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

CMP*Books*

Jeremy Bentham

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**

To Fred, Ilse, and Jane

Table of Contents

Preface

The Lean Plan

This is a hands-on book about TCP/IP (transmission control protocol/Internet protocol) networking. You can browse it to get an overview of the subject or study a particular section in detail, but to get maximum benefit, I suggest you set up your own network and try out the software for real.

Not so long ago, I would have given you a detailed description of a computer network called the Internet and how it allowed academics to pass information between their computers using the TCP/IP protocol family. Now the Internet encroaches all aspects of our lives, so an introduction to it seems totally unnecessary. Yet a hands-on introduction to TCP/IP seems highly necessary, because the very size of the Internet presents a massive barrier to those wishing to understand its inner workings.

My first attempt at implementing TCP was not a great success. I'd waded through the specifications and thought, "this isn't too bad," and waded through the few public domain sources I could find and thought, "this is horrendously complicated," then wrote my own implementation. When I came to test it, the problems started in earnest. I couldn't find a sensible set of software tools for testing; whenever I found a problem, I wasn't sure whether the fault lay with the test software, the software under test, or my understanding of the specification.

What I needed was

- **an implementation I could understand** — not a heavyweight implementation for a large multiuser operating system, but a lightweight one that clearly showed the underlying principles — and
- **software tools I could use**; that is, test utilities that allowed me to check my understanding and implementation of the protocols.

As time went by and my TCP/IP software matured, the Web became increasingly important. My industrial customers would browse the Web at home or work and could see the advantages of using a Web browser for remote control and to monitor their industrial equipment. TCP became just a vehicle for conveying Web pages. The focus shifted from "I want TCP/IP on my system" to "I want my system to produce Web pages," and these pages always included dynamic real-time data.

History was repeating itself; the software to produce these dynamic Web pages was designed for large multiuser systems, and I couldn't find small-scale implementations that were usable on simple, low-cost embedded systems hardware. I needed:

- **a description of the techniques** to insert live data into Web pages and
- **some simple platform-independent code** that I could adapt for specific projects.

Having implemented many small-scale Web servers of my own (generally an 80188 processor with 64Kb of ROM), I was delighted to hear of a 256-byte implementation on a microcontroller, although I was disappointed to discover that it could only produce fixed pages from its ROM, with no dynamic data. I wanted to know:

- **what compromises** were associated with implementing TCP and a Web server on a microcontroller and
- **what techniques** I could use to insert dynamic data into its Web pages.

Almost by chance, the first edition of this book included a miniature Web server running on a PICmicro®[1]. I wasn't the first to create such a server, but I was the first to publish a full description of the techniques used, including full source code. The success of the initial offering prompted me to update this book to broaden the range of networks and protocols supported on the PICmicro. Despite the "Web servers" in the title of this book, there are many ways to transfer data across a network, and I wanted to provide working examples of their use.

Hopefully, you'll find the answers you want in this book.

Embedded Systems

The term "embedded system" may be new to some of you and require some explanation, even though you use embedded systems every day of your life. Microwave ovens, TVs, cars, elevators, and aircraft are all controlled by computers, which don't necessarily have a screen, keyboard, and hard disk. A computer could be controlling your car without your knowledge: an engine management system takes an input signal from the accelerator and provides outputs that control the engine.

These computers are embedded in a system, of which they may be only a small component. The embedded system designer may have to work within tight constraints of size, weight, power consumption, vibration, humidity, electrical interference, and above all, cost and reliability. The PC architecture has been adapted for embedded systems operation, and rugged single-board computers (SBCs) are available from a wide variety of suppliers, together with the necessary add-on cards to process real-world signals. The ultimate in miniaturization

1. PICmicro® is the registered trademark of Microchip Technology Inc.

is the microcontroller, which is a complete computer on a single chip, including all the necessary I/O interfaces.

Regardless of the user interface, most embedded systems have an external interface for status monitoring and system diagnosis. Traditionally this has been in the form of a serial terminal, but industry is starting to see the advantages of remote diagnosis: because Web browser usage is so widespread, it seems the logical choice for a user interface. The browser is technically a Web client, which implies that the embedded system must be a Web server; hence, the title of this book.

Whether you are an embedded systems developer or not, I trust you will find plenty of interest in this book. I'll look at

- what software components are needed,
- how these components work,
- clear, simple implementation, and
- effective test strategies.

The qualities of simplicity and clarity have much to recommend them. Modern programming toolkits are very useful because they can simplify a complex programming task so it becomes a join-the-dots exercise, but the resulting bloated code may require much more complex hardware than the slim-line code of your competitor; hence, the Lean Plan.

The Hardware

At the time of writing, the PC hardware platform, although distinctly showing its age, cannot be ignored. The second-hand market is awash with perfectly serviceable PCs that don't contain the latest and fastest technology but are more than adequate for your purposes. There are low-cost industrial SBCs that have a PC core, standard network interface, and the ability to accept interface cards for a wide variety of real-world signals.

My software will run on all these PC compatibles, and even on PC incompatibles (such as the 80188 CPU) with a very small amount of modification, because I have clearly isolated all hardware and operating-system dependencies.

In addition to the PC code, I have included a miniature TCP/IP stack and Web server for a Microchip PICmicro® microcontroller, using the Custom Computer Services PCM C compiler. A standard PICmicro evaluation board can be hand-modified to include the appropriate peripherals (a circuit diagram is given), or a complete off-the-shelf board can be purchased instead. I won't pretend that it would be easy to adapt this software to another processor, but there is an in-depth analysis of the difficulties associated with microcontroller implementations, which would give you a very significant head-start if working with a different CPU.

The Network

Base-level Ethernet (10Mbit) is still widely available; complete kits, including interface cards and cabling, are available at low cost from computer retailers. My software directly supports two of the most popular Ethernet cards — Novell NE2000 compatibles and 3COM 3C509 — and can potentially (if using the Borland Compiler) support other cards through the packet driver interface, though the direct hardware interface approach is preferable because it makes experimentation and debugging much easier.

When developing network software, you are very strongly advised to use a separate scratch network, completely isolated from all other networks in the building. Not only does debugging become much easier, but you also avoid the possibility of disrupting other network traffic. It is remarkable how a minor change to the software can result in a massive increase in the network traffic and a significant disruption to other network users. You have been warned!

The software also supports serial links through SLIP (serial line Internet protocol), and a crossover serial cable between two PCs can, to a certain extent, be used as a substitute for a real network.

The Operating System

You may be surprised by the extent to which I ignore the operating system. In the embedded systems market, there is always pressure to simplify the hardware and reduce the costs, and one way of achieving this is to use the simplest possible operating system, or none at all.

For those of you wedded to complex operating systems, and even more complex software development environments, this will initially be an uncomfortable experience because you are exposed to the harsh reality of real bare-metal programming. However, I hope that you will soon come to appreciate the power, flexibility, and pure simplicity of this approach and gradually come to the realization that for many common or garden-variety applications, an operating system (even a free operating system) is an expensive luxury. Luxury or not, I want to use my desktop PC for development, so the software is compatible with Windows 95 and 98, either in DOS, extended DOS, or Win32 console application mode.

My primary development system is a Windows 95 machine equipped with two network cards — only one of which is installed in the operating system. This is extremely useful because a single machine can simultaneously act as both network client (using a standard Web browser) and server (using my Web server), making experimentation much easier.

The final target machine can be a relatively humble SBC running DOS or a microcontroller compatible with PC code without an operating system, although the latter would entail some minor changes to the software provided.

The Development Environment

The following four PC compilers are supported.

Borland C++ v3.1. An excellent DOS-hosted compiler with an integrated development environment.

Borland (Inprise) C++ v4.52. Windows-hosted compiler, which seems to be the latest version that can generate executable files for DOS.

Microsoft Visual C++ v6. Windows-hosted compiler that can generate Win32 console applications.

DJGPP v2.02 with RHIDE v1.4. Part of the GNU project, this is a remarkably good clone of the Borland 3.1 development environment, which runs in a 32-bit extended DOS environment and can be downloaded free of charge.

The Borland compilers, though ostensibly obsolete, may be found on the CD-ROM of some C programming tutorial books or may be bundled with their 32-bit cousins. The high-level software can be compiled using all of these environments, but I have not been so fortunate with the low-level network interface code.

- The Borland compilers are the easiest to use because they allow the use of interrupts without the need for machine code inserts and so can support the full range of network interfaces.
- With the Microsoft compiler, the network card and SLIP interfaces are supported, but the packet driver interface is not.
- Only the direct network card interface is supported when using the DJGPP compiler.

Because the direct network card interface is the easiest to debug, and hence more suitable for experimentation, this restriction isn't as onerous as it might appear.

If your favorite compiler isn't on the list, I apologize for the omission, but I am very unlikely to add it. Each compiler represents a very significant amount of testing, and my preference is to reduce, rather than increase, the number of compilers supported. If your compiler is similar to the above (for example, an earlier version), then you should have little or no adaptation work to perform, though I can't comment on any compiler I haven't tried.

PICmicro Compilers. The early software used the Custom Computer Services (CCS) PCM v2.693, but later developments are broadly compatible with the CCS and Hitech compilers for the PIC16xxx and PIC18xxx series microcontrollers. A detailed discussion of compatibility issues is beyond the scope of this chapter. See Appendix D and the software release notes on the CD-ROM for more information.

The Software

The enclosed CD-ROM contains complete source code to everything in this book so that you, as purchaser of the book, can experiment. However, the author retains full copyright to the software, and it may only be distributed in conjunction with the book; for example, you may not post any of the source code on the Internet or misrepresent its authorship by extracting fragments or altering the copyright notices.

If you want to sell anything that contains this software, a license is required for the "incorporation" of the software into each commercial product. This normally takes the form of a one-off payment that allows unlimited incorporation of any executable code derived from this source. There are no additional development fees (apart from purchase of the book), and license fees are kept low to encourage commercial usage. Full details and software updates are on the Iosoft Ltd. Web site at `www.iosoft.co.uk`.

Acknowledgments

The author owes a profound debt of gratitude to Berney Williams of CMP Books for being so keen on this project, Anthony Winter for his proofreading skills and advice, Glen Middleton of Arcom Control Systems Ltd. and Adrian Nicol of Io Ltd. for their help with the hardware, and, above all, to Jane McSweeney (now Jane Bentham) for her continued enthusiasm, support, and wonderful cakes.

1

Chapter 1

Introduction

The Lean Plan

This is a software book, so it contains a lot of code, most of which has been specially written (or specially adapted) for the book. The software isn't a museum piece, to be studied in a glass case, but rather a construction kit, to promote understanding through experimentation. The text is interspersed with source code fragments that illustrate the points being discussed and provide working examples of theoretical concepts. All the source code in the book, and complete project configurations for various compilers, are on the enclosed CD-ROM.

When I started writing this book, I intended to concentrate on the protocol aspects of embedded Web servers, but I came to realize that the techniques of providing dynamic content (on-the-fly Web page generation) and client/server data transfers were equally important, yet relatively unexplored. Here are some reasons for studying this book.

TCP/IP. You want to understand the inner workings of TCP/IP and need some tools and utilities to experiment with.

Dynamic Web Content. You have an embedded TCP/IP stack and need to insert dynamic data into the Web pages.

Miniaturization. You are interested in incorporating a miniature Web server in your system but need to understand what resources are required and what compromises will have to be made.

Prototyping. You want a prebuilt Web server that you can customize to evaluate the concept in a proposed application.

Data transfer. You need to transfer data across a network using standard protocols.

Client/server programming. You have to interface to standard TCP/IP applications, such as email servers.

Of course, these areas are not mutually exclusive, but I do understand that you may not want to read this book in a strict linear order. As far as possible, each chapter stands on its own and provides a stand-alone utility that allows you to experiment with the concepts discussed.

I won't assume any prior experience with network protocols, just a working knowledge of the C programming language. In the Preface, I detailed the hardware and software you would need to take full advantage of the source code in the book. You don't have to treat this book as a hands-on software development exercise, but it would help your understanding if you did.

Getting Started

On the CD-ROM, you'll find the directory `tcplean` with several subdirectories.

`BC31`	compiler-specific files for Borland C++ v3.1
`BC45`	compiler-specific files for Borland C++ v4.52
`DJGPP`	compiler-specific files for (GNU) DJGPP and RHIDE
`PCM`	the PICmicro®-specific[1] files for Chapters 9–11
`ROMDOCS`	sample documents for the PICmicro Web server
`SOURCE`	all source code for PC systems
`VC6`	compiler-specific files for Microsoft Visual C++ v6
`WEBDOCS`	sample documents for the PC Web server

You'll also find the directory `chipweb` with a two subdirectories containing the files for Chapters 12–16.

`ARCHIVE`	zip files containing older versions of the ChipWeb source code
`P16WEB`	latest ChipWeb source code

Executable copies of all the utilities, sample configuration files, and a `README` file with any late-breaking update information are in `tcplean`. Preferably, the complete directory tree `d:\tcplean` (where `d:` is the CD-ROM drive) should be copied to `c:\tcplean` on your hard disk,

1. PICmicro® is the registered trademark of Microchip Technology Inc.; PICDEM.net™ is the trademark of Microchip Technology Inc.

and `d:\chipweb` to `c:\chipweb`. If a different directory path is used, it will be necessary to edit the compiler project files.

The utilities read a configuration file to identify the network parameters and hardware configuration; the default is `tcplean.cfg`, read from the current working directory. It is unlikely that this will contain the correct hardware configuration for your system, so *it is important that you change the configuration file before running any of the utilities*. See Appendix A for details. If you attempt to use my default configuration without checking its suitability, it may conflict with your current operating system settings and cause a lockup.

It is possible to browse the source files on the CD-ROM and execute the utilities on it without loading them onto your hard disk, though you still need a to adapt the configuration file and store it in the current working directory.

```
c:\>cd tcplean
c:\tcplean>d:\tcplean\ping 10.1.1.1
```

This would execute the utility on the CD-ROM using the configuration file.

```
c:\tcplean\tcplean.cfg
```

The default configuration file may be overridden using the `-c` command-line option.

```
c:\tcplean>ping -c slip 172.16.1.1
```

This uses the alternative configuration file `slip.cfg`, which makes it possible to experiment with multiple network configurations without having to rebuild the software each time.

If you are in any doubt about the command-line arguments for a utility, use the `-?` option.

```
c:\>cd tcplean
c:\tcplean>ping -?
```

Some of the utilities have the same name as their DOS counterparts (because they do the same job), so it is important to change to `tcplean` before attempting to run them.

A final word of warning: I strongly recommend that you create a new “scratch” network for your experimentation that is completely isolated from all other networks in the building. It is a very bad idea to experiment on a “live” network.

Network Configuration

The DOS software in this book supports the following network hardware.

Direct-drive network card Novell NE2000-compatible or 3COM 3C509 Ethernet cards can be direct-driven by the software. This is the preferred option because of the ease of configuration and debugging.

Serial link A serial line Internet protocol (SLIP) link between two PCs or a PC and the PICmicro miniature Web server.

Packet driver An otherwise unsupported network card may be used via a Crynwr packet driver supplied by the card manufacturer.

Some combinations of network hardware and compiler are not supported. Consult Appendix A and the README file for full information on the network configuration options.

Compiler Configuration

Executable versions of all the DOS projects are included within the `tcplean` directory, so initial experimentation can take place without a compiler. The project files for each compiler reside in a separate directory, as described earlier, and all the compiler configuration information resides within the project files. All the source code files reside in a single shared directory. There are a few instances where compiler-specific code (generally Win32-specific code) must be generated, in which case automatic conditional compilation is used.

Load specific projects for the following compilers:

Borland C++ v3.1 In a DOS box, change to the `BC31` directory and run BC using the project filename.

```
c:\>cd \tcplean\bc31
c:\tcplean\bc31>bc ping.prj
```

Borland C++ v4.52 Launch the Integrated Development Environment (IDE) and select Project–Open Project and the desired IDE file in the `BC45` directory.

DJGPP and RHIDE Launch the RHIDE IDE and select Project–Open Project and the desired GPR file in the `DJGPP` directory.

Visual C++ v6 Launch the IDE and select File–Open Workspace and the desired DSW file in the `VC6` directory.

Custom Computer Services PCM The PICmicro cross-compiler uses a completely different set of source code that resides in the PCM directory. Open a DOS box and change directory to `\tcplean\pcm`. Copy the necessary system files (`16C76.h` and `ctype.h`) into this directory from the standard PCM distribution. Run the PCM compiler, specifying `PWEB.C` on the command line.

```
c:\>cd \tcplean\pcm
c:\tcplean\pcm>copy \picc\examples\16c76.h
c:\tcplean\pcm>copy \picc\examples\ctype.h
c:\tcplean\pcm>\picc\pcm pweb.c
```

I run the PCM compiler from within the IDE of an emulator; see the emulator documentation for details on how to do this. When first using such a setup, make a minor but readily observable change, rebuild, and check that the new executable really has been downloaded into the emulator. It is all too easy to omit a vital step in the rebuild chain, such that the old file is still being executed.

Other PICmicro® Compilers

The software in Chapters 12–16 is broadly compatible with the later versions of the CCS and Hitech PICmicro compilers, for both the PIC16xxx and PIC18xxx series of devices. There are compatibility issues with some versions of these compilers; see Appendix D for guidance on compiler-specific issues, and always refer to the release notes (in file `readme.txt`) before using a specific ChipWeb release.

Software Introduction

For the rest of this chapter, I'll look at the low-level hardware and software functions needed to support software development.

- network hardware characteristics
- network device drivers
- process timing
- state machines
- buffering
- coding conventions

Even if you're keen to get on with the protocols, I suggest you at least skim this material, since it forms the groundwork for the later chapters.

Network Hardware

To help in setting up a serial or network link, I've included some sample configurations in Appendix A, together with the relevant software installations. Assuming one or both are installed, I will examine their characteristics with a view to producing the low-level hardware device drivers.

Figure 1.1 Serial link and network topologies.

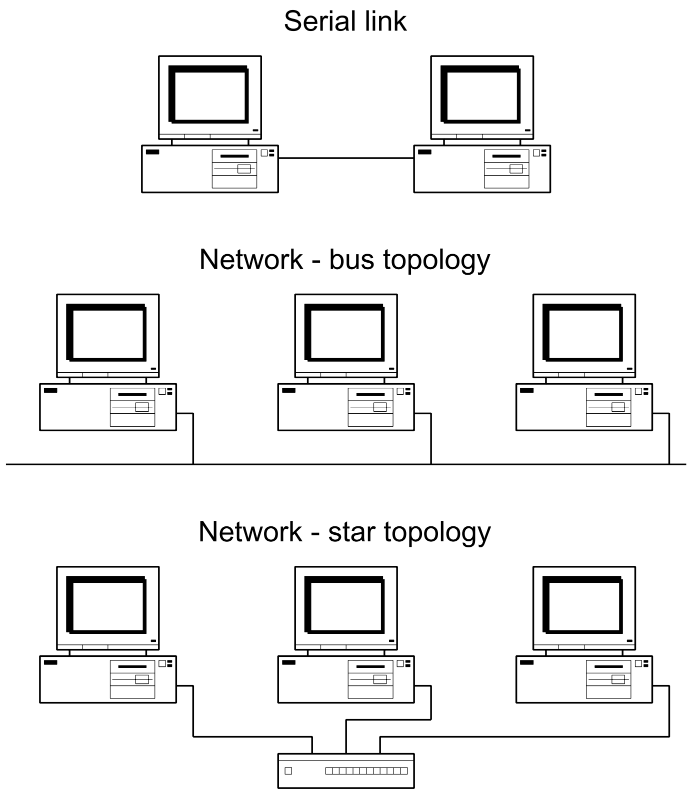


Figure 1.1 shows two types of networks (two “topologies”): the older style bus network, where the computers are connected to a single common cable, and the newer star network, where the computers are individually connected to a common box (a hub), which electrically copies the network signals from one computer to all others. Fortunately, the operation of an Ethernet hub is completely transparent to the software, so you can still treat the network as if the computers were sharing a common cable.

Serial Hardware Characteristics

The simplest communication link between two PCs (A and B) consists of three wires: a ground connection, a wire from the A transmit to the B receive, and a wire from the B transmit to the A receive. A commercial serial crossover cable (often called a null modem or “Laplink” cable) generally has more wires connected so that the handshake signals are transferred, but you’ll concentrate on the two data lines, which have the following characteristics.

Both computers have equal access to the serial link. The hardware simply acts as a “data pipe” between the two computers and does not prioritize one computer above another.

There are only two computers (nodes) on the network. Throughout this book, I’ll use “node” as shorthand for “a computer on the network.” Insofar as the simple serial link constitutes a network, it is clear that if one node transmits a message, it can only be received by the other node and no others.

A node can transmit data at any time. This is technically known as a full duplex system; both computers can transmit and receive simultaneously without any clash of data signals.

Message delivery is reliable. The assumption is that the two nodes are close to each other, with a short connecting cable, so there will be no corruption of data in transit. The predominant failure mode is a catastrophic link failure, such as a disconnection of the cable or a node powering down.

The serial data is a free-format stream of bytes, with little or no integrity checking. The serial hardware is only designed for short-distance interconnects, so it has a very simple error-checking scheme (parity bit), which is often disabled. To guarantee message integrity, error checking must be provided in software.

There is no limit on message size. Because the serial data is simply a stream of bytes with no predefined start or end, there is no physical restriction on its length.

There is no need for addressing Because there is only one possible recipient for each message, there is no need to include an address identifying that recipient.

Network Hardware Characteristics

Whatever the actual topology, a base-level Ethernet network appears logically to be two or more computers transmitting and receiving on a single shared medium (cable).

All computers on the network have equal access to the network. This is called peer-to-peer networking, in which all nodes are equal. The alternative (master–slave networking) assumes that one or more special nodes control and regulate all network traffic; they are the masters, and their slaves only speak when spoken to. Master–slave operation is very useful for industrial data acquisition, where all data and control is to be funneled through a few large computer systems but prohibits the kind of ad hoc communication that is required in an office or on the Internet.

All nodes have a 48-bit address that is unique on the network. Just as a postal address uniquely identifies a specific location in the world, so a node address (generally known as a media access and control, or MAC, address) must uniquely identify a node on the network. In fact, the standardization of Ethernet guarantees each node address to be also unique in the world; you can mix and match Ethernet adaptors from different manufacturers, secure in the knowledge that no two will have the same 48-bit address.

Any node may transmit on the network when it is idle. If a node is to communicate with another, it must wait for all others to be silent before it can transmit. Because all nodes are equal, they need not ask permission before transmitting on the network; they simply wait for a suitable gap in the network traffic.

Message delivery is unreliable. "Unreliable? Why don't you fix it?" Networks are, by their very nature, an unreliable way of sending data. The failure modes range from the catastrophic (the recipient's computer is powered down or physically disconnected from the network) to the intermittent (a packet has been corrupted by collision or electrical interference). The network hardware has the ability to detect and compensate for some intermittent faults (e.g., a retry in the event of a packet collision), but eventually an error will occur that has to be handled in software, so the software must assume the network is unreliable.

All data on the network is in blocks (frames) with a defined beginning and end and an integrity check. Nodes that are going to transmit when they want need a defined format for their transmissions so that others know when they are starting or finishing, assuming each transmission is a block with start and end markers and some form of checking (usually a CRC, or cyclic redundancy check) to ensure it hasn't been damaged in transit. The name given to this block differs according to the network used; Ethernet blocks are called frames.

The network can send a maximum of 1,500 bytes of data per frame. All networks have an upper limit on the size of data they can carry in one frame. This is called the maximum transfer unit, or MTU. Ethernet frames can contain up to 1.5Kb, but TCP/IP software will work satisfactorily with a lot smaller MTU.

All messages are equipped with a source and destination address. Frames are usually intended for a single recipient; this is known as unicast transmission. Occasionally, it may be necessary to send a frame to all nodes on the network, which is a broadcast transmission.

Device Drivers

It would be helpful if the driver software presented a common interface to the higher-level code, but it is clear from the preceding analysis that there are significant differences; these are summarized in Table 1.1.

Serial Driver Requirements

TCP/IP assumes the network data is sent in blocks, with a defined beginning and end, so the serial drivers must convert the free-format serial byte stream into well-defined blocks.

Table 1.1 RS232 serial versus Ethernet.

	RS232 Serial	Ethernet
Access	Equal	Equal
Address range	None	48-bit
Transmit	Any time	When network is idle
Delivery	Reliable	Unreliable
Format	None (data stream)	Frame
Data length	Unlimited	1.5Kb per frame
Addressing	None	Source, destination, broadcast

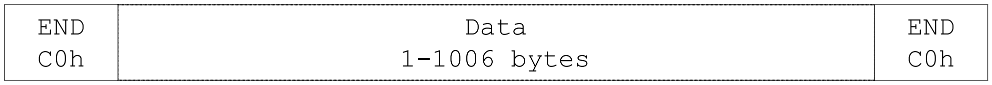
SLIP

Fortunately, one of the TCP/IP families of standards, SLIP, provides exactly this functionality. It uses simple escape codes inserted in the serial data stream to signal block boundaries as follows.

- The end of each block is signaled by a special End byte, with a value of `C0h`.
- If a data byte equals `C0h`, two bytes with the values `DB`, `DC` are sent instead.
- If a data byte equals `DBh`, two bytes with the values `DB`, `DD` are sent instead.

Additionally, most implementations send the End byte at the beginning of each block to clear out garbage characters prior to starting the new message (Figure 1.2).

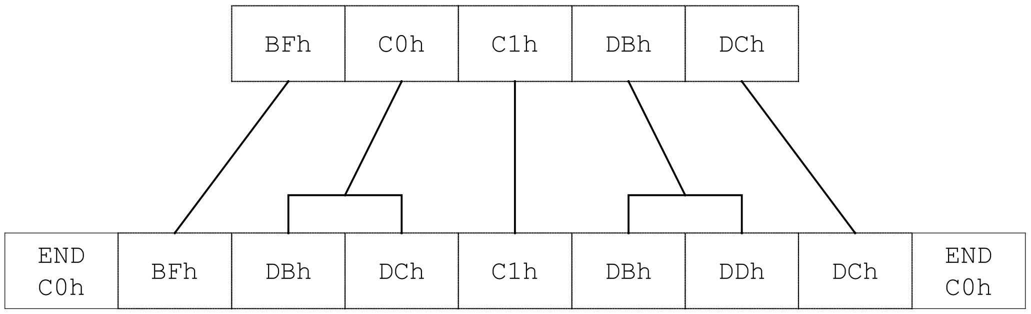
Figure 1.2 SLIP frame.



There is effectively no limit to the size of the data block, but you have to decide on some value in order to dimension the data buffers. With old slow serial links, a maximum size of 256 bytes was generally used, but you'll be using faster links, and a larger size is better for minimizing protocol overhead. By convention, 1,006 bytes is often used.

The encoding method can best be illustrated by an example (Figure 1.3). Assume a six-byte block of data with the hex values `BF C0 C1 DB DC` is sent; it is expanded to `C0 BF DB DC C1 DB DD DC C0`.

Figure 1.3 SLIP example.

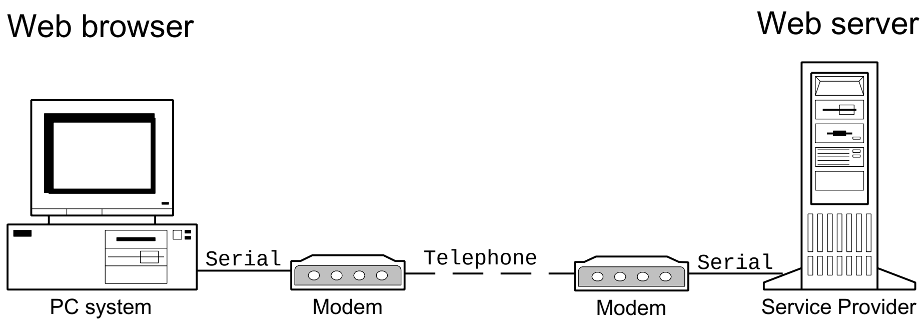


The original data has nearly doubled in size, due to my deliberately awkward choice of data values. In normal data streams, the overhead is much lower.

Modem Emulation

An additional problem with serial networking is that most PCs are configured to use a modem (Figure 1.4) to an Internet Service Provider (ISP).

Figure 1.4 Modem communication.



I'll create a Web server, but instead of two modems, I'll use a serial (null modem) cable to link it to the browser. The problem is that my Web server will then receive the browser's commands to its modem. If these go unanswered, the browser will assume its modem is faulty and report this to the user.

The easiest solution is to include a simple modem emulator in your serial driver so that the browser is fooled into thinking it is talking to a modem. Because modem commands are text based, you can easily distinguish between them and the SLIP message blocks prefixed by the delimiter character (`C0h`); when the latter appears, disengage the modem emulation.

Modem commands begin with the uppercase letters `AT`, followed by zero or more alphabetic command letters, with alphabetic or numeric arguments, terminated by an ASCII carriage return (`<CR>`) character. The usual reply are the uppercase letters `OK`, followed by a carriage return and line feed (`<CR><LF>`). Table 1.2 shows a few typical command–response sequences for a simple modem. This emulation would respond OK to all commands; this is normally sufficient.

Table 1.2 Modem command–response sequences.

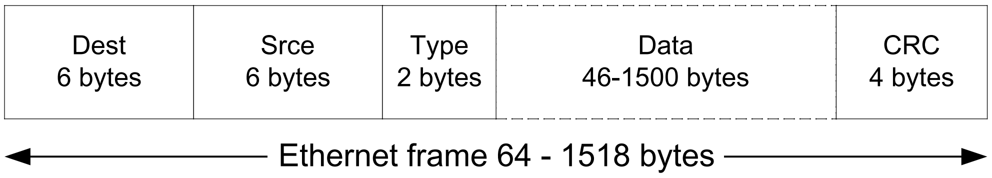
Browser	Modem	Response
AT<CR>	OK<CR><LF>	Check modem present
ATZ<CR>	OK<CR><LF>	Reset modem
ATDT12345	OK<CR><LF> CONNECT 38400<CR><LF>	Dial phone number

Ethernet Driver Requirements

The Ethernet message (frame) is necessarily more complicated than the serial message (Figure 1.5). It contains the

- destination address,
- source address,
- type/length field,
- data, and
- cyclic redundancy check (CRC).

Figure 1.5 Ethernet frame.



It is traditional to include the CRC when quoting the Ethernet frame size (e.g. a maximum frame size of 1518 bytes), even though it is ignored by the software, and is usually removed by the lower-level driver code.

```
#define MACLEN      6           /* Ethernet (MAC) address length */

/* Ehernet hardware Rx frame length includes the trailing CRC */
#define MAXFRAMEC   1518        /* Maximum frame size (incl CRC) */
#define MINFRAMEC   64          /* Minimum frame size (incl CRC) */

/* Higher-level drivers exclude the CRC from the frame length */
#define MAXFRAME    1514        /* Maximum frame size (excl CRC) */
#define MINFRAME    60          /* Minimum frame size (excl CRC) */
```

```
/* Ethernet (DIX) header */
typedef struct {
    BYTE dest[MACLEN];          /* Destination MAC address */
    BYTE srce[MACLEN];          /* Source MAC address */
    WORD ptype;                 /* Protocol type or length */
} ETHERHDR;

/* Ethernet (DIX) frame; data size is frame size minus header & CRC */
#define ETHERMTU (MAXFRAME-sizeof(ETHERHDR))
typedef struct {
    ETHERHDR h;                 /* Header */
    BYTE data[ETHERMTU];        /* Data */
    LWORD crc;                  /* CRC */
} ETHERFRAME;
```

This is the basic Ethernet frame, also known as Ethernet 2 (Ethernet 1 is obsolete), or DIX Ethernet (after its creators, DEC, Intel, and Xerox).

Destination and Source Addresses

These six-byte values identify the recipient and sender of the frame and are generally known as media access and control (MAC) addresses. They are standardized by the IEEE; the first three bytes identify the network hardware vendor, and the next three are used by that vendor to guarantee the address is unique, so they are different for every network adaptor that the manufacturer has ever produced.

Each adaptor has its six-byte address burned into a memory device at manufacture, but it is normally the responsibility of the networking software to copy this value into the appropriate field of the network packet. A destination address of all ones indicates a broadcast address.

Type/Length Field

Unfortunately, there are several Ethernet standards, and they make different use of this two-byte field. One standard uses it as a length, giving the total count of bytes in the data field. Others use it as a protocol type, indicating the protocol that is being used in the data field. Mercifully there are simple ways of detecting and handling these standards, which are discussed in Chapter 3.

Data

This area contains user data in any format; the only restrictions are that its minimum size is 46 bytes and its maximum is 1,500 bytes. The minimum is necessary to ensure that the overall frame is at least 64 bytes. If it were smaller, there would be a danger that frame collisions wouldn’t be detected on large networks.

Cyclic Redundancy Check

This is a check value that allows the network controller to discard corrupted frames. It is automatically appended by the Ethernet controller on transmit and checked on receive. The bit-by-bit algorithm is particularly suited to hardware implementation. The following code fragment is equivalent but operates on byte values.

```
#define ETHERPOLY 0xedb88320L

/* Update CRC for next input byte */
unsigned long crc32(unsigned long crc, unsigned char b)
{
    int i;

    for (i=0; i<8; i++)
    {
        if ((crc ^ b) & 1)
            crc = (crc >> 1) ^ ETHERPOLY;
        else
            crc >>= 1;
        b >>= 1;
    }
    return(crc);
}
```

A starting CRC value of FFFFFFFFh is sent to this function, together with the first byte value. A new CRC value is returned, which is sent to this function together with the next byte value, and so on. When all bytes have been processed, the final CRC value is inverted (one's complement) to produce the four-byte Ethernet CRC, which would be transmitted least significant byte first.

Generic Driver Functions

You need some generic network driver functions that are usable for a variety of network types and hardware configurations. This node-specific information will be in a configuration file and read from disk at boot time. The following code fragments show what a line in this file might look like.

```
net ether ne 0x280
```

This specifies an Ethernet interface using an NE2000-compatible card at I/O address 280h. See Appendix A for details on the cards and networks supported.

This string passed to a network initialization function, to open the required interface.

```
WORD open_net(char *cfgstr);
```

This function opens up the network driver, given a string specifying the type of driver and configuration parameters, and returns a driver type, which must be used in all subsequent accesses, or a 0 on error (e.g., when the hardware is in use by other software).

```
void close_net(WORD dtype);
```

This function shuts down the network driver. The returned value for the driver type serves two purposes: it provides a unique handle for the interface, and its flags inform you of the type of interface in use. This allows you to create software that can handle multiple network interfaces, each with different hardware characteristics.

You need a generic frame that can accommodate any one of the different frame types. Its header includes the driver type.

```
/* General-purpose frame header, and frame including header */
typedef struct {
    WORD len;                   /* Length of data in genframe buffer */
    WORD dtype;                 /* Driver type */
    WORD fragoff;               /* Offset of fragment within buffer */
} GENHDR;

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

The header also has a length word to assist in low-level buffering (e.g., polygonal buffering, described later) and support for fragmentation. This is where a frame that exceeds the MTU size is broken up, sent as two smaller frames, and reassembled at the far end. This will be discussed further in Chapter 3; for now, you need to be aware that the maximum frame size (MAXGEN in the above definitions) need not be constrained to the maximum Ethernet frame size. You'll use a MAXGEN of just over 3Kb, so two complete Ethernet frames can be stored in the one GENFRAME.

Having standardized on a generic frame, you can create the driver functions to read and write these frames.

`WORD get_net(GENFRAME *gfp);` Checks for an incoming frame. If present, it copies it into the given buffer and returns the data length. If there is no frame, it returns 0.

`WORD put_net(GENFRAME *gfp, WORD len);` Sends a frame, given its length, and returns the total transmitted length or 0 if error.

You don't need to specify which network interface is used because the function can examine the driver-type field to determine this. Sample device drivers have been included on the CD-ROM, but they will not be discussed here because they are highly specific to the hardware (and operating system).

Configuration File Format

As part of the experimentation in this book, you'll frequently need to change the software parameters at run time. Because it is tedious to type these in every time the program runs, they'll be incorporated into a configuration file called tcplean.cfg. By default, utilities will read this file from the default file path, although an alternative configuration filename can be specified on the command line.

The file consists of ASCII text lines, each line referring to one configuration item.

```
# TCP/IP Lean configuration file

net     ether ne 0x280
id      node1
ip      10.1.1.1
gate    10.1.1.111

# EOF
```

Blank lines, or lines beginning with #, are treated as comments. At the start of each line is a single lowercase configuration parameter name delimited by white space and followed by a string giving the required parameter value(s).

The content of the file is specific to the software being run; if any configuration parameter is unrecognized, it is ignored. In the above example, the net entry defines the network driver to be used and its base I/O address. The node name is identified as node1, with IP address 10.1.1.1 and gateway address 10.1.1.111 given. Appendix A gives guidance on how to customize the configuration file for the network hardware you are using.

Process Timer

When implementing a protocol, an event for a future time is often scheduled. Whenever you send a packet on the network, you must assume that it, or the response to it, might go astray. After a suitable time has elapsed, you may want to attempt a retry or alert the user.

Most modern operating systems have a built-in provision for scheduling such events, but I am very keen to keep the code Operating System (OS) independent and to be able to run it on the bare metal of small embedded systems. To this end, my software includes a minimal event scheduler of its own, which requires a minimum of OS support and can be adapted to use the specific features of your favorite OS.

The simplest scheduling algorithm is to delay between one event and another.

```
putpacket(...);             /* Packet Tx */
delay(2000);                /* Wait 2 seconds */
if (getpacket(...))         /* Check for packet Rx */
{
    /* Handle response packet */
}
else
{
    /* Handle error condition */
}
```

The dead time between transmission and reception is highly inefficient. If the response arrives within 100 milliseconds (ms), the system would wait a further 900ms before processing it. With a multitasking OS, you could use `sleep` instead of `delay`, which would wake up on time-out or when the packet arrived (a method called blocking, since it blocks execution until an event occurs). An alternative pseudo-multitasking method is to use timer interrupts to keep track of elapsed time and to initiate corrective action as necessary, but this approach would be highly specific to the OS.

A simple compromise, not entirely unfamiliar to old-style Windows programmers, is to have the software check for its own events and handle them appropriately.

```
putpacket(...);          /* Packet Tx */
timeout(&txtimer, 0);    /* Start timer */
while (1)
{
    /* Check for packet Rx */
    if (getpacket(...))
    {
        /* Handle response packet */
    }
    /* Check for timeout on response */
    else if (timeout(&txtimer, 2))
    {
        /* Handle error condition */
    }
    /* Check for other events */
    else if ...

}
```

The `timeout()` function takes two arguments: the first is a pointer to a variable that will hold the starting time (tick count), and the second is the required time-out in seconds. When the time-out is exceeded, the function triggers an event by reloading the starting time with the current time and returning a non-zero value. For example, the following code fragment prints a seconds count every second.

```
WORD sectimer, secs=0;

timeout(&sectimer, 0);
while (1)
{
    if (timeout(&sectimer, 1))
        printf("%u sec\n", ++secs);
}
```

Before a timer is used, a timeout() call must be made using time value 0. This forces an immediate time-out, which loads the current (starting) time into the timer variable. The tim-eout() function is easy to implement, providing you take care with the data types.

```
/* Check for timeout on a given tick counter, return non-zero if true */
int timeout(WORD *timep, int sec)
{

    WORD tim, diff;
    int tout=0;

    tim = (WORD)time(0);
    diff = tim - *timep;
    if (sec==0 || diff>=sec)
    {
        *timep = tim;
        tout = 1;
    }
    return(tout);
}
```

If the use of unsigned arithmetic appears counterintuitive, consider the following code.

```
WORD a, b, diff;
a = <any starting value>;
b = a + 10;
diff = b - a;
```

What is the value of diff? It must be 10, whatever the starting value.

There is a hidden trap that is due to timer granularity. The if statement in the code

```
timeout(&sectimer, 0);
if (timeout(&sectimer, 1))
  …
```

will sometimes return TRUE, even though much less than a second has elapsed. This is because the two statements happen to bracket a timer tick, so it appears that one second has elapsed when it has not.

A cure for this problem is to change the unit of measurement to milliseconds, although the nonstandard millisecond timer, mstime(), must be coded for each operating system.

```
/* Check for timeout on a given msec counter, return non-zero if true */
int mstimeout(LWORD *timep, int msec)
{
    LWORD tim;
    long diff;
    int tout=0;

    tim = mstime();
    diff = tim - *timep;
    if (msec==0 || diff>=msec)
    {
        *timep = tim;
        tout = 1;
    }
    return(tout);
}
```

Alternatively, you can just document this feature by saying that there is a tolerance of –1/+0 seconds on the time measurement. Given this timing tolerance, you might be surprised that my trivial example of printing seconds works as suggested.

```
WORD sectimer, secs=0;

timeout(&sectimer, 0);
while (1)
{
    if (timeout(&sectimer, 1))
        printf("%u sec\n", ++secs);
}
```

It works because the state changes in the main loop are locked to the timer tick changes. The whole operation has become synchronous with the timer, so after a random delay of up to one second, the one-second ticks are displayed correctly.

When working with protocols, you will frequently see software processes synchronizing with external events, such as the arrival of data frames, to form a pseudo-synchronous system. When testing your software, you must be sure that this rhythm is regularly disrupted (e.g., by interleaving accesses to another system) to ensure adequate test coverage.

State Machines

When learning to program, I always avoided state machines and skipped the examples (which always seemed to be based on traffic lights) because I couldn't see the point. Why go to all the effort of drawing those awkward diagrams when a simple bit of procedural code would do the job very effectively?

Tackling network protocols finally convinced me of the error of my ways. You may think a network transaction is a tightly specified sequence of events that can be handled by simple procedural code, but that is to deny the unpredictability (or unreliability, as I discussed earlier) of any network. In the middle of an orderly transaction, your software might see some strangely inconsistent data, perhaps caused by a bug in the someone else's software or your own. Either way, your software must make a sensible response to this situation, and it can't do that if you didn't plan for this possibility. True, you can't foresee every problem that may occur, but with proper analysis you can foresee every *type* of problem and write in a strategy to handle it.

Only the simplest of network transactions are stateless; that is, neither side needs to keep any state information about the other. Usually, each side keeps track of the other and uses the network to

- signal a change of state,
- signal the other machine to change its state, or
- check whether the other machine has signaled a change of state.

The key word is *signal*. Signals are sent and received over the network to ensure that two machines remain in sync; that is, they track each other's state changes. The signals may be explicit (an indicator variable set to a specific value) or implicit (a quantity exceeding a given threshold). Either way, the signals must be detected and tracked by the recipient.

Any error in this tracking will usually lead to a rapid breakdown in communications. When such problems occur, inexperienced network programmers tend to concentrate on the data, rather than the states. If a file transfer fails, they might seek deep meaning in the actual number of bytes transferred, whereas an older hand would try to establish whether a state change had occurred and what caused it at the moment of failure. This process is made much easier if the protocol software has specifically defined states and has the ability to display or log the state information while it is running.

At the risk of creating a chapter that you will skip, I'd like to present a simple, worked example of state machine design, showing the relationship between state diagram, state table, and software for a simple communications device, the telephone.

Telephone State Machine

If you ignore outgoing calls, what states can a telephone be in?

Idle	on-hook, unused
Ringing	on-hook, bell ringing
Connected	off-hook, connected to another phone
Sending	sending speech to other phone
Receiving	receiving speech from other phone

The last two states are debatable, since a telephone can send and receive simultaneously. However, most human beings possess a half-duplex audio system (they seemingly can't speak and listen at the same time), so the separation into transmission and reception is logical.

A telephone changes state by a combination of electrical messages down the phone cable and by user actions. From the point of view of a hypothetical microcontroller in the telephone, these might all be considered *signals*.

Line ring	ring signal from another phone
Line idle	no signal on phone line
Pick up	user picks up handset
Mic. speech	user speaks into microphone
Line speech	speech signal from other phone
Hang up	user replaces handset

It is now necessary to define which signals cause *transitions* between states; for example, to change state from *idle* to *ringing*, a *ring signal* is required.

It is traditional to document these state changes using a *state diagram* such as Figure 1.6, which is a form of flowchart with special symbols. Each circle represents a defined state, and the arrows between circles are the state transitions, labeled with the signal that causes the transition. So *line speech* causes a transition from the *connected* state to the *receiving* state, and *line idle* causes the transition back to *connected*.

Because of the inherent limitations of the drawing method, these diagrams tend to oversimplify the state transitions; for example, Figure 1.6 doesn't show a state change if the user hangs up while receiving.

A more rigorous approach is to list all the states as rows of a table and all the signals as columns (Table 1.3). The table entries give a new state or are blank if there is no change of state.

Figure 1.6 Telephone state diagram.

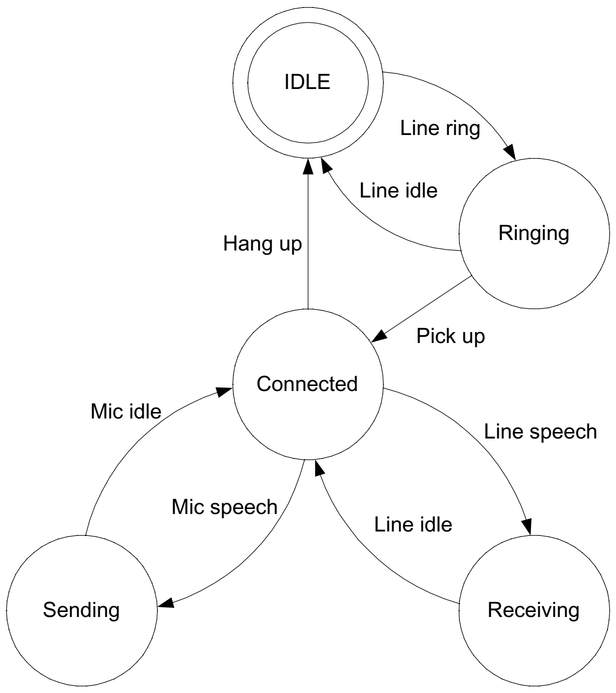


Table 1.3 Telephone state table.

	Line Ring	Line idle	Pick up	Mic. speech	Mic. idle	Line speech	Hang up
Idle	Ringing						
Ringing		Idle	Connected				Idle
Connected				Sending		Receiving	Idle
Sending					Connected		Idle
Receiving		Connected					Idle

Once the table has been created, it isn't difficult to generate the corresponding code. You could use a two-dimensional lookup table, although a series of conditional statements are generally more appropriate.

```
switch(state)
{
    case STATE_IDLE:
        if (signal == SIG_LINE_RING)
            newstate(STATE_RINGING);
        break;
    case STATE_RINGING:
        if (signal == SIG_PICKUP)
            newstate(STATE_CONNECTED);
        else if (signal == SIG_LINE_IDLE)
            newstate(STATE_IDLE);
        break;

    case STATE_CONNECTED:
        // ..and so on
}
```

I have created an *explicit state machine* where the states, signals, and relationship between them are clearly and explicitly identified. Contrast this with an *implicit* state machine, where the current state is buried in function calls.

```
void idle(void)
{
    while (1)
    {
        if (signal == SIG_LINE_RING)
            ringing();
    }
}
void ringing(void)
{
    while (signal != SIG_HANGUP)
    {
        if (signal == SIG_PICKUP)
            connected();
    }
}
void connected(void)
{
    // ... and so on
```

Here, the current state is indicated *implicitly* by the current position in the code, and it is far harder to keep control of all the possible state transitions, particularly under error conditions. The stack-based call return mechanism imposes a hierarchical structure that is ill suited to the arbitrary state transitions required. It is important that the state machine is explicitly created, rather than being an accidental by-product of the way the software has been structured. The requirements of the state machine must dictate the software structure, not (as is often the case) the other way around.

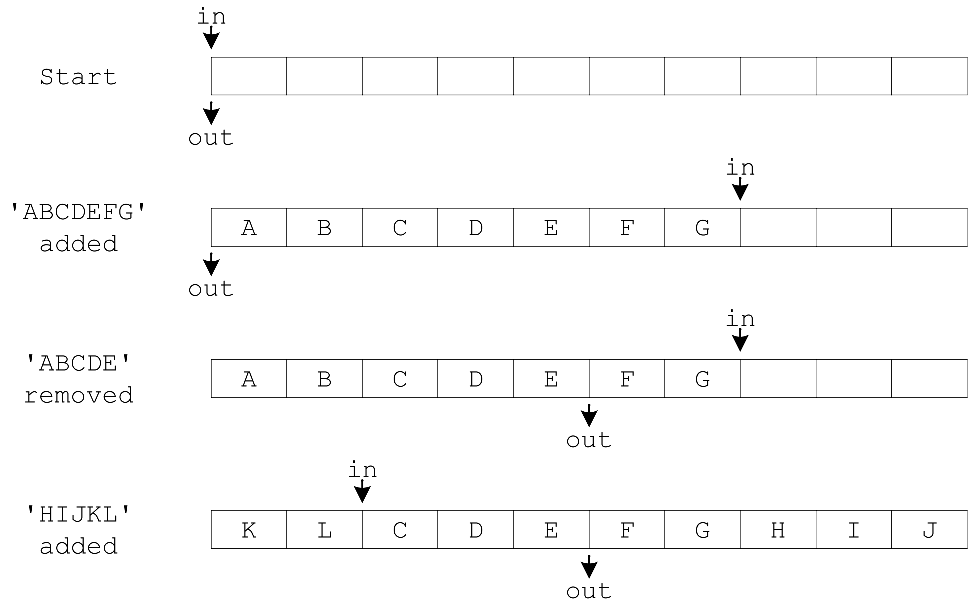
Buffering

To support the protocols, three special buffer types will be used. The first is a modified version of the standard first in, first out (FIFO) to accommodate an extra trial pointer; the second is a fixed-data-length variant of this, and the third is a FIFO specifically designed for bit-wide, rather than byte-wide, transfers.

FITO Buffer

The FITO (first in, trial out) is a variant of the standard FIFO, or circular buffer (Figure 1.7). A normal FIFO has one input and one output pointer; data is added to the buffer using the input pointer and removed using the output pointer. For example, assume that a 10-character FIFO has the letters “ABCDEFG” added, then “ABCDE” removed, then “HIJKL” added.

Figure 1.7 FIFO example.



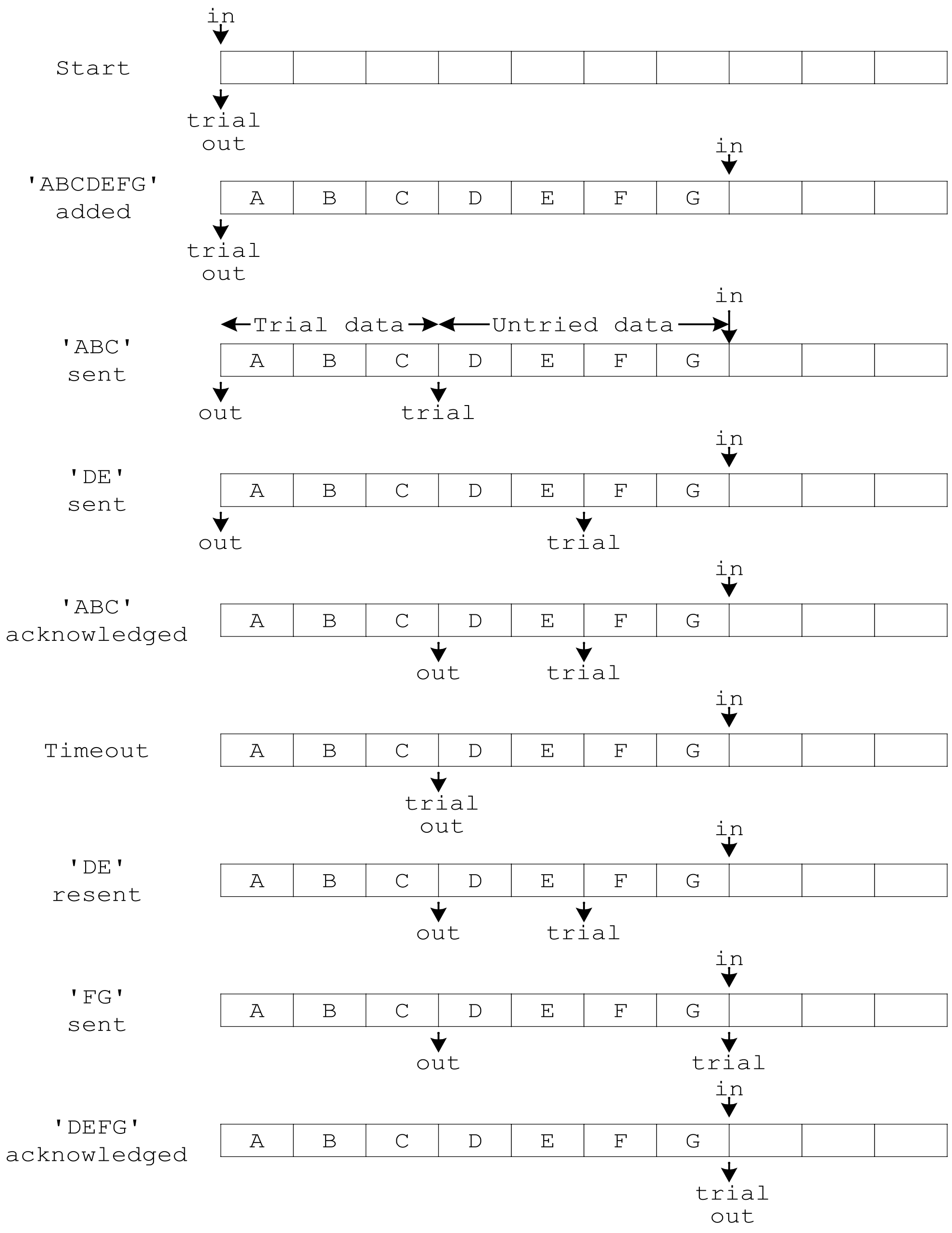
The circularity of the buffer is demonstrated in Figure 1.7 by the second addition; instead of running off the end, the input pointer wraps around to the start, providing there is sufficient space (i.e., the pointers do not collide). Note that after removal, the characters “ABCDE” are shown as still present in the buffer; only the output pointer has changed position. This reflects standard practice, in that there is little point in clearing out unused locations, so the old characters remain until overwritten.

Now imagine this FIFO is being used in a Web server; the input text is a Web page stored on disk, and the output is being transmitted on the network. Due to network unreliability, you don’t actually know whether the transmitted data has been received or has been lost in transit. If the latter, then the data will have to be retransmitted, but it is no longer in the FIFO, so it must be refetched from disk.

It would be better if the FIFO had the ability to retain transmitted data until an acknowledgment was received; that is, it keeps a marker for output data that may still be needed, which I will call *trial* data, in contrast to *untried* data, which is data in the buffer that hasn’t been transmitted yet; hence, the FITO buffer has one input and two output pointers, as shown in Figure 1.8.

Having loaded “ABCDEFG” in the buffer, data fragments “ABC” and “DE” are sent out on the network, and the trial pointer is moved up to mark the end of the trail data. “ABC” is then acknowledged, so the output pointer can be moved up, but the rest of the data is not, so the unacknowledged data between the output and trial pointers is retransmitted on the network, followed by the remaining untried data. Finally that is all acknowledged, so the output pointer can be moved up to join the input pointer.

Figure 1.8 FITO example.



A structure stores the data and its pointers (as index values into the data array). The first word indicates the buffer length, which allows for a variety of buffer sizes. For speed, the buffer size is constrained to be a power of two.

```
#ifndef _CBUFFLEN_
#define _CBUFFLEN_ 0x800
#endif
/* Circular buffer structure */
typedef struct
{
    WORD len;                   /* Length of data (must be first) */
    LWORD in;                   /* Incoming data */
    LWORD out;                  /* Outgoing data */
    LWORD trial;                /* Outgoing data 'on trial' */
    BYTE data[_CBUFFLEN_];      /* Buffer */
} CBUFF;
```

A default buffer size of 2Kb is provided, which may be overridden if required. This permits a buffer to be declared as a simple static structure.

```
#include "netutil.h"
CBUFF rxpkts = {_CBUFFLEN_};
```

Or, consider the code when using dynamically allocated memory.

```
#define BUFFLEN 0x2000
CBUFF *rxp;
if ((rxp = (CBUFF *)malloc(BUFFLEN+16))!=0)
{
    rxp.len = BUFFLEN;
    ...
}
```

In both cases, the length value is set when the buffer is created; this is very important if strange bugs are to be avoided.

The use of `LWORD` (unsigned 32-bit) buffer pointers with `WORD` (unsigned 16-bit) data length may seem strange. The former is part of a Cunning Plan to map the TCP 32-bit sequencing values directly onto these pointers, whereas the latter permits the code to be compiled into a 16-bit memory space (e.g., small model), if necessary. All should become clear in subsequent chapters.

In creating the buffer-handling software, it is important to retain a clear idea of what is meant by *untried* data (not yet sent), and *trial* data (sent but not acknowledged).

```
/* Return total length of data in buffer */
WORD buff_dlen(CBUFF *bp)
{
    return((WORD)((bp->in - bp->out) & (bp->len - 1)));
}
/* Return length of untried (i.e. unsent) data in buffer */
WORD buff_untriedlen(CBUFF *bp)
{
    return((WORD)((bp->in - bp->trial) & (bp->len - 1)));
}
/* Return length of trial data in buffer (i.e. data sent but unacked) */
WORD buff_trylen(CBUFF *bp)
{
    return((WORD)((bp->trial - bp->out) & (bp->len - 1)));
}
/* Return length of free space in buffer */
WORD buff_freelen(CBUFF *bp)
{
    return(bp->len ? bp->len - 1 - buff_dlen(bp) : 0);
}

/* Set all the buffer pointers to a starting value */
void buff_setall(CBUFF *bp, LWORD start)
{
    bp->out = bp->in = bp->trial = start;
}
```

When loading data into the buffer, the simple but slow method is to copy it byte-by-byte. Instead, I'll use either one or two calls to a fast block-copy function, depending on whether the new data wraps around the end of the buffer. If the data is too big for the buffer, it is truncated, because I'm assuming the programmer has checked the free space before calling this function. The free space is always reported as one byte less than the actual space, so there is no danger of the input pointer catching up with the output pointer.

```
/* Load data into buffer, return num of bytes that could be accepted
** If data pointer is null, adjust pointers but don't transfer data */
WORD buff_in(CBUFF *bp, BYTE *data, WORD len)
{
    WORD in, n, n1, n2;
    in = (WORD)bp->in & (bp->len-1);        /* Mask I/P ptr to buffer area */
    n = minw(len, buff_freelen(bp));        /* Get max allowable length */
    n1 = minw(n, (WORD)(bp->len - in));     /* Length up to end of buff */
    n2 = n - n1;                            /* Length from start of buff */
    if (n1 && data)                         /* If anything to copy.. */
        memcpy(&bp->data[in], data, n1);    /* ..copy up to end of buffer.. */
    if (n2 && data)                         /* ..and maybe also.. */
        memcpy(bp->data, &data[n1], n2);    /* ..copy into start of buffer */
    bp->in += n;                            /* Bump I/P pointer */
    return(n);
}

/* Load string into buffer, return num of chars that could be accepted */
WORD buff_instr(CBUFF *bp, char *str)
{
    return(buff_in(bp, (BYTE *)str, (WORD)strlen(str)));
}
```

Removal of untried data from the buffer, so that it becomes trial data, is essentially the inverse of the above.

```
/* Remove waiting data from buffer, return number of bytes */
** If data pointer is null, adjust pointers but don't transfer data */
WORD buff_try(CBUFF *bp, BYTE *data, WORD maxlen)
{
    WORD trial, n, n1, n2;

    trial = (WORD)bp->trial & (bp->len-1);  /* Mask trial ptr to buffer area */
    n = minw(maxlen, buff_untriedlen(bp));  /* Get max allowable length */
    n1 = minw(n, (WORD)(bp->len - trial));  /* Length up to end of buff */
    n2 = n - n1;                            /* Length from start of buff */
    if (n1 && data)                         /* If anything to copy.. */
        memcpy(data, &bp->data[trial], n1); /* ..copy up to end of buffer.. */
    if (n2 && data)                         /* ..and maybe also.. */
        memcpy(&data[n1], bp->data, n2);    /* ..copy from start of buffer */
    bp->trial += n;                         /* Bump trial pointer */
    return(n);
}
```

Functions to remove data from the buffer are required to complete the set and to wind back the trial pointer so that the data is waiting for retransmission.

```
/* Remove data from buffer, return number of bytes
** If data pointer is null, adjust pointers but don't transfer data */
WORD buff_out(CBUFF *bp, BYTE *data, WORD maxlen)
{
    WORD out, n, n1, n2;
    out = (WORD)bp->out & (bp->len-1);      /* Mask O/P ptr to buffer area */
    n = minw(maxlen, buff_dlen(bp));        /* Get max allowable length */
    n1 = minw(n, (WORD)(bp->len - out));    /* Length up to end of buff */
    n2 = n - n1;                            /* Length from start of buff */
    if (n1 && data)                         /* If anything to copy.. */
        memcpy(data, &bp->data[out], n1);   /* ..copy up to end of buffer.. */
    if (n2 && data)                         /* ..and maybe also.. */
        memcpy(&data[n1], bp->data, n2);    /* ..copy from start of buffer */
    bp->out += n;                           /* Bump O/P pointer */
    if (buff_untriedlen(bp) > buff_dlen(bp))/* ..and maybe trial pointer */
        bp->trial = bp->out;
    return(n);
}
/* Rewind the trial pointer by the given byte count, return actual count */
WORD buff_retry(CBUFF *bp, WORD len)
{
    len = minw(len, buff_trylen(bp));
    bp->trial -= len;
    return(len);
}
```

As a useful extra feature, a null data pointer can be given to the function, in which case it goes through the same motions, but without copying any actual data. This is handy for discarding unwanted data (e.g., trial data that has been acknowledged).

I've made extensive use of `minw()`, which returns the lower of two word values and so is similar to the standard function `min()`.

```
WORD minw(WORD a, WORD b)
{
    return(a<b ? a : b);
}
```

Why define my own? Because `min()` is usually implemented as a macro,

```
#define min(a, b) (a<b ? a : b)
```

and any function arguments may be executed twice, which is a major problem in interrupt-driven (reentrant) code. Take a line from `buff_out()`.

```
n = minw(maxlen, buff_dlen(bp));        /* Get max allowable length */
```

The macro expands this to the following.

```
n = maxlen < buff_dlen(bp) ? maxlen : buff_dlen(bp);
```

Imagine that the first time `buff_dlen()` is executed, the source buffer is almost empty, so all its data can be transferred into the destination. However, before the function is executed a second time, an interrupt occurs that fills the buffer with data, so the actual data length copied exceeds the maximum the destination can accept, with disastrous results. The easiest way to avoid this problem is to buffer the comparison values in a function's local variables; hence, the usage of `minw()`.

Polygonal Buffer

A circular buffer is useful for handling unterminated streams of data, but sometimes you'll need to store blocks of known length. The classic case is a packet buffer, in which you can queue packets prior to transmission or on reception. The standard technique is to have a *buffer pool*, from which the storage for individual packets can be allocated. A simpler technique is to use a circular buffer as before but to prefix each addition to it with a length word, to show how much data is being added.

```
    if (len>0 && buff_freelen(&rxpkts) >= len+2)/* If space in circ buffer..*/
    {
        buff_in(&rxpkts, (BYTE *)&len, 2);      /* Store data len.. */
        buff_in(&rxpkts, buff, len);            /* ..and data */
    }
```

The smooth circle of data has been replaced by indivisible straight-line segments; when recovering the data, check that the whole block is available (if there is a risk that part of the block may be in transit). The trial system comes in handy because you can retry (i.e., push back) the length if the entire data block isn't available yet.

```
    if ((dlen=buff_dlen(&txpkts)) >= 2)
    {
        buff_try(&txpkts, (BYTE *)&len, 2); /* Get length */
        if (dlen >= len+2)                  /* If all there.. */
        {
            buff_out(&txpkts, 0, 2);        /* ..remove len */
            buff_out(&txpkts, buff, len);   /* ..and data */
        }
        else
            buff_retry(&txpkts, 2);         /* Else push back len */
    }
```

This explains the length parameter on the front of the generic frame. It allows you to store and retrieve GENFRAME structures from circular buffers without having to understand the contents of the frame.

Coding Conventions

It isn't essential that you use the same coding conventions (source code formatting) as I do, though it may help if I describe the rules I've used, so you can choose whether to follow them or not.

Data Types

When defining protocol structures, it is really important to use the correct data width. You may be used to assuming that an `int` is 16 bits wide, but that isn't true in a 32-bit system. I've made the following assumptions for all the DOS compilers.

- `char` is an 8-bit signed value
- `short` is 16 bits
- `long` is 32 bits

From these, I have derived the following width-specific definitions.

```
#define BYTE unsigned char
#define WORD unsigned short
#define LWORD unsigned long
```

I have used `#define` in preference to `typedef` because compilers use better optimization strategies for their native data types. A notable omission is a Boolean (`TRUE`/`FALSE`) data type; I use an integer value and assume `TRUE` is any non-zero value.

Keeping compatibility with both 16- and 32-bit compilers also necessitates the addition of some redundant-looking typecasts.

```
WORD a, b;
a = (WORD)(b + 1);
```

If the typecast is omitted, the Visual C++ compiler issues a warning message because `b` is promoted to a 32-bit value for the addition, which must be truncated when assigned to `a`.

Another tendency of 32-bit compilers is, by default, to pad data elements out to four- or eight-byte boundaries, which blows gaping holes in the structures.

```
typedef struct {
    BYTE a;
    BYTE b;
    LWORD c;
    WORD d;
} MYSTRUCT;
```

If the mix of bytes, words, and long words is to be transmitted on the network, it is vital that the compiler is set so that it does not introduce any padding between these structure elements; that is, the structure member alignment is one byte, not four or eight bytes.

All the necessary configuration data is included in the appropriate compiler-specific project file, so should be loaded automatically. The PICmicro cross-compiler is very different from all the others; its peculiarities are explored in Chapter 9.

Source Code Format

I have attempted to illustrate all the techniques in this book by embedding source code fragments in the text. Because the full source code is included on the accompanying CD and you won't be retyping the code from the book, I have used some artistic license in the preparation of these fragments by stripping out all nonessential stuff so the essence of the code is clear to see.

You will see frequent ellipses (...) indicating deletion of the lines I think are irrelevant to the topic under discussion. If the absence of these items hinders your comprehension of the text, I suggest you print out a copy of the latest source code and use that in conjunction with this book.

Also, the version printed in the book is not necessarily the latest version. All released versions of the software have revision headers, containing entries as follows.

```
/*
** v0.01 JPB 1/1/00  First version
** v0.02 JPB 2/1/00  Added widget-handling option
*/
```

The version number of the complete utility is taken from the version number of the file bearing its name. For example, if `test.exe` is built from `utils.c` v0.12, `test.c` v1.23, and `driver.c` v2.34, then `MYTEST` has the overall version number v1.23. To confirm the build status, each project has a CRC file containing the file names, version numbers, and a 32-bit CRC for each file.

```
Project PING v0.17 archived 24-Mar-00 11:49

ether3c.c    6376B05D v0.05
etherne.c    9759AE0A v0.24
ip.c         C3BB52D4 v0.10
net.c        1EB2A188 v0.07
netutil.c    73320977 v0.11
ping.c       B66F030C v0.17
pktd.c       183DCB73 v0.09
serpc.c      DD6477B8 v1.08

ether.h      4FF45517
ip.h         C739761D
net.h        C19A2BBD
netutil.h    B8B96B7F
pktd.h       0ACBA7F6
serpc.h      089DEAF1
```

The Iosoft Ltd. Web site (`www.iosoft.co.uk`) has up-to-date versions of the software in this book and useful extra information, such as application notes.

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

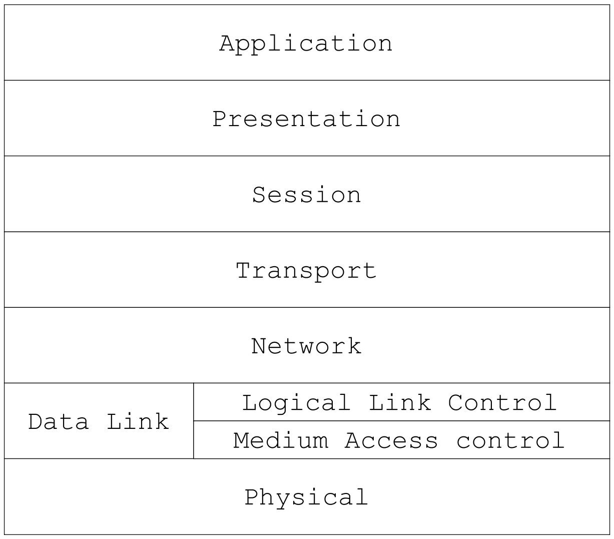
The international community, in its wisdom, decided to standardize on the number of protocol layers in a stack, and the International Standards Organization (ISO) Open Systems Interconnection (OSI) model was born (Figure 2.1). Their layers are listed below from top down.

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.

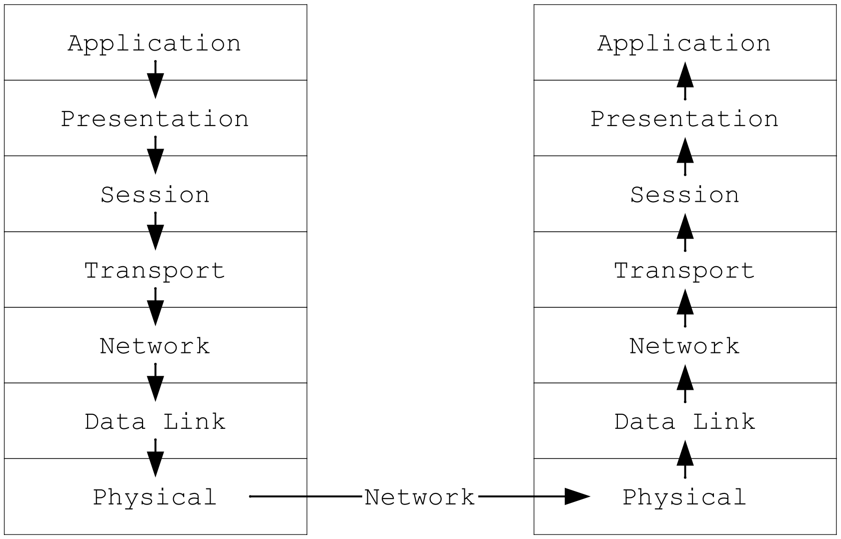


When two applications are communicating over a network, it can be useful to think in terms of the data entering at the top of one protocol stack then traveling downward on that machine, across to the other machine at the physical layer, and back up the second stack (Figure 2.2).

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.

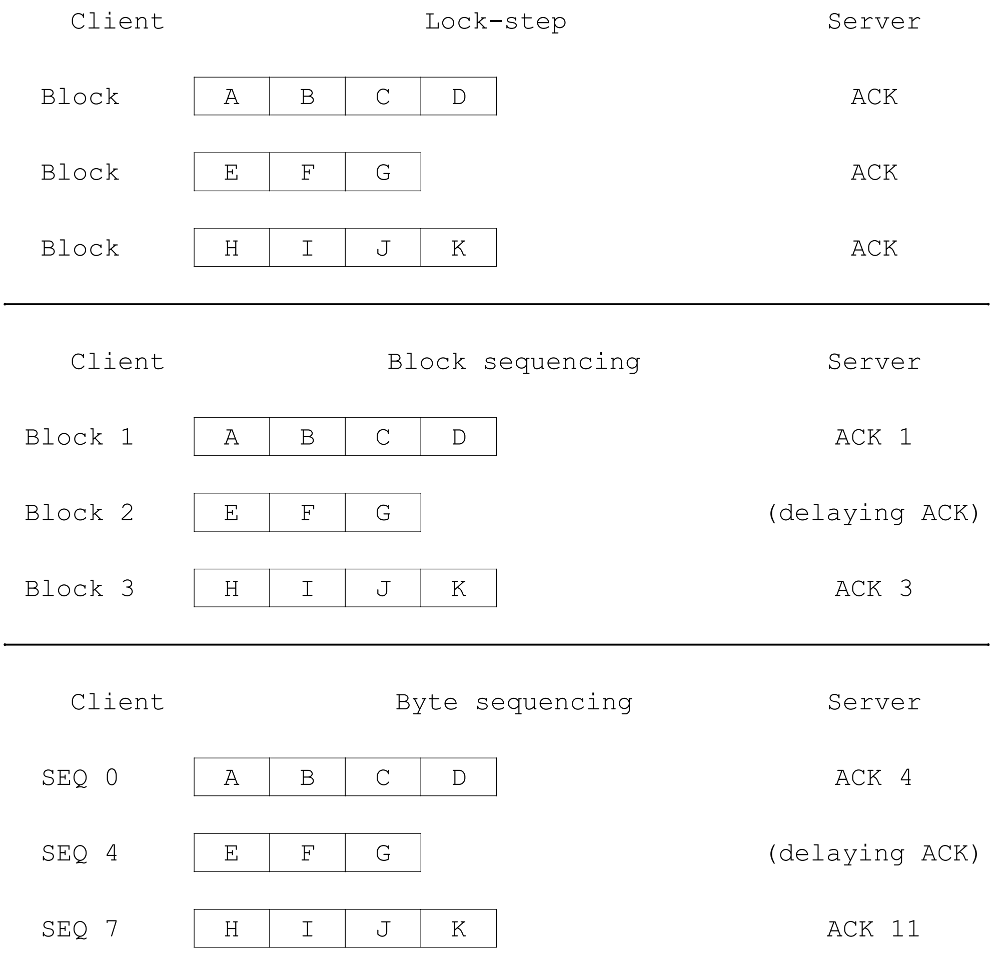


Figure 2.3 shows the client–server interactions, assuming the client is sending 11 bytes in three blocks to the server. The lock-step method doesn't need to identify each block individually, since only one can be in transit at any one time (in railway parlance: one engine in steam). The sequencing methods 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.

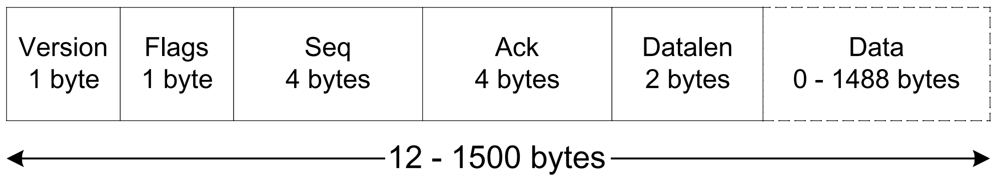
There is little to choose between the two sequencing methods. Block acknowledgment is used in the ISO link layer LLC and is slightly easier to implement than byte sequencing, provided a fixed block size is used. TCP has a variable block size (a "sliding window"), so it employs a byte-sequencing method.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

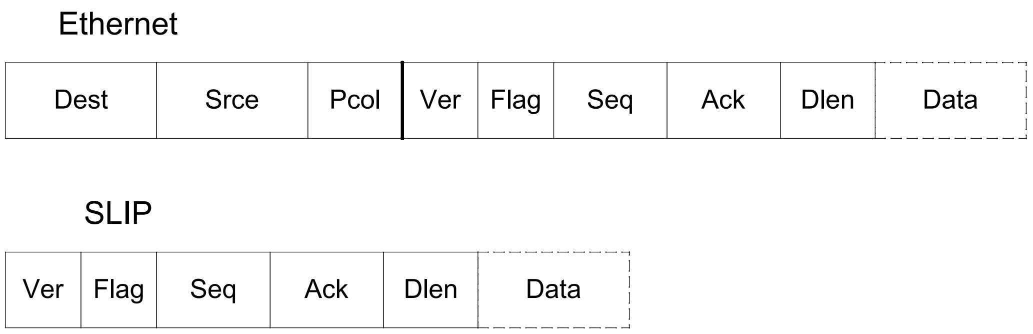
Internal Storage

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.

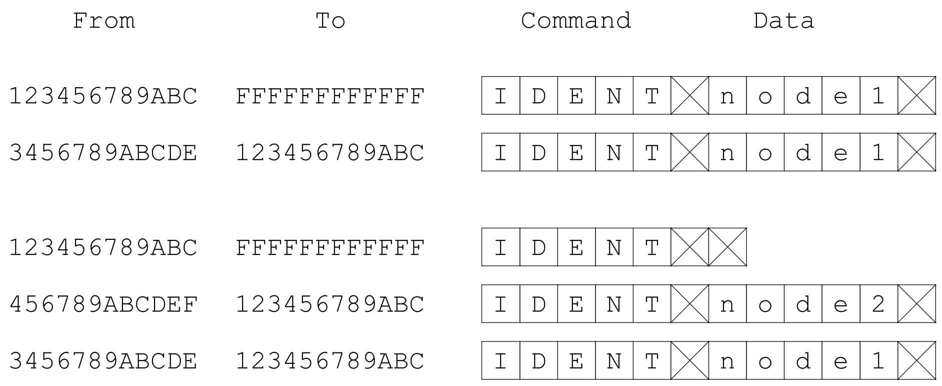

From
To
Command
Data
123456789ABC FFFFFFFFFFFF I D E N T n o d e 1
3456789ABCDE 123456789ABC I D E N T n o d e 1
123456789ABC FFFFFFFFFFFF I D E N T
456789ABCDEF 123456789ABC I D E N T n o d e 2
3456789ABCDE 123456789ABC I D E N T n o d e 1

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.

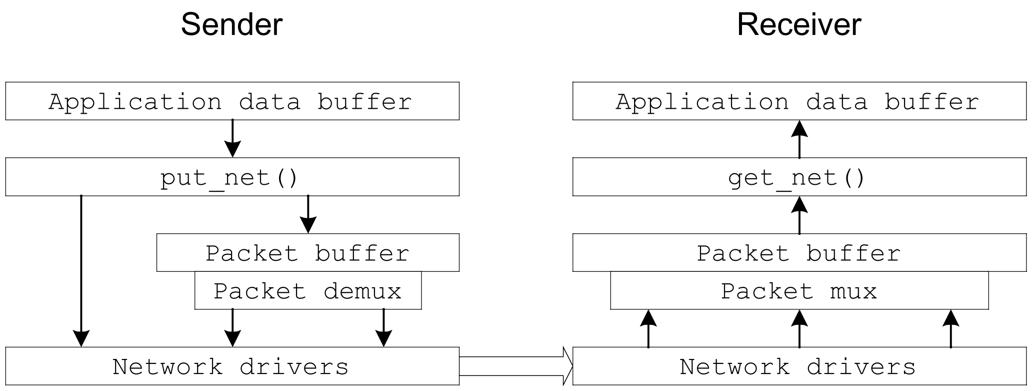
Protocol Identification

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.



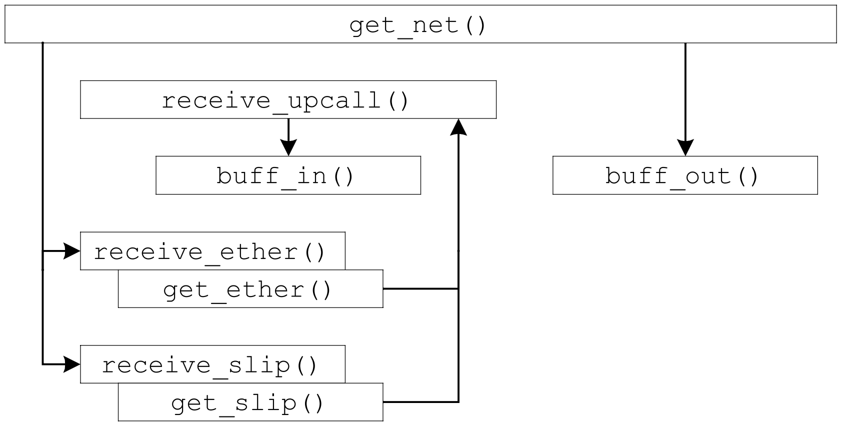
The mux/demux operation (Figure 2.7) is automatically performed by the network driver layer. Submitting a packet to `put_net()` automatically routes it to the appropriate network driver, possibly via a (polygonal, as described in the previous chapter) packet buffer, if the interface doesn't have its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

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 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 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 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.

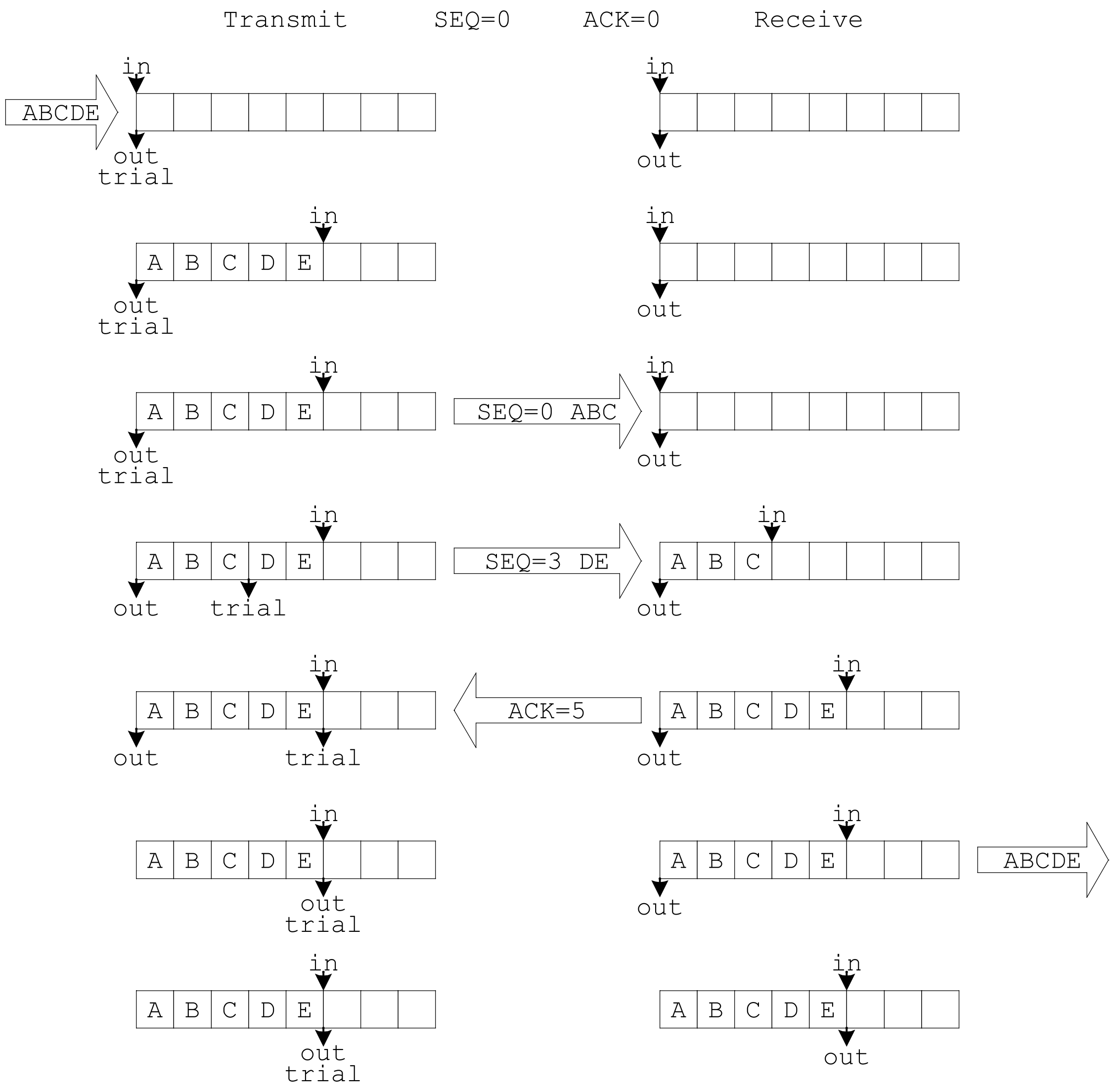
To address the first point, imagine that both nodes have circular buffers of data, and you are simply trying to keep the circular buffer pointers in sync. The circular buffer pointers have 32-bit values (even though the buffer size doesn't warrant it) to allow a simple mapping onto the sequence and acknowledgment 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

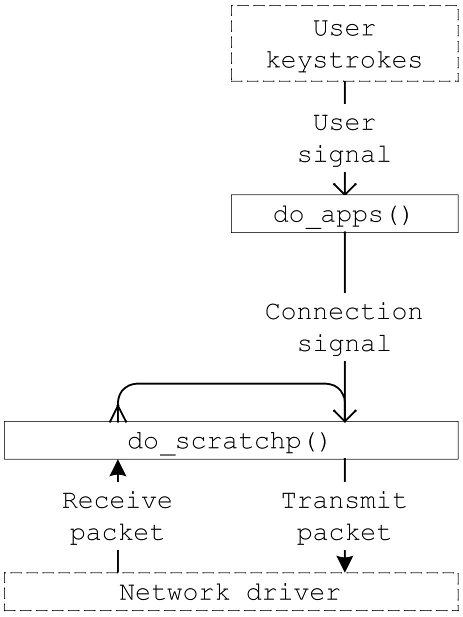
Now that all the hard work of creating, maintaining, and destroying connections is done, there is the relatively simple job of creating application code for

- `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(consig);
}
```

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	