"))==0)
                printf("Can't open file\n");
            else
            {                               /* Send command & name to remote */
                buff_instr(txb, CMD_GET " ");
                buff_in(txb, (BYTE *)filename, (WORD)(strlen(filename)+1));
                newappstate(APP_FILE_RECEIVER); /* Become receiver */
            }
        }
        else if (usersig == SIG_USER_PUT)   /* User 'PUT' command? */
        {
            filelen = 0;                    /* Open file */
            if ((fhandle = fopen(filename, "rb"))==0)
                printf("Can't open file\n");
            else
            {                               /* Send command & name to remote */
                buff_instr(txb, CMD_PUT " ");
                buff_in(txb, (BYTE *)filename, (WORD)(strlen(filename)+1));
                newappstate(APP_FILE_SENDER);   /* Become sender */
            }
        }
        else if (usersig == SIG_USER_ECHO)  /* User equested echo? */
        {
            buff_in(txb, (BYTE *)CMD_ECHO, sizeof(CMD_ECHO));
            txoff = rxoff = 0;              /* Send echo command */
            newappstate(APP_ECHO_CLIENT);   /* Become echo client */
        }
        else if ((len=buff_strlen(rxb))>0 && len<=CMDLEN)
        {
            len++;                          /* Possible command string? */
            buff_out(rxb, (BYTE *)cmd, len);
            if (!strcmp(cmd, CMD_ECHO))     /* Echo command? */
                newappstate(APP_ECHO_SERVER);   /* Become echo server */
            else if (!strcmp(cmd, CMD_DIR)) /* DIR command? */
                do_dir(txb);                    /* Send DIR O/P to buffer */
```

```
            else if (!strncmp(cmd, CMD_GET, 3)) /* GET command? */
            {                                   /* Try to open file */
                filelen = 0;
                strcpy(filename, &cmd[4]);
                if ((fhandle = fopen(filename, "rb"))!=0)
                    newappstate(APP_FILE_SENDER);   /* If OK, become sender */
                else                            /* If not, respond with null */
                    buff_in(txb, (BYTE *)"\0", 1);
            }
            else if (!strncmp(cmd, CMD_PUT, 3)) /* PUT command? */
            {
                filelen = 0;
                strcpy(filename, &cmd[4]);      /* Try to open file */
                fhandle = fopen(filename, "wb");
                newappstate(APP_FILE_RECEIVER); /* Become receiver */
            }
        }
        else                                /* Default: show data from remote */
        {
            len = buff_out(rxb, apptemp, TESTLEN);
            apptemp[len] = 0;
            printf("%s", apptemp);
        }
    }
    else if (appstate == APP_ECHO_CLIENT)   /* If I'm an echo client.. */
    {
        if (usersig==SIG_USER_CLOSE)        /* User closing connection? */
            newappstate(APP_IDLE);
        else
        {                                   /* Generate echo data.. */
            if ((len = minw(buff_freelen(txb), TESTLEN)) > TESTLEN/2)
            {
                len = rand() % len;             /* ..random data length */
                buff_in(&txbuff, &testdata[txoff], len);
                txoff = (txoff + len) % TESTLEN;/*..move & wrap data pointer*/
            }
            if ((len = buff_out(rxb, apptemp, TESTLEN)) > 0)
            {                                   /* Check response data */
                if (!memcmp(apptemp, &testdata[rxoff], len))
```

```
            {                                   /* ..match with data buffer */
                rxoff = (rxoff + len) % TESTLEN;/*..move & wrap data ptr*/
                testlen += len;
                printf("%lu bytes OK      \r", testlen);
            }
            else
            {
                printf("\nEcho response incorrect!\n");
                connsig = SIG_STOP;     /* If error, close connection */
            }
        }
    }
}
else if (appstate == APP_ECHO_SERVER)   /* If I'm an echo server.. */
{
    if (usersig == SIG_USER_CLOSE)      /* User closing connection? */
        newappstate(APP_IDLE);
    else if ((len = minw(buff_freelen(txb), TESTLEN))>0 &&
             (len = buff_out(rxb, apptemp, len)) > 0)
        buff_in(txb, apptemp, len);     /* Else copy I/P data to O/P */
}
else if (appstate == APP_FILE_RECEIVER) /* If I'm receiving a file.. */
{
    while (buff_try(rxb, &lenb, 1))     /* Get length byte */
    {                                   /* If rest of block absent.. */
        if (buff_untriedlen(rxb) < lenb)
        {
            buff_retry(rxb, 1);         /* .. push length byte back */
            break;
        }
        else
        {
            filelen += lenb;
            buff_out(rxb, 0, 1);        /* Check length */
            if (lenb == 0)              /* If null, end of file */
            {
                if (!fhandle || ferror(fhandle))
                    printf("ERROR writing file\n");
                fclose(fhandle);
```

```
                    fhandle = 0;
                    newappstate(APP_IDLE);
                }
                else                        /* If not null, get block */
                {
                    buff_out(rxb, apptemp, (WORD)lenb);
                    if (fhandle)
                        fwrite(apptemp, 1, lenb, fhandle);
                }
            }
        }
    }
    else if (appstate == APP_FILE_SENDER)  /* If I'm sending a file.. */
    {                                      /* While room for another block.. */
        while (fhandle && buff_freelen(txb)>=BLOCKLEN+2)
        {                                  /* Get block from disk */
            lenb = (BYTE)fread(apptemp, 1, BLOCKLEN, fhandle);
            filelen += lenb;
            buff_in(txb, &lenb, 1);         /* Send length byte */
            buff_in(txb, apptemp, lenb);    /* ..and data */
            if (lenb < BLOCKLEN)            /* If end of file.. */
            {                               /* ..send null length */
                buff_in(txb, (BYTE *)"\0", 1);
                fclose(fhandle);
                fhandle = 0;
                newappstate(APP_IDLE);
            }
        }
    }
    return(connsig);
}
```

Summary

I've looked at the elements of a protocol and how it can be slotted into the ISO standardization framework. There are a lot of decisions to be made when creating a new protocol, and I looked at the client–server model, with both modal and modeless clients. The *logical connection* is at the heart of any reliable data transfer scheme, and connection management (opening, maintaining, and closing the connection) requires very careful organization.

In my implementation of the nonstandard SCRATCHP protocol, I looked at the issues of low-level packet storage and addressing and the strategies for buffering, byte-swapping, transmitting, and receiving packets.

The SCRATCHP utility I developed can be used to evaluate the performance of my protocol or as a test bed for the development of new protocols. It has some of the features of a "real" protocol (address resolution, reliable connection) but is implemented in a much simpler fashion.

The main weakness of my implementation is the inability to handle more than one connection at a time. In future, I'll use the *socket* concept to group together all the information for one connection and support multiple sockets, where each may be in a different state.

Source Files

`ether3c.c`	3C509 Ethernet card driver
`etherne.c`	NE2000 Ethernet card driver
`net.c`	Network interface functions
`netutil.c`	Network utility functions
`pktd.c`	Packet driver (BC only)
`scratchp.c`	SCRATCHP protocol
`serpc.c` or `serwin.c`	Serial drivers (BC or VC)
`dosdef.h`	MS-DOS definitions (BC only)
`ether.h`	Ethernet definitions
`net.h`	Network driver definitions
`netutil.h`	Utility function and general frame definitions
`scratchp.h`	SCRATCHP protocol definitions
`serpc.h`	Serial driver definitions (BC or VC)
`win32def.h`	Win32 definitions (VC only)

SCRATCHP Utility

Utility	Test bed for a nonstandard protocol
Usage	`scratchp [`*`configfile`*`]`
	Reads `tcplean.cfg` from default directory if no file specified
Options	None
Example	`scratchp test.cfg`
Interface	Single keypress with user prompts
	[I] Identify remote node
	[O] Open connection to remote node
	[Q] Quit
	When connected
	[D] Directory of remote
	[E] Echo data test
	[G] Get file from remote
	[P] Put file into remote
Config	`net` to identify network type
	`ident` to identification string for node
Modes	Defaults to server mode unless otherwise directed