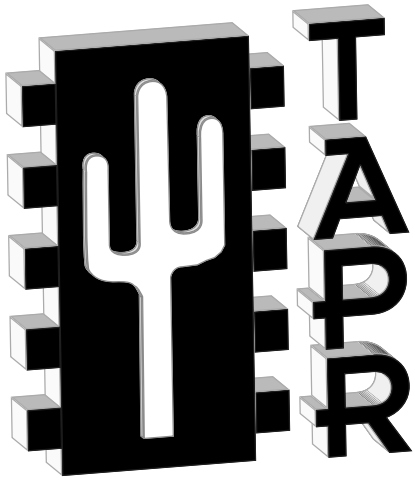
Networking Without Wires: Radio Based TCP/IP

By: John Ackermann, N8UR
n8ur@tapr.org



Published by: Tucson Amateur Packet Radio Corporation

ISBN: 0-9644707-2-1

First Edition
First Printing, 1999

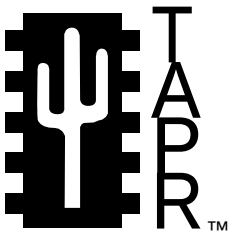
TAPR Production Editor: Greg Jones, WD5IVD

Portions of this book first appeared electronically as "IntroNOS: Getting Started with TCP/IP on Packet Radio." John Ackermann, N8UR. 1992

Acknowledgments

Thanks to the following folks; though some of them many not know it, they made this book better.

Orv Beach, WB6WEY; Mike Bilow, N1BEE; James Dugal, N5KNX; Paul Evans, W4/G4BKI; Bdale Garbee, N3EUA; Gary Grebus, K8LT; Gary Hauge, N4CHV; Greg Jones, WD5IVD; Brian Kantor, WB6CYT; Phil Karn, KA9Q; Brian Lantz, KO4KS; Liam McCann, G7TVC; Fred Peerenboom, KE8TQ, Steve Stroh, N8GNJ, Terry Dawson VK2KTJ, Tomi Manninen, OH2BNS, James Fuller, N7VMR, Jonathan Naylor, G4KLX, Carl Makin, VK1KCM, and Barry McLarnon, VE3JF.



Tucson Amateur Packet Radio
8987-309 E. Tanque Verde Rd #337
Tucson, Arizona • 85749-9399
Office: (940) 383-0000 • Fax: (940) 566-2544
Internet: tapr@tapr.org • www.tapr.org
Non-Profit Research and Development Corporation

Contents

TAPR on the Internet

World-Wide-Web!

TAPR's Home Page can be reached at: http://www.tapr.org

FTP

The TAPR Software Library is available via anonymous FTP. You can access the library by ftp to 'ftp.tapr.org'. Login in as 'anonymous', with a password of 'your_account@internet_address'. In addition, 'ftp.hereford.ampr.org' also has the library available.

TAPR via Listserv

TAPR can also be reached via its automated information server. Anyone can request information on TAPR, products, newsletters, and lots of other files, by sending an e-mail message to `listserv@tapr.org` with the subject line "`Request`" and the following text in the body of the message:

```
help                        (for a brief set of instructions)
index -all                  (for a list of all file by topic area)
get tapr taprinfo.txt               (or info on TAPR)
```

In the above message example, "`help`" retrieves a brief set of instructions for info, "`index -all`" retrieves a list of available files by topic and "`taprinfo.txt`" retrieves a text file containing information on TAPR and what TAPR is all about. If you want to retrieve several text files with one message, use a separate line for each "get filename" request.

TAPR Special Interest Groups

TAPR maintains several SIGs in various areas of interest (Spread Spectrum, DSP, HF Digital, APRS, etc). Send an e-mail message to `listserv@tapr.org` with the subject line "`Request`" and the following text in the body of the message:

```
lists                       (for list of mail groups on TAPR.ORG)

information listname        (where listname is the name of the mail group)
```

When you are ready to subscribe, send e-mail to `listserv@tapr.org` with the following command in the message text:

```
subscribe list YourFirstName YourLastName Callsign
```

The TAPR web page also provides an interface for subscribing to mail lists.

Foreword

Digital transmissions are rapidly supplanting analog transmissions in both amateur and commercial radio communications. Digital radio transmissions offer many inherent advantages. Digital techniques allow virtually noise-free transmission of data, messages, control commands, sounds and images to be achieved. Digital transmissions allow information to flow from multiple sources to multiple destinations on the same radio channel in an organized and efficient way. Information is communicated in a form that is easy to store and access. And when needed, information security can be readily added to a digital transmission.

Amateur Radio can be proud of its pioneering contributions to digital radio communications techniques, and especially the Tucson Amateur Packet Radio Corporation (TAPR), which developed the first widely used terminal node controllers for amateur packet radio.

I believe one of the most important contributions TAPR has made to digital radio communications recently is the publishing of this book. Tom McDermott, who has an extensive professional background in digital telecommunications, covers the topic of digital radio transmissions in a remarkably lucid and accessible way. Tom has held the complex mathematics typically found in most books on this subject to a minimum. Instead, Tom uses an easy to understand graphical approach to explain digital communications concepts. Tom has also included a collection of Excel™ spreadsheets with this book that allows the reader to explore a number of digital communications principles.

Amateur radio enthusiasts, engineering students and radio communications professionals will find this book both an excellent introduction to the digital radio communications and a useful day-to-day reference.

Frank H. Perkins, Jr. WB5IPM
TAPR/AMSAT DSP-93 Development Team Member

Excel™ is a trademark of Microsoft Corporation.

Introduction

This book describes how ham radio operators can harness the power of the world's most popular networking protocol, TCP/IP, to build better — and more useful — packet radio networks

Networking Without Wires is an introduction to TCP/IP operation over ham radio. It's intended to help hams with some experience in packet radio get started with NOS—the TCP/IP software written by Phil Karn, KA9Q, and others. It is not intended to take the place of an in-depth NOS reference manual, but rather to provide an understanding of what TCP/IP is, and how hams can use it to improve our packet radio operations. Additionally, in the couple of years it's become practical to use tools other than NOS to put TCP/IP on the air, and I will talk about those options as well. In particular, I'll describe how to use the built-in AX.25 capabilities of the Linux operating system to put the full Linux TCP/IP suite on the air.

I had three goals in writing this book, and its parts reflect them. Part 1 describes how TCP/IP works generally, and concentrates on IP addressing and routing in particular because these two areas are the meat of the TCP/IP scheme. Parts 2 and 3 show how to get NOS running on your computer—how to configure and use its basic features, including the electronic mail subsystem. Finally, Part 4 describes some of the more advanced features of NOS and provides basic information on how to set up NOS PBBS systems, NET/ROM nodes, "Convers" servers, and Internet gateways. It also provides details about configuring and using Linux for amateur radio TCP/IP. Although I've tried to address the common questions that come up when configuring and running NOS, much has necessarily been left unsaid. Consider this book to be a starting point, not the end of the road.

It will be very helpful if you have the NOS reference manual handy when you read this book, and especially when you start configuring and using NOS. There are some subtle variations in command syntax from one version of the program to another, but with the reference manual and a bit of trial and error you'll have things working in no time.

NOS Versions

Although Phil Karn is still doing development on NOS, several others have taken his work and added to it. NOS is a growing and changing creature with several quite different versions, some of which are being updated rapidly. I'll use the term "xNOS" to refer generically to the family of NOS derivatives. I can't tell you precisely where to find the latest and greatest version, but Appendix A includes a partial list of places you can find the various flavors of NOS and supporting programs. It also lists a number of NOS resources, such as mailing lists and users groups.

The NOS variant I use on my home PC, and which is the basis for most of the examples in this book, is called **JNOS**. The version used for the examples in this book is 1.11M, which was current in January, 1998. JNOS was developed by Johan Reinalda, WG7J[1-1] from the original work done by KA9Q and PA0GRI's further development in the early 1990s, and it's probably the most popular variant of NOS in use today. The biggest thing that sets JNOS apart from earlier NOS versions is its full-blown packet BBS (PBBS) functionality. It also supports a sophisticated "converse" bridge that provides a conferencing or "chat" facility between users worldwide via both radio and Internet links.

[1-1] *Johan has moved to Japan and is no longer actively working on JNOS. James Dugal, N5KNX, has taken over development and he periodically makes new releases available.*

Another version of NOS that is gaining in popularity is **TNOS**, developed and maintained by Brian Lantz, KO4KS. TNOS is based on JNOS, but adds additional features, including much more sophisticated BBS functionality. One of the interesting things about TNOS is that although it will run under MS-DOS, it is primarily intended for use under UNIX operating systems like **Linux,** a freeware UNIX clone. Running TNOS under UNIX eliminates the memory bottleneck that is the biggest problem with packing lots of functionality into a DOS-based NOS. The PBBS that I help run has switched to the TNOS/Linux combination, and I use TNOS version 2.30 as the reference for the discussion of PBBS operation in Chapter 18. For the most part, JNOS and TNOS use compatible commands; where they differ I'll try point that out.

A last note before plunging in—I said it before, and I'll say it again: this book barely scratches the surface of NOS and TCP/IP. Nearly every command described here has options or parameters that I may not mention. My goal is to give you a feel for what TCP/IP does, and to get you on the air with NOS. To get beyond the novice stage you need to look at the reference manual and experiment with the software. You'll find a local TCP/IP Elmer to be an invaluable asset; this stuff is much more fun when there's someone else to talk to, and learn with.

Chapter

1

TCP/IP and Amateur Radio

TCP/IP is not a program, or even a single concept. It's a set of communication **protocols** that have become almost universal standards in the computer networking world. A "protocol" is simply a standard that describes the way two things — in this case, computer systems — communicate with one another. A protocol describes things like the data format, the way systems identify themselves, the messages they send, and how they respond to those messages.

Some of the TCP/IP protocols are quite complex, while others are simple. Together, they specify how different types of computers can exchange information with one another over different kinds of networks. That's one of the reasons for TCP/IP's power and popularity: it allows **internetworking** across the whole spectrum of hardware and software environments. TCP/IP provides both system-level protocols — the standards that let computers actually talk to one another — and user applications that allow for file downloading, mail transfer, remote computer login, and other useful services.

There are lots of TCP/IP software implementations; it runs on nearly every kind of computer available, from IBM mainframes and UNIX systems to PCs, Macs, Amigas, and Ataris. Most PC operating systems, like Windows NT, Windows 95/98, and OS/2

Warp Connect, include built-in TCP/IP support (though that support is missing some important elements for ham radio work). For hams interested in TCP/IP over packet radio, there's one program in particular that's important.

NOS (short for **N**etwork **O**perating **S**ystem) is special because Phil Karn, KA9Q, wrote it specifically for ham radio use. Apart from duplicating all the functionality of a complete networking system in a single PC program, it includes the special features necessary to make TCP/IP work well over radio channels that are slower and less reliable than an Ethernet. Since Phil first started coding NET—the predecessor of NOS—in 1985, his program has grown tremendously, been ported to other computer types and operating systems, and expanded into environments other than amateur packet radio. KA9Q has always made the source code to NOS available, and as a result many other hams have built upon his work. More than a decade later, one of the many xNOS variants remains the most popular choice if you want to use TCP/IP as part of your ham radio station.

If you've looked at the size of the NOS documentation—or of this book—you're probably asking yourself why it's worth the effort to master this fairly complex stuff. NOS has many features that improve on regular packet radio. It has powerful routing tools that help us build wide-area packet radio networks. It has much more sophisticated electronic mail capabilities than our present PBBS systems. It has a well-proven file transfer protocol. It supports multiple simultaneous connections. It has transport methods that improve the reliability and throughput of slow and congested channels. And, it's expandable: new features and protocols can easily be added. In fact, one of the problems with NOS is that it can include too many features to work in the limited memory space provided by a DOS computer.

The bottom line is that NOS provides the basis for flexible, robust, computer networking over amateur radio.

The TCP/IP protocol suite allows different kinds of computers to talk to one another across different kinds of networks. The services it provides include terminal sessions, file transfer, electronic mail, and data routing. Computers running TCP/IP (referred to as **hosts**) can run some or all of these applications simultaneously; it's possible to sit at a PC running NOS and carry on a keyboard-to-keyboard chat with one station, while another retrieves a file from your hard disk and your computer sends electronic mail to a third. (Figure 1-1)

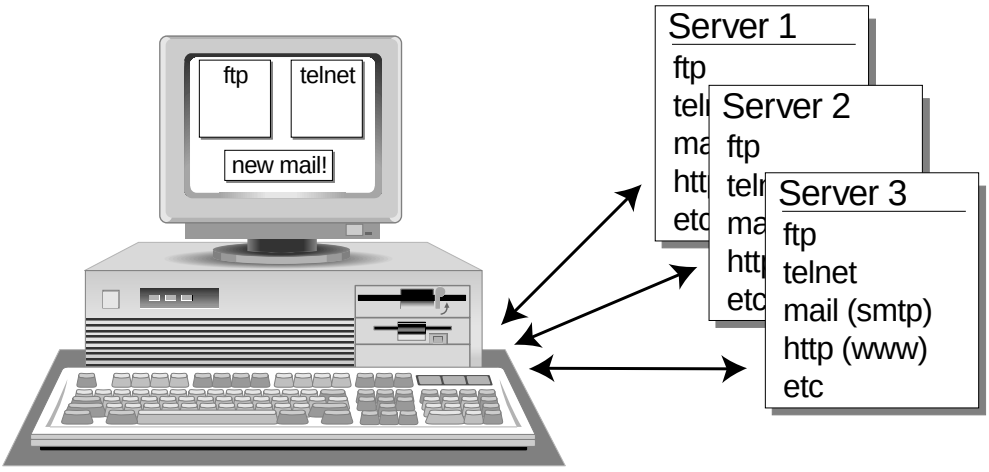


Figure 1.1 - Host running multiple applications

NOS also has the ability to route transmissions to distant stations without the user needing to know every stop along the way; all you need to do is get your data to a **gateway**[1-2] —an intermediate station that knows how to move it one step closer to its destination. NOS can also use protocols that automatically discover routes, and can adjust to changes in the network. Although the tools to do this are still under development, they offer the promise of a network that intelligently figures out the best route from point A to point B, and can deal with propagation and hardware problems along the way. (Figure 1-2)

---

*[1-2] There are two other terms that are often used interchangeably with "gateway": "**router**" and "**switch**". Although all three terms have precise technical meanings, they are often used without regard to those meanings. Throughout this book, I'll use the broad definition and use the terms gateway and switch interchangeably. I won't use router at all; it's a widely used term in the real world, but not in the ham IP community.*

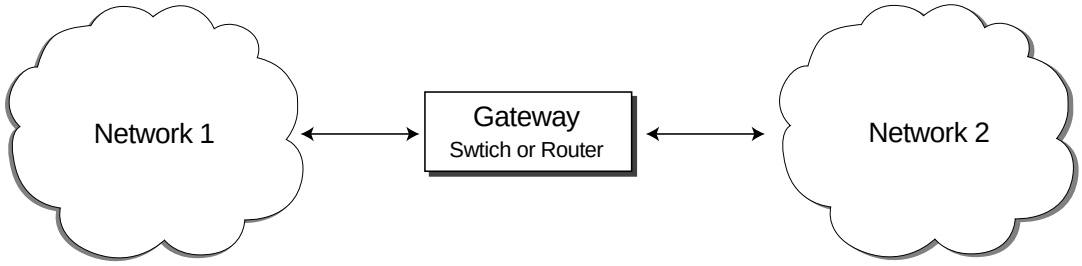


Figure 1-2 - Internetworking using a gateway.

Since NOS is based on the de facto standard system for interconnecting computers, it offers the possibility of sophisticated services far beyond anything available on regular packet radio. For example, in some areas ham TCP/IP users can log into multi-user UNIX computer systems and run applications as if they were directly connected to those machines. Email can be transparently moved between the ham TCP/IP community and the Internet. JNOS and TNOS can even act as Worldwide Web servers!

NOS and Traditional Packet Radio

All these capabilities are impressive, but you may wonder if the cost of TCP/IP is giving up the ability to connect to the local BBS, or check DX spots from a PacketCluster. When you run NOS, you don't give up the ability to carry on normal packet communications. In fact, NOS works very nicely to allow multiple AX.25 connections. You can use it just like a terminal program to establish connections with your local PBBS or PacketCluster, or to chat with friends who don't run NOS (yet) — or do all these things simultaneously (and run TCP/IP applications as well)! You can think of NOS as a general purpose packet radio program, not as something specialized and of limited use. It's the only packet software you need to run.

Chapter

2

What is TCP/IP ?

We've said that TCP/IP is a set of protocols for the transfer of data across networks of dissimilar computers. You can think of these protocols as forming two sets. One group establishes the way two computers talk to each other at a basic level, dealing with things like addresses and routing, and making sure that data actually gets to its destination in the right sequence and without corruption. The other provides more directly useful services that ride on top of the lower-level protocols.

Two of the lower-level protocols give the set its popular name: **TCP** (**T**ransport **C**ontrol **P**rotocol) is a "reliable stream service," which is a fancy way of saying it makes sure that all the data sent to a remote host actually gets there in the correct order; and **IP (I**nternet **P**rotocol) handles the dirty details of addressing and routing data packets over a network.

Frankly, I think we'd have been better off if we'd called it "IP/TCP" instead of the other way around, because it's IP that does the most interesting work — it handles the addressing and routing of packets. TCP's job is important but much less interesting: it makes sure that all the data sent by a higher level protocol arrives at the far end without error and in the proper sequence. Consequently, this part of the book focuses on IP and the other low-level protocols that help IP do its job.

Protocols and Services

We shouldn't get into the nitty-gritty, though, without first describing some of the higher-level protocols that let folks do useful things, because TCP/IP wouldn't be of much use if it didn't support applications that human beings want to use. TCP and IP alone don't do anything that a user can interact with—they're building blocks. It's the higher level protocols in the TCP/IP suite that do useful things. There are lots of these user services, and NOS supports many of them, both as a **client** that uses the service, and as a **server** that makes it available to other hosts. (Figure 2-1)

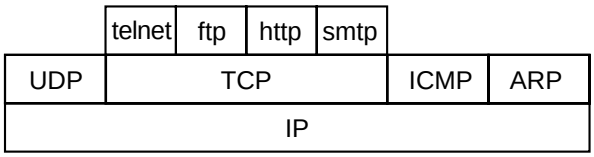


Figure 2-1 - Some TCP/IP Protocols (described in Chapter 2)

Some of these protocols – and the ones we use most on the ham network – are:

TELNET—The terminal emulation program. In "real" networks, telnet[2-1] lets a user at one host remotely access a remote host, just as if he was on a terminal directly connected to that computer. In NOS, the telnet function usually connects you to a remote host's mailbox, which acts very much like a personal PBBS. The NOS telnet command also allows you to remotely login to a host that supports that function; in some areas UNIX computers connected to the ham TCP/IP network allow remote login and allow users to run application programs over the air.

FTP—File **T**ransfer **P**rotocol. A means of transferring both ASCII (text) and binary (program, data, or compressed) files between hosts. NOS systems can be ftp servers, making files available to others for download, as well as ftp clients.

[2-1] *TCP/IP is intimately related to the UNIX world, and UNIX types favor lower-case names for programs and protocols.*

SMTP—Simple **M**ail **T**ransfer **P**rotocol. A (mostly) invisible way of moving electronic mail from one host to another. If you create a message on your computer (using the NOS mailbox or the BM program discussed below), SMTP will automatically attempt to transfer it to the destination computer. SMTP is a protocol that runs in the background — unless you look for it, you won't know that it's there.

POP—Post **O**ffice **P**rotocol. SMTP is neat, but it's really designed to work with hosts that are available full time. Most ham TCP/IP stations aren't. POP is designed for them; it allows incoming mail to be stored at a host that acts as a **mail server**; when you come on the air, your system uses POP to ask the server to send you your mail.

PING—Packet **I**nter**N**et **G**roper. A simple program, usually used as a diagnostic tool, that sends a packet to a specified host using the Internet Control Message Protocol (ICMP). If the host is reachable, it responds with another packet. PING tells you the **round trip time** (**RTT**) — how long it took from the time your packet was sent until you receive the acknowledgment. Ping can also send multiple packets and report back how many of them received a response. This is a good way to test the reliability of a path.

FINGER—A way of finding out information about the users at a host. The finger command can simply list all the users at a host, or spit out information (like the "brag tape" of RTTY days) about a specific user.

DNS—Domain **N**ame **S**ervice. Remembering IP addresses isn't easy. NOS can use a file to map hostnames to IP addresses, but that means you need to update this file whenever hosts are added, removed, or changed on the network. Alternatively, a remote host may agree to serve as a **domain name server** that NOS can query when it needs to know the address of a host. NOS can be configured as a name server as well as a client.

NNTP—Network **N**ews **T**ransport **P**rotocol. This is a way to distribute Usenet articles via TCP/IP. It can also be used to distribute packet BBS bulletins from a NOS PBBS system to TCP/IP users. NOS includes an NNTP server, and a somewhat limited NNTP client.

WWW—World**W**ide **W**eb. This is the hottest thing to hit the Internet. It's the protocol that supports the "web browsers" that are making folks like Netscape rich. As a practical matter, WWW requires lots more bandwidth than a 1200 baud radio channel can offer, but both JNOS and TNOS now provide support for the Hypertext Transfer Protocol ("HTTP") used by web servers and browsers.

Building a Network Stack

Protocol Layers

Understanding the TCP/IP suite is much easier if you think of the protocols as being "stacked" atop one another in several layers. Figure 2-2 shows the TCP/IP stack, and referring to it may help the paragraphs that follow make more sense. This seven-layer stack is formally known as the **OSI** (**O**pen **S**ystems **I**nterconnect) model, and being able to talk smugly about Layer 2 versus Layer 3 is a mark of the true networking geek.

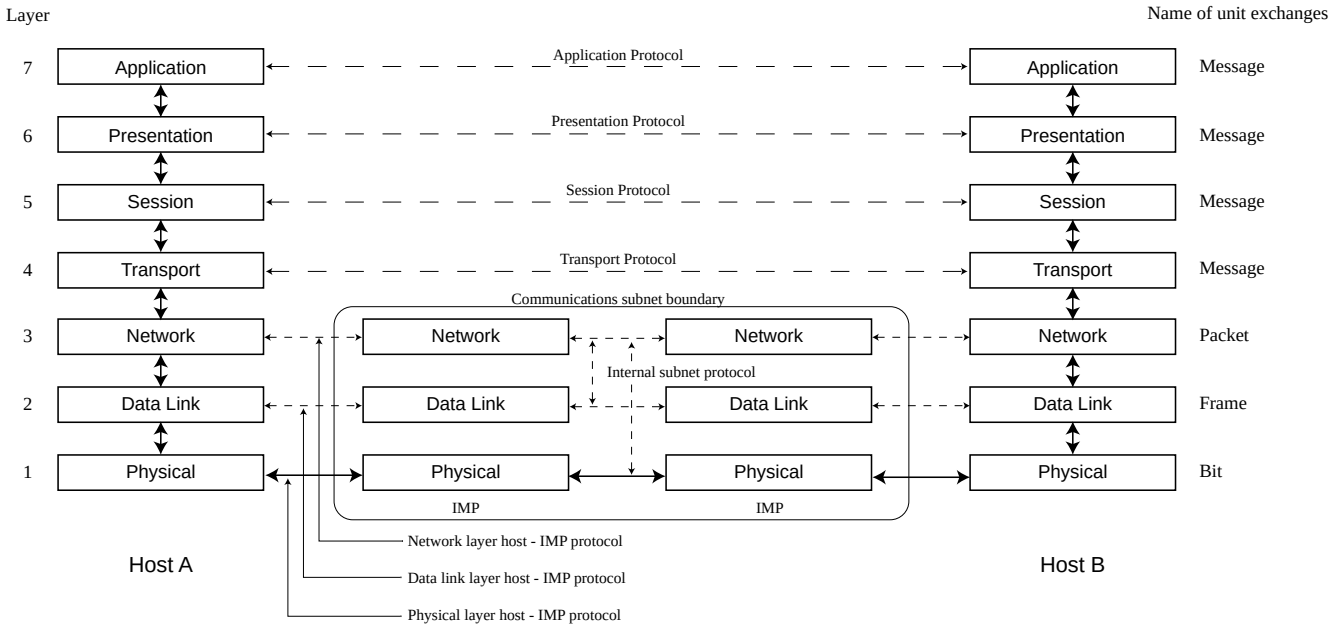


Figure 2-2 - Open Systems Interconnect (OSI) Layers

At the bottom of the stack is the **physical layer** which actually communicates between sending and receiving stations. This may be an Ethernet cable, a serial link, a radio channel, or some other path that electrons can follow.

The next level is the **data link layer.** It contains the lowest-level **framing** structure, i.e., the way the bits are put together into packets. The most common link layer protocol used in packet radio is **AX.25**, an amateur-built standard based on two commercial protocols.[2-2] The first is **HDLC** (**H**igh-level **D**ata **L**ink **C**ommunication), which is usually generated by a chip in the TNC and not by a software program. By the way, as we use the word **frame** (interchangeably with **packet**) in this discussion, remember that a frame has three parts: address and control information at the beginning, data in the middle, and at the end, control information and often error checking information.

The second protocol underlying AX.25 is **X.25** ("AX.25" actually stands for **A**mateur **X.25**). HDLC is able to transmit and receive data, but all it does is wrap that data up in frames; it doesn't include rules for station identification, addressing, or error detection. In the commercial world, the X.25 protocol is often used to add addressing and other necessary pieces to HDLC to create a complete link layer protocol. In the ham network, AX.25 does the same thing. It provides the additional data and control necessary for two stations to connect to each other and reliably exchange data. AX.25 is technically described as an **LAPB** (**L**ink **A**ccess **P**rotocol - **B**alanced) protocol, because the station involved are peers – that's different than an unbalanced link layer protocol, where one station is the "master" and the other is the "slave". [2-3]

Next comes the **network layer**. IP, the network layer protocol, manages the task of getting data from one place to another. It provides the addressing and routing information needed to move data independently of the underlying link layer. IP frames are encapsulated in the data part of data link layer (AX.25) frames. (This is a short description of a very powerful protocol. I'll be talking a *lot* about IP later in this book.)

Fourth up in the protocol stack is the **transport layer**, which ensures that data is not corrupted and arrives in the right sequence. That's TCP's job—it handles retries when packets are lost, and puts the incoming data back into the correct order. TCP frames are encapsulated within the data part of IP frames.

[2-2] *Until recently, FCC regulations required that packet radio use AX.25 as the link layer protocol. That requirement has been lifted, but nearly all ham packet activity is still based on AX.25 today.*

[2-3] *Using AX.25 as the link layer protocol for a TCP/IP network isn't the most efficient way to go. AX.25 has features that IP doesn't need or use, and those features add channel overhead that we can do without. It is also missing some things, like error correction capabilities, that would allow us to get better performance from our RF links. Now that the FCC requirement to use AX.25 is gone, it's likely that someday we'll see a new link layer protocol specifically designed to work with TCP/IP.*

Above the transport layer ride three more slices of the stack. A single TCP/IP implementation may be handling incoming and outgoing data for numerous services and protocols at the same time. The **session** and **presentation** layers manage all that data and make sure it's handed off to the correct top-level service in the **application layer**. The application layer is where the protocols that we actually see and interact with, like ftp and telnet, live.

Encapsulation

The protocols that move data around within TCP/IP network package that data up in **frames** – blocks of bits divided into **fields** in a precisely defined way, as shown in Figure 2-3. The part of the frame that carries the user's data is just one of the fields; other fields contain control information used by the protocol.

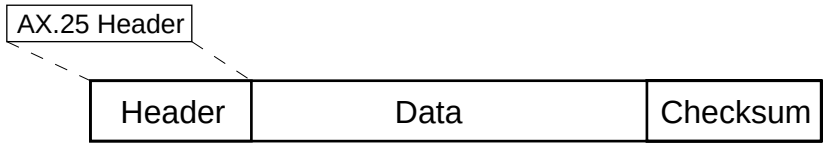


Figure 2-3 - The frame

You can think of a frame and its control fields as a wrapper around its "real" information content. Although each protocol divides frames in its own way, common characteristics of frames are:

- a way to define the beginning and the end of the frame;
- fields at the beginning of the frame that identify its type, and include other information that's useful in processing the frame;
- a data field, which is usually but not always the largest field; and
- additional fields at the end of the frame, often containing information such as checksums that help detect errors in transmission.

The use of defined frames for each protocol makes a layered networking model possible. Frames for higher level protocols are encapsulated as data within lower-layer protocol frames. **Encapsulation** simply means that a lower-level protocol can treat frames of a higher protocol as nothing more than data – a string of bits whose content doesn't

have any impact on the lower protocol's operation. Figure 2-4 shows how encapsulation works.

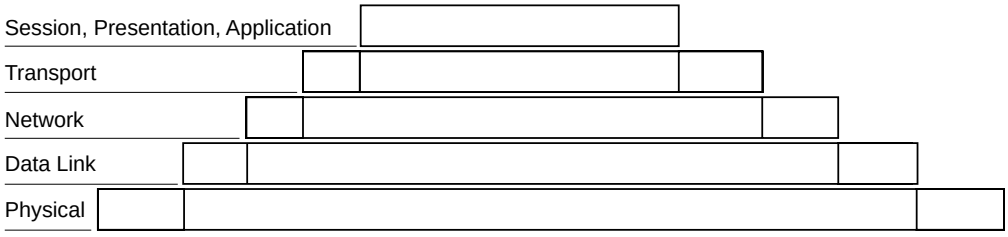


Figure 2-4 - Encapsulation

Encapsulation is important because it allows each layer of the protocol stack to operate independently of the layers above and below it. This isolation means that changes to one protocol affect only the layers using that protocol; the other layers don't need to know about the change. It also allows protocol substitution. For example, we usually talk about using TCP as the transport layer, but there's another transport layer protocol, **UDP** (**U**nreliable **D**atagram **P**rotocol[2-4]), that can also ride atop the IP network layer. Because both a UDP packet and a TCP packet appear to the IP layer as streams of bits, it can handle either one transparently.

Encapsulation has a cost, however. By the time application protocol frames are crammed into TCP frames, TCP frames are shoved into IP frames, and IP frames are wrapped up in AX.25 frames, there's lots of overhead to contend with, because each protocol layer adds its own address and control information. That's one of the tradeoffs made to obtain an elegant and portable networking system. Since the developers of TCP/IP were thinking of Ethernet, or at worst 56kB leased lines, as the physical layer of their network, this wasn't a problem they had to worry much about. But in the ham world, where we're trying to run TCP/IP over 1200 baud radio channels, this overhead is important: every packet of data sent via TCP/IP over AX.25 carries with it 40 bytes of overhead from the TCP/IP protocols, plus from 13 to 62 bytes (depending on the number of digipeaters in the path) of AX.25 overhead. If all the data and overhead has to fit into

[2-4] *Why on earth would anyone want an unreliable protocol??? "Unreliable" is a relative term – what it means here is that the protocol doesn't have an acknowledgement/retry mechanism because it relies on a higher-level protocol to handle that. In some specialized applications, like the Network File System (NSF) protocol, this results in better performance than TCP can provide.*

256 byte packets, as is standard with AX.25, the overhead takes up more than 20 percent of each packet.

One way to increase efficiency is to use larger frame sizes to reduce the proportion of overhead to data; that's why Ethernet uses 1500 byte frames. Adding 40 bytes of overhead to 1460 bytes of data is quite a bit more efficient than using 40 bytes of a 256 byte packet[2-5]. Again, there's a tradeoff: larger frames are more likely to be damaged by collisions or interference (remember that in most data link layer protocols, one scrambled byte clobbers an entire packet), so there's a balancing act between increased efficiency and increased retries.

Encapsulation is a powerful concept, and it can be very useful. For example, let's say you have two computers running NOS and connected to each other via an Ethernet cable, as shown in Figure 2-5. Each computer has a radio port on a different channel, and you want to retransmit on radio channel B the AX.25 packets heard by radio A. It's possible for NOS system A to take those AX.25 packets, encapsulate them inside TCP (and IP, and Ethernet) frames, and send them via the Ethernet to NOS system B. System B can strip off the TCP, IP, and Ethernet headers to recover the original AX.25 frame, and send it to the radio for retransmission.

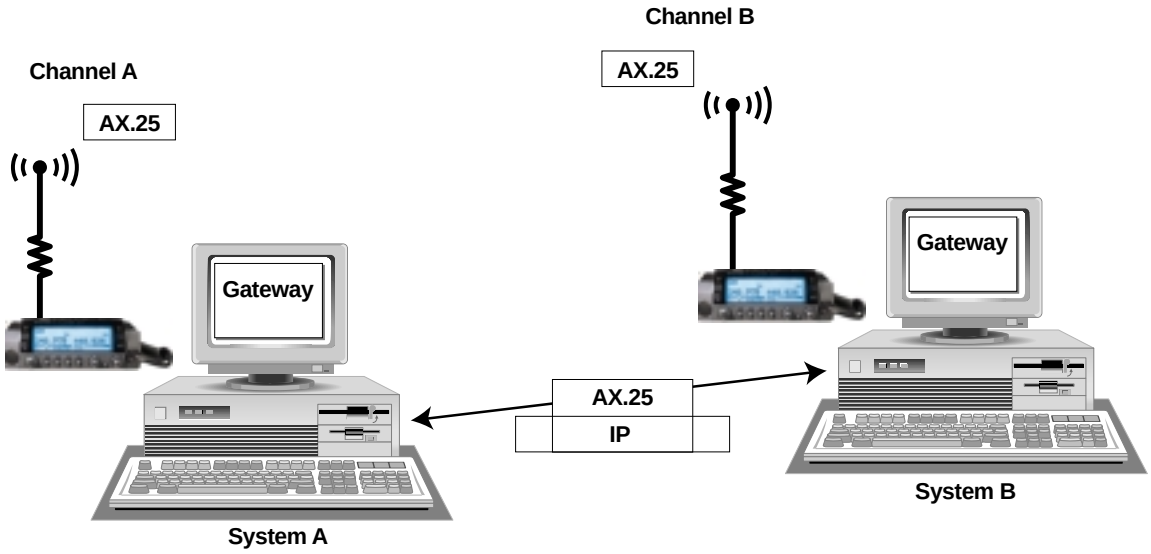


Figure 2-5 - Encapsulation Example

[2-5] *Although AX.25 technically limits packet length to 256 bytes, NOS can bypass this limitation.*

This is exactly what "Internet gateway" systems do, except over a larger network. They accept AX.25 data, encapsulate it in IP, and send it over the Internet to another gateway, which recovers the original data and sends it out over the air. Except for the strange callsigns you might see, this process is completely transparent to the AX.25 users at each end; to them the NOS system looks just like any other digipeater or NET/ROM node. Extending the encapsulation concept further, the AX.25 packets sent via the gateway can themselves encapsulate amateur TCP/IP frames. See Figure 2-6.

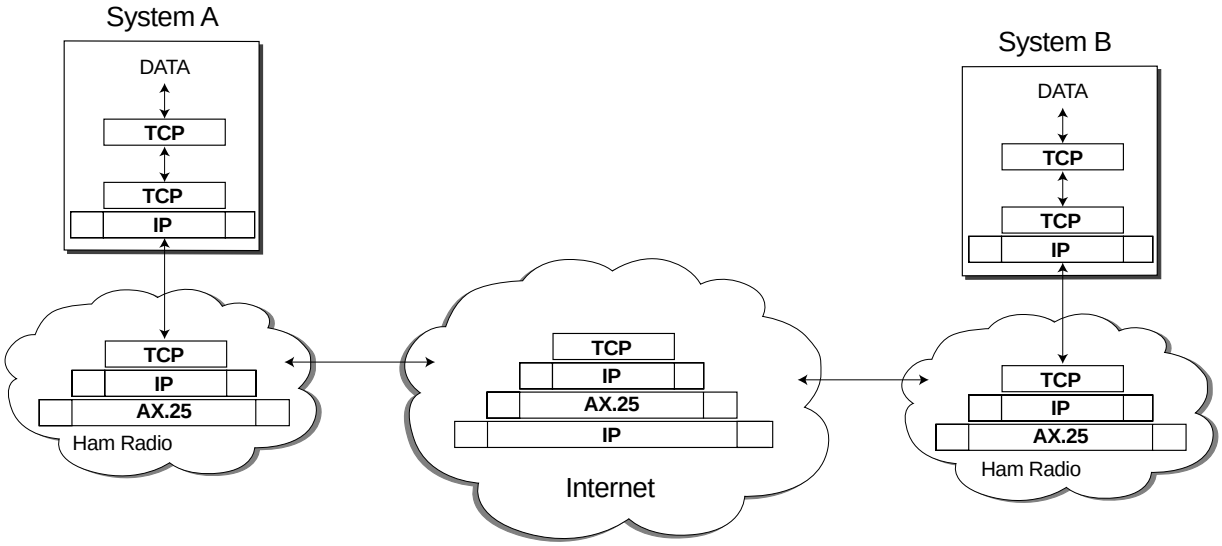


Figure 2-6 - Encapsulation - IP to AX.25 to IP

Why Bother?

This stacking of protocols may seem like a real house of cards, but it gives us some important advantages. By separating layers so that each builds on the one below, developing new protocols is made much easier—the lower level pieces don't need to be reinvented each time. Each layer can be optimized to do what it does best. As the example of Internet gateways shows, encapsulation also allows great flexibility because it lets us "wrap up" protocols inside one another, as we will see later. That makes all sorts of interesting things possible.

Chapter

3

Names and Addresses

It's the IP protocol that does most of the complex stuff in a TCP/IP network—it is responsible for getting data from the sender to its ultimate destination. This chapter explains how IP handles one of the basic problems in any communication system: identifying who's talking to whom. Along the way, we'll see how machine-friendly bits and bytes are converted to human-friendly names.

Why IP Addresses?

Hosts on an IP network are identified with an **IP Address**, which is guaranteed to be unique to that host. This guarantee exists because a central agency manages the worldwide assignment of IP addresses (delegating that task to local **address coordinators** for each network). Every IP frame includes the IP address of both the sender and the final recipient of that frame, and these are the key pieces of information that make the Internet possible.

Why not just use our amateur callsigns as addresses? There are a couple of good reasons. Probably the most important is that a standardized address scheme can provide information for **routing** data from station to station. We'll talk a lot more about routing in the next chapter, but for now you should know that TCP/IP allows packets to move from one location to another via a series of "hops" from gateway to gateway.

IP addresses are designed so that each gateway can figure out which direction to send a packet on its next hop. Ham callsigns don't provide this kind of information, and the assignment of callsigns by national authorities isn't consistent enough to allow routing rules to work (what's good for the prefix hunters isn't good for routing!).

Another reason to use something other than ham callsigns for network addresses is that the world is a lot bigger than ham radio. Callsigns may mean a lot to us, but they aren't very intelligible to the non-ham world. Standardized addresses let us interoperate with other TCP/IP based computer networks.

As we'll see later, there is a place for ham callsigns in the scheme of amateur radio TCP/IP. We use them as **hardware addresses** that identify specific participants in a local area radio network. Although all networking schemes use some sort of low-level address, the format of that hardware address is different for different network hardware. For example, Ethernet uses a unique eight byte address burned into each Ethernet card's ROM chip. Other network cards use different, incompatible addresses. When we're running AX.25 over the air, we use callsigns as hardware addresses.

IP Address Structure

Since these protocols are used on lots of different hardware, it is necessary to use an addressing system that works with all of them, that provides adequate routing information, and that doesn't take up a lot of space. To meet these needs, IP addresses are formed as binary numbers 32 bits long, as shown in Figure 3-1. A defined number of bits starting from the left are used for the network identification, or **netid**, while the remainder form the host identification, or **hostid**. The number of bits used for each part of the address varies depending on the type of network and is controlled by the **netmask** associated with the network. The next pages will describe all this in much more detail.

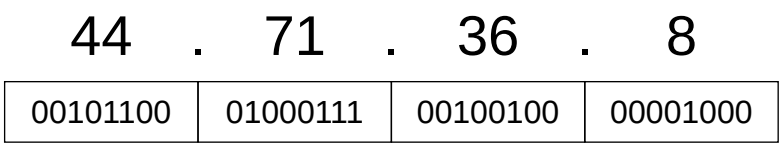


Figure 3-1 - IP address

For convenience, we usually treat the 32 bit IP address as four bytes of 8 bits each[3-1], and print and talk about IP addresses like this: **44.71.36.8**. This is known as **dotted decimal** or **dotted quad** notation because each of the four, period-separated numbers is the decimal representation of one of the address bytes. By the way, you should never add a leading zero to the numbers – "044.071.036.008" – because many systems can't deal with address written that way. IP addresses can also be written in octal or hexadecimal format, but except for the way some systems describe netmasks (which I'll talk about later), you'll seldom see either of those notations used.

Separating Networks and Hosts

An IP address consists of two parts: a **network** part (**netid**) and **a host** part (**hostid**). Though it's simplest to think of the address as using all its 32 bits to identify a specific host, that will lead to confusion. An IP address identifies both a network, and a host's connection to that network. Gateways use the network part of the address to identify a network, and use their routing tables (discussed in the next chapter) to decide how to move the packet closer to that network.

Once the packet reaches a gateway on the same network as the desired host, that gateway uses the hostid to send the packet on the final hop to its destination. Without this separation of networks from hosts, gateway routing tables would have to contain entries for each host, and that's simply not practical. Using the IP address to carry network information greatly simplifies the gateway's task.

[3-1] A note for trivia lovers: In the IP world, an eight bit chunk of data is formally known as an "octect" and not a "byte", because there are some computers systems that use bytes of different lengths. In this book, I use "byte" because all the machines commonly used for amateur TCP/IP have eight-bit bytes, and "byte" is a much more natural term for most of us.

The division between the network and host parts of an address is based on the **class** of the network, as assigned by the international organization in charge of such things. The division can occur after the first byte (a **Class A** network), after the second byte (**Class B**) or after the third byte (**Class C**). These classes are designed to accommodate networks of different sizes. A Class A network can support *lots* of hosts — 2^{24} minus 3, to be exact. But there can only be a handful of Class A networks because the single network byte supports only at most 256 networks — and actually the allocation scheme allows for only 127 of them. Class B networks use the first two bytes for the netid, and the last two bytes for the hostid. There is room for about 16,000 Class B networks that can each support about 65,000 hosts. Finally, with three bytes of netid, there can be lots and lots of Class C networks, but each one only supports 253 hosts[3-2].

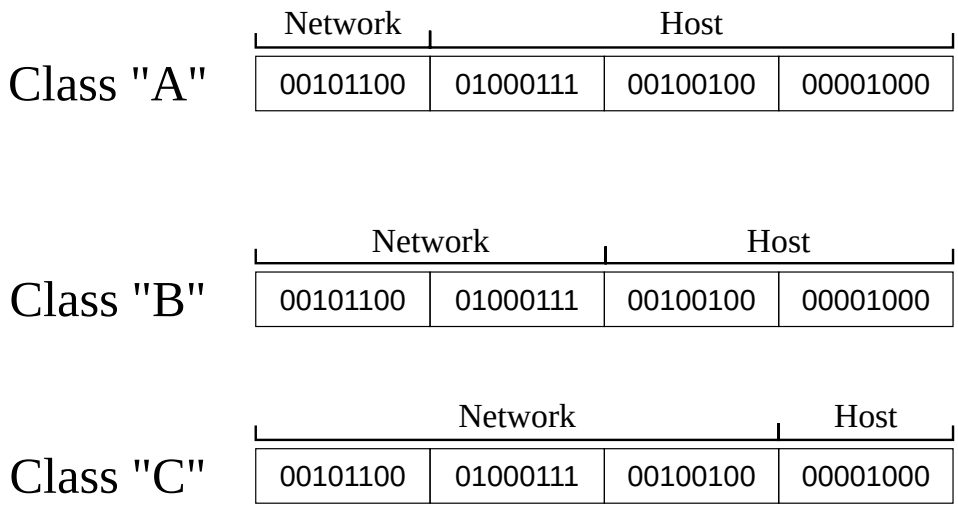


Figure 3-2 - Network and Host Parts of an IP address

For administrative purposes, networks assigned by Network Information Center are given names. Thanks to some forsighted effort by Brian Kantor, WB6CYT and others, the amateur radio community has been granted use of a Class A network (one of only 127) that's known as **amprnet** – a handy abbreviation that I'll use frequently.

[3-2] *Hostids with bits that are all zero or all one are reserved for special purposes, so each network can have at most two hosts less than the maximum number – 253 instead of 255 for a Class C network.*

Networks can be broken into **subnets**—smaller groups of systems that are treated as their own networks. Routes can point to subnets as well as to networks, and this provides flexibility to deal with strange routing problems and complicated physical designs. Subnets can also simplify network administration by delegating some of the responsibility for a network down to local groups. Subnet are created by using some of the most significant (left-most) bits of the hostid to instead identify subnets. Although it's most common to create a subnet on a byte boundary, it's possible to steal only a few bits rather than a whole byte or more to serve as the subnet part of the address[3-3]. Figure 3-3 shows how an IP address is broken into network, subnet, and host parts.

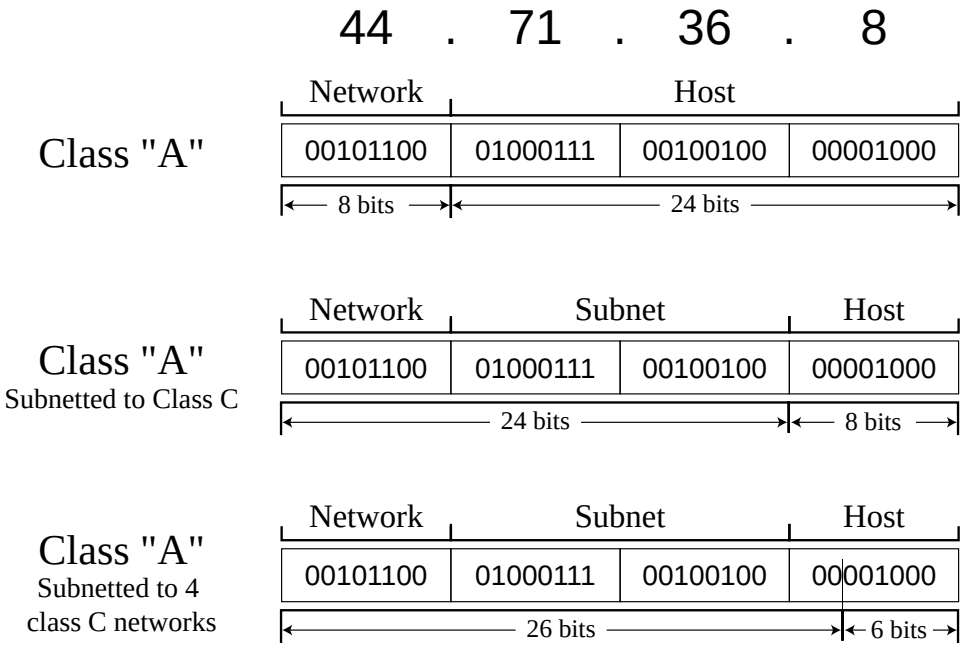


Figure 3-3 - Subnetted Addresses

[3-3] *Until recently, the official rules for creating subnets limited how many subnets could be created by requiring that the subnet part of the host ID could neither contain all zeroes, nor all ones. Without going through all the math, this meant that a Class C network could not be split in half to create a pair of 126 host networks; the best you could do was break it into four parts, throw away two of them, and end up with two 62-host subnets. However, most IP implementations today, including all versions of NOS and Linux, ignore that restriction, and current Internet design rules permit essentially unlimited subnetting. Most ham IP networks ignore this limitation without ill effect, but this may be a "gotcha" if you need to internetwork through commercial routers that still follow the old rule.*

Amprnet is subnetted. It is a Class A network and is therefore "flat"– it supports *lots* of hosts that (in theory, but not in ham radio practice) can all talk to each other directly. But since ham IP systems are scattered around the world and don't have any common interconnection, we've decided to create subnets—using the second byte of the address (the first byte of the hostid) to establish a subnet for each state or country, and some or all of the third byte (the second byte of the hostid) for regional subnets. This makes it easier to administer the network, and makes it *much* easier to set up routing systems[3-4]. Table 3-4 shows how subnets of the amateur radio network are set up.

44.002	USA:Calif: Sacramento	44.054	USA:Vermont	44.082	USA:Montana
44.004	USA:Calif: Si Valley - SFO	44.056	USA:Eastern&Central Mass	44.084	USA:Colorado: Western
44.006	USA:Calif: Sta Barb	44.056.12	USA:Mass: Worcester	44.086	USA:Wyoming
44.008	USA:Calif: San Diego	44.058	USA:West Virginia	44.088	USA:Connecticut
44.010	USA:Calif: Orange County	44.060	USA:Maryland	44.090	USA:Nebraska
44.012	USA:Eastern Wash,Idaho	44.062.0	USA:Virginia-Central	44.092	USA:Wis, up pen Michigan
44.014	USA:Hawaii & Pacific	44.062.32	USA:Virginia-Charlottsville	44.094	USA:Minnesota
44.016	USA:Calif: LA/Valley	44.062.64	USA:Virginia-Eastern	44.096	USA:District of Columbia
44.017	USA:Calif: Kern County	44.062.128	USA:Virginia-Western	44.098	USA:Florida
44.018	USA:Calif: San Brdo	44.062.192	USA:Virginia-Northern	44.100	USA:Alabama
44.020	USA:Colorado: Northeast	44.064	USA:New Jersey: northern	44.102	USA:Mich (W lower pen)
44.022	USA:Alaska	44.065	USA:New Jersey: southern	44.102	USA:Mich (E lower pen)
44.024	USA:Washington:Western	44.066	USA:Delaware	44.104	USA:Rhode Island
44.026	USA:Oregon	44.068.1	USA:New York: NY & LI	44.106	USA:Kentucky
44.028	USA:Texas: North	44.068.48	USA:New York: 30 Rock	44.108	USA:Louisiana
44.030	USA:New Mexico	44.068.52	USA:New York: NY & LI	44.110	USA:Arkansas
44.032	USA:Colorado: Southeast	44.068.64	USA:New York: ENY	44.112	USA:Penn: western
44.034	USA:Tennesee	44.069	USA:New York: WNY	44.114	USA:N&S Dakota
44.036	USA:Georgia	44.070	USA:Ohio - oldnet	44.116	USA:Oregon,WA
44.038	USA:South Carolina	44.071	USA:Ohio - newnet	44.118	USA:Maine
44.040	USA:Utah	44.072/16	USA:Illinois	44.120	USA:special use in Nevada
44.042	USA:Mississippi	44.073/16	USA:Illinois	44.122	USA:Kansas
44.044	USA:Mass:western	44.074	USA:North Carolina (east)	44.123	USA:Virgin Islands
44.046	USA:Missouri	44.075	USA:North Carolina (west)	44.124	USA:Arizona
44.048	USA:Indiana	44.076	USA:Texas: south	44.125.0	USA:Southern Nevada
44.050	USA:Iowa	44.077	USA:Texas: west	44.125.128	USA:Northern Nevada
44.052	USA:New Hampshire	44.078	USA:Oklahoma	44.126	USA:Puerto Rico
		44.080	USA:Pennsylvania: eastern		

Table 3-4 - Amprnet subnets

This list does not contain International subnets. Refer to UCSD master subnet list for all subnets.

[3-4] Some may argue that amprnet subnets exist only for administrative and not routing purposes, but as a practical matter we rely on these subnet for routing.

We won't get into all the semantics of addressing just yet, but as an example the address 44.71.36.8, as illustrated in Figure 3-5, breaks down as

44. The network assigned to amateur radio TCP/IP.
71. The subnetwork for Ohio.
36. The Dayton subnetwork.
8 A specific host address within that area.

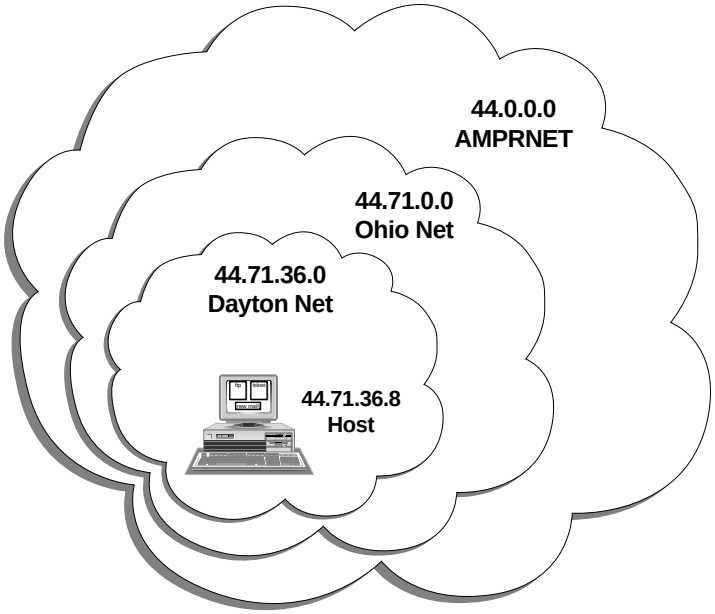


Figure 3-5 - Dayton Net Subnet example

It follows from all this that an IP network (or subnet) has one important characteristic: *all the hosts and subnets on a network are reachable from outside the network through a single gateway*. Figure 3-6 shows an example network with subnets. To the outside world, all the hosts on the Ohio network can be reached through gateway 44.71.1.1. Within the network, all the hosts on subnet 44.71.36.0 can be reached through gateway 44.71.36.1, and all the hosts on subnet 44.71.32.0 can be reached through gateway 44.71.32.1. Although the subnets are meaningful within the 44.71.0.0 network, they are invisible to the outside world—the network gateway accepts packets bound for the subnets, and forwards them on to the appropriate subnet gateway.

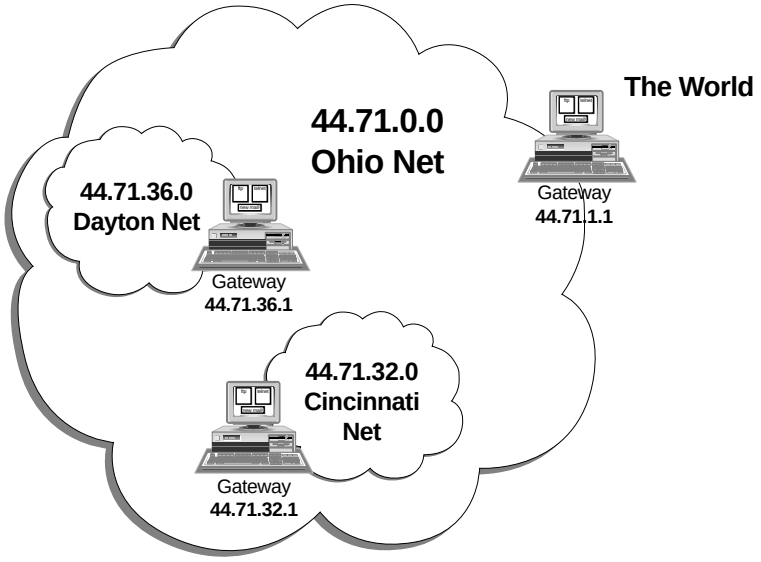


Figure 3-6 - Example network with subnets

By the way, it's possible for a single host to have more than one IP address (Figure 3-7). If it has more than one **interface** (a physical connection to a single network), each interface may have its own address. Although NOS doesn't require you to use a different IP address on each interface, it's a good idea to do so. If a host has two interfaces, each talking to a different network or subnet, each interface needs an address that's part of the network it belongs to. You'll see why when we talk about how IP routing works.

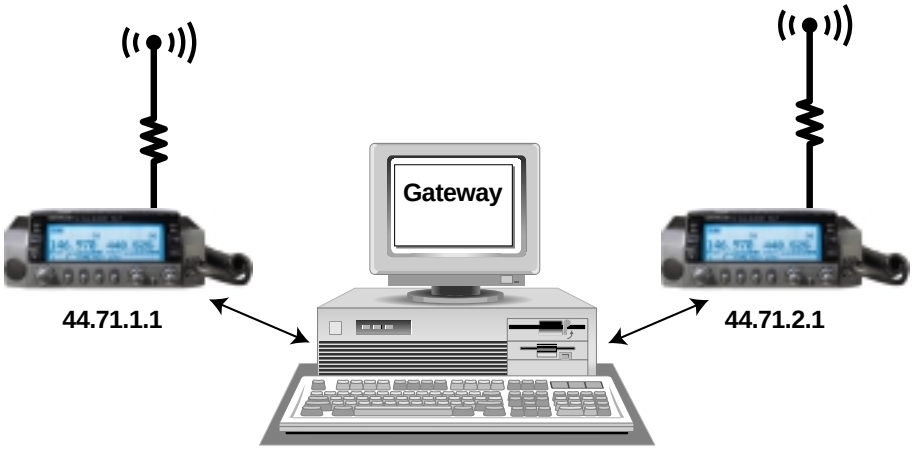


Figure 3-7 - "Multi-homed" host.
Each interface has its own address that is local to the network with which it communicates.

IP Address Coordination

If you're the only IP user in town, or you and a couple of other local hams are experimenting with NOS and don't have any sort of gateway or other connection to other IP hosts, it's not necessary to obtain a registered IP address. As long as you don't connect to the outside world, the addresses you use won't matter to anyone else. But if you have any sort of link to other IP hosts, you'll need to get one or more IP addresses assigned to your system. The amprnet subnet 44.128.0.0 has been set aside for experimental and uncoordinated use.

Addresses are assigned by coordinators who derive their authority from a central registry. The coordinator for amprnet is Brian Kantor, WB6CYT. He has delegated authority for subnets to various state and national coordinators, and the coordinator assigned to your country or state is the person you should contact to obtain an address. Appendix B lists these coordinators as of this printing; an up-to-date version is available from the **ftp.ucsd.edu** site on the Internet.

IP Ports

There's another part of the IP address scheme that you should be aware of. You'll recall that a host can have many simultaneous streams of data flowing, supporting multiple sessions of telnet, ftp, smtp, and other protocols, and those multiple sessions can all be going to a single remote host. How does it keep them all straight ?

The answer is **ports**. A port is an additional part of the address information sent in each IP frame. Each server, like telnet, ftp, and smtp, listens for new connections on a specific, standard port. For example, the telnet server listens on port 23.

So, when a packet arrives at a host with the destination port set to 23, the TCP/IP system hands that packet to the telnet server to open a new telnet session. The server assigns a unique port (usually numbered above 1024) to that session and sends that port number back to the client. This **session port** is used to identify the packets, both incoming and outgoing, belonging to that session. This structure allows packets for different servers and sessions to be **multiplexed** using a single IP address.

Port numbers are usually invisible to the user, and don't normally require local configuration; the numbers used are standardized and are built into the default configuration of most TCP/IP protocol stacks, including NOS. However, there are times – particularly for debugging – when knowing about port numbers is very useful.[3-5]

Hostnames

Though IP addresses make perfect sense to a computer, they are not very intuitive to human beings, and they are a real pain to remember. English-like **hostnames** are much easier to use, and TCP/IP programs, including NOS, map hostnames to IP addresses for the benefit of human beings.

The convention in ham radio TCP/IP is to use callsigns as hostnames. To help reduce confusion, we usually print hostnames in lower case, and callsigns in capital letters—my hostname is *n8ur*,[3-6] and my call is "N8UR". (Hostnames aren't case-sensitive, so you are free to use any combination of upper or lower case that suits your fancy — the computer won't care.)

There's no requirement, though, that the hostname match your call, and there are some instances where another name is more descriptive. For example, the hostname ***dayton-switch*** is more meaningful to a distant station than ***w8apr*** would be. A hostname can be any combination of letters or numbers (plus a few punctuation symbols) up to __ characters long. Hostnames are not case sensitive.

You should be a bit restrained in your creativity, though. There are two important things you need to remember when deciding on hostnames. First, any hostname on the amprnet needs to be unique -- if Dayton, Ohio; Dayton, Indiana; and Dayton, Tennessee all claim ***dayton-switch.ampr.org***, there's going to be trouble. This is a strong argument in favor of using callsigns as hostnames, or at least including a call as part of the name; you can be pretty confident that no one else has the same callsign you do.

[3-5] Port numbers can be an issue if you are connecting to the Internet from behind a firewall. As a security measure, many firewalls block packets bound for certain ports from passing; this can prevent some protocols from working through a firewall.

[3-6] Except in sample configuration files, I'll put hostnames in italics to set them off. Unless there's a good reason, I won't include the trailing ampr.org in the examples; just assume that you'd add that part to the end of the hostname to create a full address.

Second, where appropriate you should try to make hostnames descriptive. ***switch-w8apr*** provides a useful clue about the purpose this host serves. By the way, it's OK to create multi-part hostnames using periods as separators, like ***switch.w8apr.ampr.org*** but I personally don't like this format -- it implies a hierarchical structure that probably doesn't exist. If you have multiple interfaces on your host, you may want to give each one a descriptive name, like ***vhf-n8ur***, but it's probably a good idea to ensure that you have at least one published hostname that is no more or less than your callsign -- because that's the name people will naturally try when they want to reach you.

Aliases (also known as **CNAME records**) provide better way to create descriptive names for local hosts, and I'll describe how to use them a bit later.

By itself, a hostname doesn't constitute a complete address. Tacked to the end of the hostname is a **domain name**. A **domain** is a group of machines that are logically (though not necessarily physically) connected together. Domain names, like IP addresses, are hierarchical; periods separate parts of the name, with each part representing a different level. The domain name, however, is ordered backwards from an IP address—its highest-level portion is at the right.

The ham network's domain is *ampr.org*[3-7]; "org" (short for "organizations") is the top level domain, and "ampr" (for **AM**ateur **P**acket **R**adio) is the second level domain, containing all ham TCP/IP hosts. Any lower level parts of the hostname are optional — the ampr.org domain doesn't officially recognize, for example, state or country subdomains like "usa.ampr.org".

When you combine a hostname with a domain name, you get something like *n8ur.ampr.org*. This is called a **Fully Qualified Domain Name** (**FQDN—**knowing this acronym allows you to sound like a real expert). For services like email, where the identity of a specific user is important, we add the user's login name to the beginning of the address, separated from the FQDN by a "@" character. This combination is commonly

---

[3-7] Not to be confused with the "amprnet" IP network. IP networks and domains of hostnames are two very different things; I'll talk more about the distinction later.

known as an **Internet address** and is the address form used for most electronic mail in the real world. For example, if there is a user "jra" at *n8ur*, *jra@n8ur.ampr.org* would be that user's full internet address. Figure 3-8 illustrates this.

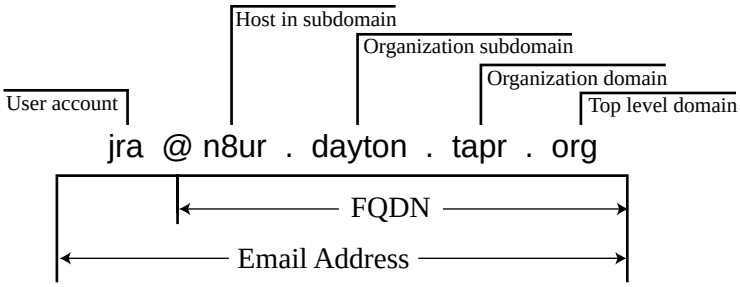


Figure 3-8 - Internet E-Mail Address and FQDN

Although all the hosts on a single IP subnet should be able to reach each other directly, that's not the case for domains. Assuming that domains and networks map directly to each other can cause a *lot* of confusion. One network can contain hosts from several domains, and vice versa. Think of the IP network and its connectivity as an engineering concept, and the domain as an administrative one. IP subnets make life easier for routing software; domains are intended to make life easier for system administrators.

The "amprnet" 44.0.0.0 Class A IP network (often referred to as our **address space**), and "ampr.org" as our hostname domain (our **namespace**) are two completely separate concepts that don't have any linkage to one another. Any hostname can point to any IP address, regardless of he network or domain to which each belongs.

For example, *n8ur.ampr.org* could point to a host on network 192.128.55.55, and similarly *n8ur.febo.com* could legitimately point to a host with 44.71.36.8 as its IP address. This sort of mix-and-match identification is particularly common at Internet to amprnet gateways. Understanding that address spaces and namespaces are completely separate (in technical terms, **disjoint**) is key to understanding TCP/IP's flexibility. An example is shown in Figure 3-9.

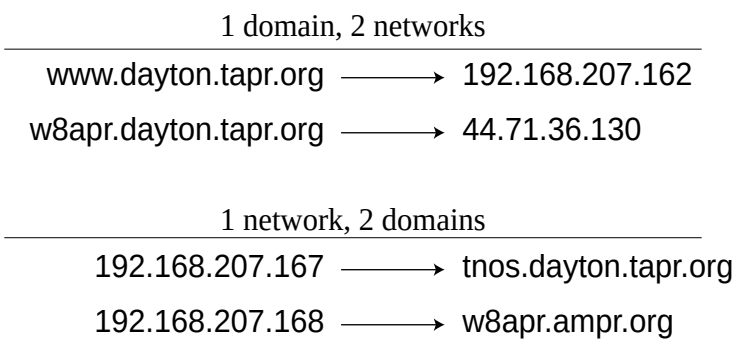


Figure 3-9 - Domains and Networks aren't the same thing!

Now is a good time to address one very common area of confusion. Having an internet address doesn't mean that you can be reached on the Internet! The Internet (with a capital "I") is the name used for the maze of interconnected networks that use TCP/IP protocols to support public access to World Wide Web, ftp servers, the worldwide email system, and much of the Usenet or "NetNews" system. However, you can have access to many of the services offered by the Internet without actually being an Internet host. Conversely, just because you have an internet address *doesn't* mean that email from the Internet will magically arrive at your doorstep.

This is especially true of the ampr.org domain. Unless you have an arrangement with an Internet or uucp gateway, and an appropriate record at the *ampr.org* domain name server[3-8], your host doesn't exist as far as the Internet is concerned. The bottom line is that *ampr.org* uses internet protocols, and its services look like those offered by the Internet, but it's really a parallel universe. Only where a bridge exists between the Internet and *ampr.org* universes can data flow between them.

For example, until recently my home system didn't have a direct connection to the Internet, but mail sent to my ampr.org address still reached me at home. That's because I had an arrangement with another system that was on the Internet to act as a **mail**

[3-8] *The master DNS database for* ampr.org *lives at* ucsd.edu, *and this sort of email gatewaying requires proper records in that database. Your amprnet IP address coordinator can add those records as necessary.*

exchanger for me. When people sent mail to my ampr.org address, the **DNS** (short for **Domain Name System** – more on that in the next pages), pointed that traffic to my mail exchanger, which then sent the messages on to me via **uucp**, a UNIX-based networking system that's designed to transfer messages via dialup phone lines. So, even though I wasn't "on the Internet" I was able to send and receive Internet email. Similarly, it's possible to be part of the Usenet system via uucp without having a direct Internet connection.

Mapping Hostnames to IP Addresses

Hostnames have to be linked somehow with the IP addresses they represent. This can happen in two ways: either by manually maintaining a file containing the hostname to IP address mappings of all the stations the host ever wants to reach, or by asking a nameserver to provide the mappings as they are needed (Figure 3-10). Whether manually or automatically obtained, on NOS systems the information is stored for future use in a file called DOMAIN.TXT.

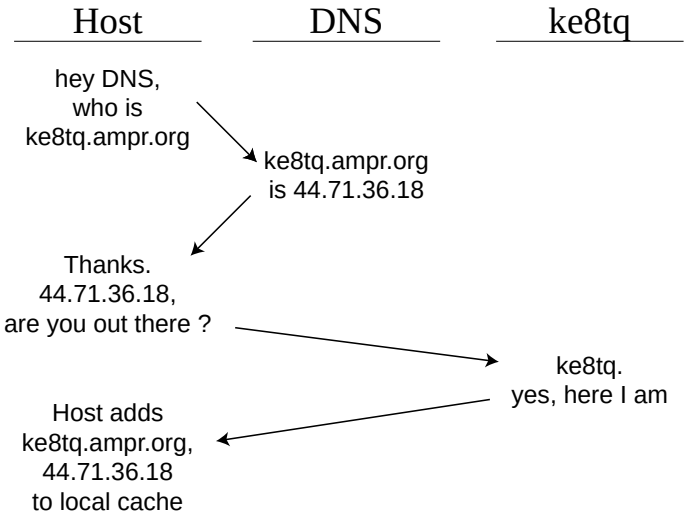


Figure 3-10 - DNS lookup

It should be apparent that using a nameserver is a lot easier than manually creating and maintaining a DOMAIN.TXT file (Figure 3-11). NOS includes nameserver support and if there's a full-time host on the local network that is willing to do the job, everyone else's life will be made easier.

```
# Regular address records
w8apr.ampr.org.          IN      A       44.71.36.2
n8ur.ampr.org.           IN      A       44.71.36.17
ke8tq.ampro.org.         IN      A       44.71.36.18

#A "cname" or alias to give an easy to remember name to a popular server
mvfma.ampr.org.          IN      A       w8apr.ampr.org.

# A mail exchanger record -- mail to ke8tq will be sent to w8apr
ke8tq.ampr.org.          IN      MX      w8apr.ampr.org.
```

Figure 3-11 - Sample Domain.txt entries

DNS relies on the existence of a nameserver which either knows, or can find, the IP addresses for all the hosts that need to be reached. In the connected Internet, each domain has a master nameserver that provides authoritative answers to DNS requests. The DNS system is designed so that all requests for mappings ultimately relate back to that authoritative data. If your network is connected to the Internet via a gateway, it's possible to configure the local namserver to work in this way, using data from the master ampr.org nameserver at *ucsd.edu* (setting all that up is beyond the scope of this book). However, using the master nameserver also requires that all the hosts you need to reach are registered there – and many *amp.org* hostname/IP address pairs have been assigned by local coordinators who don't upload their data to the master database.

If you can't get authoritative data via the Internet DNS system, you'll need to manually maintain the necessary records at your local nameserver. This isn't as good as having it done automatically, but it's still a lot less effort than requiring everyone to maintain their own DOMAIN.TXT file by hand.

DNS uses several types of address records. The most common is a simple mapping of an official hostname to its IP address — these are called "**A**" (for **A**ddress) records. Another type is the "MX" record (for **M**ail e**X**changer). It doesn't match a host to an IP address, but rather to another host that will receive mail on its behalf. That's how my ampr.org email link worked. When someone on the internet wanted to send a mail message to me, their mail system asked DNS to find *ag9v.ampr.org*. DNS responded with the my mail exchanger's address, and the sender shot the message off to the mail exchanger using the smtp mail transfer protocol.

A third type of DNS record handles aliases (nicknames) for hostnames. It's called a **CNAME** record (short for **C**anonical **NAME**). CNAME records are useful because sometimes a system's official hostname is long, or doesn't bear much resemblance to what that host does. For example, a system that offers a public ftp server might have an A record in DNS under the hostname *s321.iggy.net5.foobar.com*. It could also have a CNAME record for *ftp.foobar.com* that points to that long name but is much easier to remember (and type!). That's what a CNAME record really is: a pointer to the official hostname. One important thing to remember is that a CNAME record *doesn't* directly match an IP address; it can *only* point to a hostname that's in the DNS system, and DNS must then obtain the IP address from the A record for that host.

Of course, if you choose nice, descriptive CNAMEs like "mailserver" and upload them to the master ampr.org database, you may find that you're overwriting someone else's use of the same alias. That's not very polite. But the chances are good that most of the CNAMEs you want to use are really of local interest only, and don't need to be part of the master database that's accessible worldwide. If that's the case, you can configure your local nameserver to support these alias records, without uploading them to the database at ucsd.edu. Just add them to the domain.txt file of the local server.

The DOMAIN.TXT file that NOS uses to find hostname-to-address mappings is a DNS database, and it can contain all three of these record types (as well as some other less common types). Appendix F shows the layout of DOMAIN.TXT.

There's one last twist to hostnames and FQDNs. Some services (such as DNS) need to know whether an address they are processing is in fact an FQDN. To do so, they look for a trailing period at the end of the domain name. Old versions of NOS ignore this issue, but newer ones (such as JNOS and TNOS) insist that you "anchor" all domain names with a period at the end of the name. If you don't, they'll add the default domain suffix (usually *ampr.org*, set by the **domain suffix** command described later) to the end of the address.

In other words, if you issue the command "hostname n8ur.ampr.org" to JNOS, it will think your hostname is *n8ur.ampr.org.ampr.org* and things will not work at all the way you expected. Assuming you've set the domain suffix to *ampr.org*, entering either "**hostname n8ur**" or "**hostname n8ur.ampr.org.**" (note the trailing period) makes things work the way they should. To avoid confusion, I prefer to always use FQDNs with trailing periods in the DOMAIN.TXT file. It's dangerous to rely on the domain suffix being properly set. Figure 3-12 shows how the domain suffix affects entries in DOMAIN.TXT.

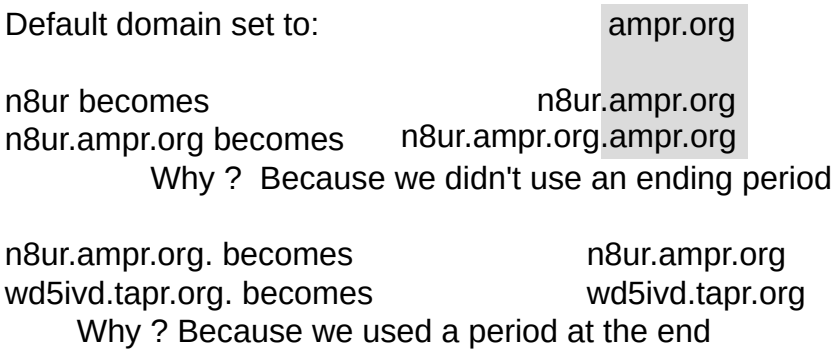


Figure 3-12 - Importance of the trailing dots

Hardware Addresses

Earlier, I said that although ham callsigns don't make good candidates for IP addresses, they do serve an important purpose in the amateur radio TCP/IP architecture. That purpose is to serve as the **hardware address** that identifies a host (more specifically, a particular network interface on a host) to other hosts on the local network.

Remember that protocol layers are encapsulated, and that lower layer protocols have no knowledge of what's in the higher level frames they transport. Thus, it's not possible for the data link (in our case, probably AX.25) to know the IP addresses contained in the IP frames it is handling. And, knowing them wouldn't be of much use anyway because the AX.25 protocol relies on ham callsigns as identifiers. AX.25 packets are sent from one callsign to another, not from on IP address to another. The same is true of other physical and data link layer mechanisms: Network cards also have unique identifiers, which their data link layer protocols use as hardware addresses.

The **A**ddress **R**esolution **P**rotocol (ARP) is used to map IP addresses to hardware addresses. One important thing to understand about hardware addresses and ARP is that they operate only on the local area network – in other words, between those hosts who can communicate with each other directly without going through an IP switch. Hardware addresses never travel beyond the LAN – IP addresses are the identifiers that travel through gateways.

Because hardware addresses and ARP work intimately with the routing process that's discussed in the next chapter, it's best to leave a further explanation of how hardware addresses and ARP fit into the picture until then.

Chapter

4

Routing

It's time now to talk a bit more about how the Internet Protocol routes packets so that data can move from one host to another through gateways and other complications.

First, what is routing, and why bother? Part of the magic of TCP/IP, and particularly the IP part of it, is that it can be used to bridge networks and connect hosts that are thousands of miles, and dozens of networks, away. The IP routing mechanism give us a way to move a packet toward its ultimate destination in small steps, without requiring hosts to know the full route to each and every other host.

The concept is simple: the responsibility of each IP host that receives a packet is to move it one step closer to its final destination. If the host is the final destination, it moves the packet up to whatever higher-level protocol is going to deal with it; otherwise, it sends the packet on to the next gateway in line.

How does a host decide who the "next gateway in line" is? That's where routing comes in. Routes set in each host's configuration files provide rules that tell it how to handle incoming IP frames. These routing rules operate on the frame's destination IP address, and they are what this chapter is all about.

This scheme provides one of IP's greatest strengths: the originator of a packet does not have to know how to get that packet to its destination. It only needs to know how to reach a gateway that can move the packet a bit further along. To paraphrase the stockbroker's advertisement, "At IP Hutton, we measure success one hop at a time."

Subnets, LANs, and Routing Rules

In an ideal world, the networking software would be able to figure out all by itself the best way to get a packet from point A to point B, and to work around system outages and propagation quirks. The Internet uses several protocols to accomplish this feat, and they work pretty well. Unfortunately, that sort of **dynamic routing** is difficult to accomplish in RF networks like ours, and although folks are working on the problem, at present most ham IP networks use **static routing** instead. Static routing simply means that the rules defining where a host should send packets are manually entered into a **routing** table on each host, and they need to be manually changed when the network changes.

A TCP/IP host has one or more routes defined in its configuration files. At run time, these build a **routing table** in memory that describes each route's destination network, the hardware interface used to reach that network, and, if the destination is a remote network, a gateway on the local network which knows how to forward packets toward that network.[4-1]

Every host has at least one route, most hosts have two, and some have three routes or more. The one route that must be present is the **default route**, which tells the system where to send packets when there's no more specific route available. If your network has one gateway system that reaches the rest of the world, your default route probably points there. The second route handles hosts on the local area network—there's no

[4-1] *The destination in a routing rule can be a single host, or more usually, a network.*

reason to go through a gateway to talk to hosts you can reach directly. In NOS, the following commands might be used to establish these two routes:

```
route add default vhf 44.71.36.1
route add 44.71.36.0/24 vhf
```

The first command says that all packets should be sent via the vhf interface to the gateway station with IP address 44.71.36.1. The second command says that all packets bound for stations whose first three IP address bytes are 44.71.36 should be sent directly via the vhf interface — meaning these stations can be reached directly, without going through a gateway.

Note the **44.71.36.0/24** in that command. That syntax the key to NOS routing. The part before the "/" identifies the beginning address of the network, and the "24" — the network mask, or **netmask** — indicates that the first 24 bits of that address are the **netid** (network part). 24 bits is three bytes, so in this example the left three bytes – 44.71.36 – form the netid. Each incoming packet's destination address is checked against the routing table, and if the first three bytes of the packet's address match the first three bytes of one of the routes, that route is used to send that packet on its way toward its destination. In this example, a packet destined for 44.71.36.8 would match this route, because the first 24 bits are the same as the netid.

The use of the "/xx" notation to define the number of network bits in a routing statement is unique to NOS. Most TCP/IP implementations use a different notation. They usually write netmasks in dotted decimal form, though some systems may use hexadecimal, octal, or even binary format (NOS supports netmasks in hex format as part of the **ifconfig** command).

In netmask notation, all the bits assigned as the netid are set to one, while those assigned to the hostid are set to zero; for ease of use, the result is converted to dotted decimal or hex form. For example, a netmask of 255.255.255.0 (or "0xffffff00" in hex) means that the first three bytes of the route serve as the netid, and the last byte serves as the hostid. A system using this method might format a route statement like this:

```
route add 44.71.36.0 netmask 255.255.255.0 vhf
```

Figure 4-1 shows how the netmask is used to derive the netid and hostid parts of an IP address. Using netmasks yields the same results as the "/24" notation described above. I use the NOS "/xx" notation in the rest of this discussion because it's both shorter than the other method, and more descriptive of what's happening.

Address	44.71.38.2	=	*00101100 01000111 00100100 00001000*
Netmask 24 bits	255.255.255.0	=	*11111111 11111111 11111111 00000000*
Result		=	*11111111 11111111 11111111* *00001000*
			Host Part

Figure 4-1 - Netmasks

But wait a minute — what if two or more routes match a destination address> Since the default route, which matches *all* addresses, is defined first, why don't all packets, even those intended for the local network, get sent via that route to the gateway? The default route doesn't get in the way because each packet is evaluated against all known routes, and matches the destination address against the *most specific* network — the one with the longest netmask — that appears in the routing table. The default route has a netmask length of 0 — meaning that any other route is more specific, and will take precedence over it.

Adding a third route to this example shows the power of routing rules. Let's say that for some reason, one station — *ke8tq* [44.71.36.18] — can't be reached directly even though he's part of the 44.71.36 network. We need to go through a gateway whose address is 44.71.36.3 to reach him. Here's a routing command to do this:

```
route add 44.71.36.8/32 vhf 44.71.36.3
```

Our table now includes three routes:

```
default            vhf       44.71.36.1
44.71.36.0/24      vhf
44.71.36.18/32     vhf       44.71.36.3
```

To reach *ke8tq*, NOS would use the last route in the table to send the packet via 44.71.36.3, not one of the other two routes. Why? "44.71.36.18/32" tells NOS to match incoming packets against the entire address ("/32" signifies that all four bytes is used as the netmask, and thus the whole address is used as the netid). This 32 bit match is more specific than the 24 bit match for 44.71.36.0 — it will match exactly one station! — in the second route, and will take precedence over it. Figure 4-2 shows how the route matching process works.

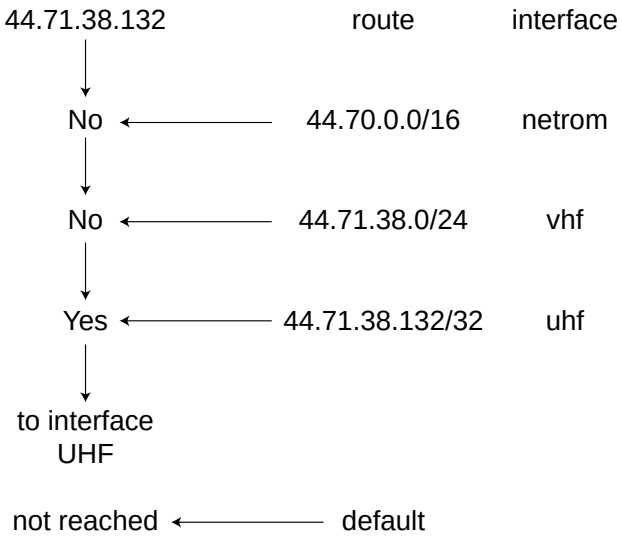


Figure 4-2 - Route matching
44.71.38.0/24 does not match because 44.71.38.132/32 is more specific

Although in this example all the routes line up on byte boundaries (the number of bits is 0 or a multiple of 8), they don't need to. The netmask can be anything from 1 to 32. For example, a route of 44.71.36.0/25 uses the first three bytes of the address, plus the first bit of the fourth byte, as the netmask. This leaves 7 bits of the last byte to identify up to 128 hosts for this network. Figure 4-3 shows how a 25 bit netmask works.

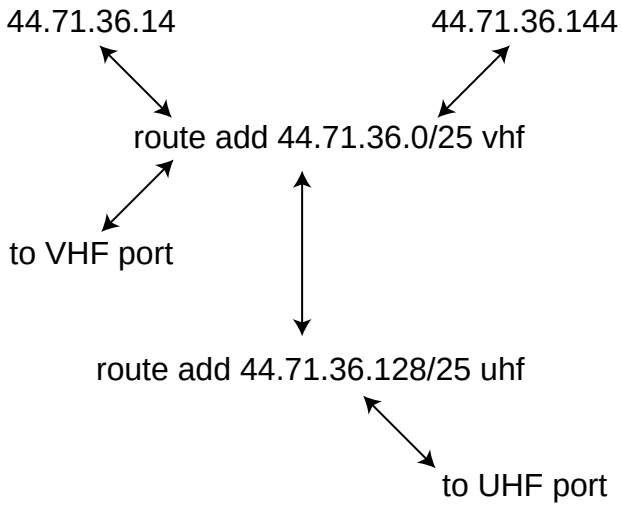


Figure 4-3 - How a 25 bit netmask works

Why would you want to subnet on something other than a byte boundary? The reason is efficiency. Chances are there aren't 253[4-2] ham IP hosts sharing a single channel that you can reach directly from your station, and that there won't be for a while. Assigning a full subnet to that channel probably wastes at least 200 addresses. By using a 25 bit netmask, you can split the subnet into two pieces, each capable of supporting 127 hosts. In a more rural area a 26 bit netmask could provide four networks of 63 hosts each. This kind of subnetting allows much more efficient use of the address space we have available.

[4-2] *Wait a minute. Why only 253 hosts when 8 bits of binary equals 255 decimal? Host address "0" is reserved as a network ID, and host address "255" is used as a broadcast address (discussed later) so it's not available, either.*

This routing mechanism is pretty slick. With proper subnet definitions, we can use a few routing commands to ship IP frames all over the place. However, it's necessary to put some thought into how IP addresses are assigned. For routing purposes, you should consider each local group of stations working on the same frequency and able to contact each other directly (without needing a gateway) to be a local area network, or LAN. Each LAN should be assigned an IP subnet so that a single routing statement can be built to hit that subnet. Figure 4-4 shows the routing process from end to end.

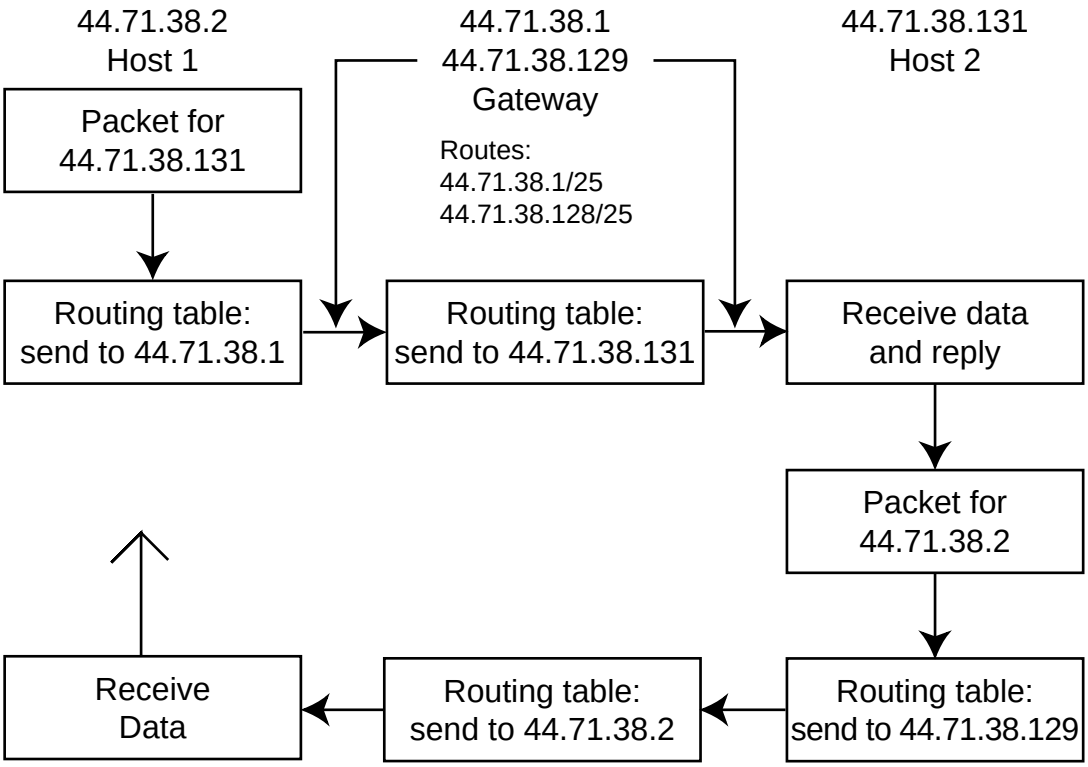


Figure 4-4 - Routing

For example, if an area has two different frequencies carrying IP traffic – two LANs – each one should be assigned its own subnet. That way, a switch can route traffic between them, or to the rest of the network, with only a few routing statements. If addresses are randomly assigned from the same subnet to hosts on both frequencies, the switch and all the users will either need to maintain a routing statement for each and every host, pointing to the interface the switch uses to reach that host, or they will all have to run a dynamic routing protocol. And, unfortunately, there's no dynamic routing protocol in existence yet that works well with networks like ours.

What this means is that we should follow a simple rule when setting up IP subnets: subnets should be based on *connectivity*, not geography. In other words, don't assign a subnet to a county, or to one quadrant of a state. Instead, assign subnets to natural networks — the folks in an area who are all on the same frequency and can all communicate directly with each other. If local activity takes place on two or more frequencies, assign a separate subnet for each. Figure 4-5 illustrates this.

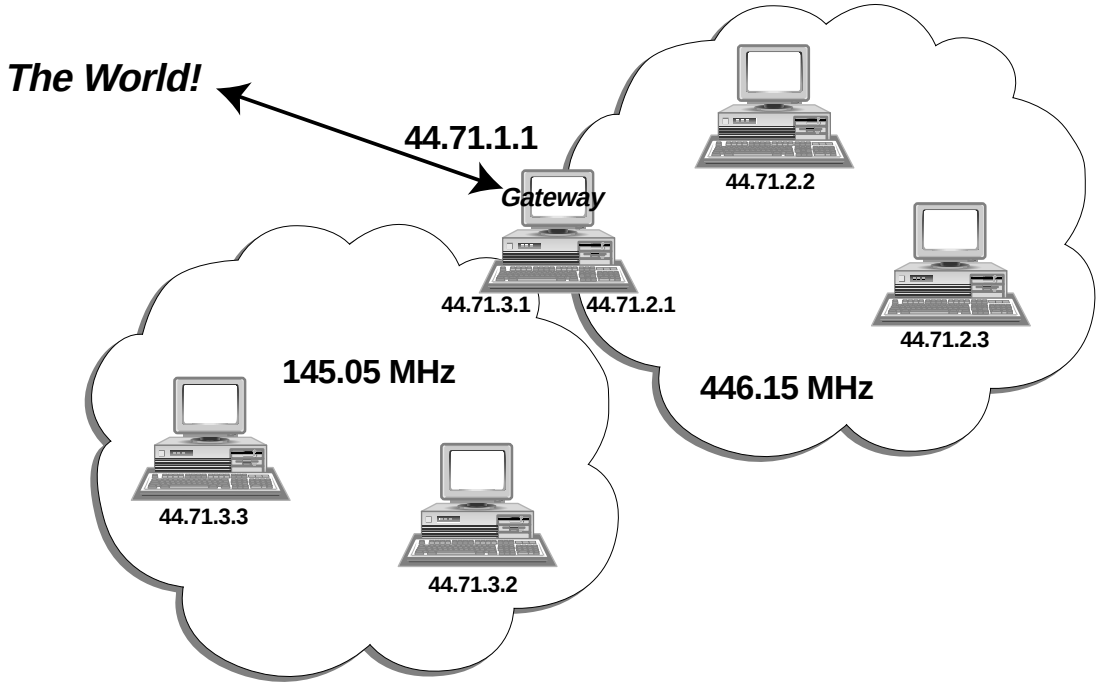


Figure 4-5 - Example subnet. Gateway serving two subnets.

Assuming there's some sort of backbone or wide-area network that connects switches together, with each switch serving a local network, the sensible thing to do is assign subnets around those switches. In other words, if you connect to the rest of the network via switch A, your IP address should be on the subnet assigned to switch A. The fact that you happen to be across a county, or even a state, line from switch A is irrelevant. Radio waves don't pay any attention to borders!

Subnets should be invisible to the outside world; unique routing to deal with subnets should only be required within the network. The same should be true of sub-subnets such as we have on amprnet (remember that the state and country networks are usually subdivided down to local networks). If you are designing a subnet scheme, one of your goals should be to make routing from the outside world as simple as possible.

For example, if we decide to divide the 44.71.36.0 (sub)network into two further subnets (within 25 bit netmasks) to support separate LANs on VHF and UHF channels there should be a single gateway visible to the rest of the network that will handle packets for the full 44.71.36.0/24 range. That gateway will have an interface on each of the LANs, and can route packets from the outside world to either of them. That way, other networks won't have to maintain multiple routes.

ARP: Mapping IP Addresses to Hardware Addresses

You may see a problem with this idea of gateways routing packets. Unlike AX.25 packets that include the callsigns of all the digipeaters along the way, IP frames don't include the addresses of gateways; they only identify the destination address. How do I send a packet addressed to 44.62.32.15 to a local gateway addressed 44.71.36.1? Here's where the distinction between IP addresses and hardware addresses comes into play.

To understand the mechanics of IP routing, you need to know two things: first, a host monitors the channel for packets sent to its hardware address, *not its IP address*. Second, the destination IP address of a packet is set by the originating station, and is not changed by any intermediate gateway.

When a host needs to send a packet, it examines the destination IP address and determines from the routing table whether to send the packet directly to the destination host (because that host is on the local LAN), or to a gateway. In either case, it finds the hardware address of the recipient and uses that address at the data link layer to send the packet to the recipient station. The IP address is used on an end-to-end basis to route packets; hardware addresses are used hop-by-hop to reach the next stop along the way.

How does a host find a hardware address? It's not practical to maintain a huge configuration file with every hardware address to IP address match; you would need to have an entry for every host you might ever want to reach, as well as special rules to deal with gateways. And, callsigns are subject to change; maintaining such a database would be very difficult indeed. There's a pretty clever solution to this problem. Remember that an IP host only needs to worry about getting a packet to the next hop along the line; by definition that has to be a station that the host can talk to directly. That host could have any hardware address; it could be reached via a radio link, or perhaps via an Ethernet segment inside a ham's house. A protocol called **ARP** (**A**ddress **R**esolution **P**rotocol) finds the correct hardware address, whatever the nature of the hardware in use.

ARP is the tool that puts IP addresses together with hardware addresses on a local network. Figure 4-6 shows how it works in the simplest case. *n8ur* wants to send a packet to *ke8tq*. It knows that *ke8tq*'s IP address is 44.71.36.18, and it knows that *ke8tq* is on the local network (because *ke8tq*'s network number is the same as *n8ur*'s), but it hasn't a clue what the hardware address is. So, *n8ur* sends an **arp request** – a packet

addressed to the local network as a whole – using a special broadcast address that all stations listen for.[4-3] My arp request says, in effect, "Hey, I'm looking for 44.71.36.18. Are you out there?" *ke8tq* hears this packet and replies, in a packet addressed to *n8ur*, saying "Here I am. My hardware address is KE8TQ." Now, *n8ur* knows how to reach *ke8tq*, and the dialog can continue.

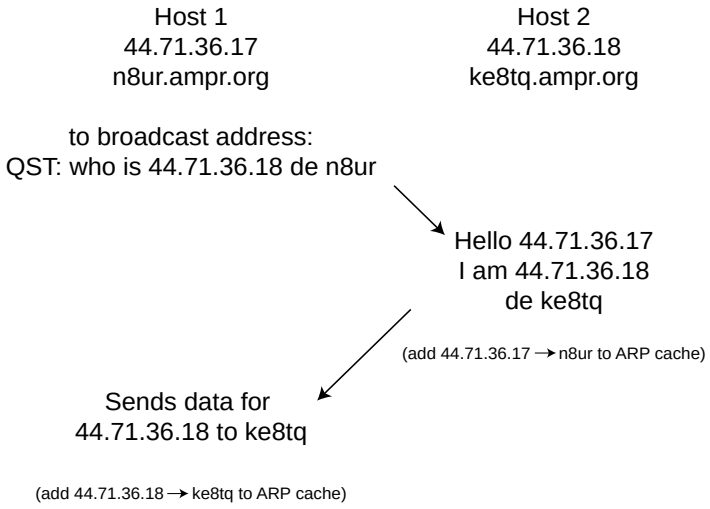


Figure 4-6 - ARP

For efficiency, NOS keeps a cache of recently used IP address/hardware address mappings in memory[4-4], so this process doesn't have to be repeated each time two hosts want to talk to each other. You can also manually add entries to the arp table through AUTOEXEC.NOS or the NOS command line. This is useful to speed things at startup (though you must remember to update these entries if the addresses changes), and is necessary when NOS is communicating via digipeaters or NET/ROM nodes. JNOS and TNOS also offer a feature called **arp eavesdrop** that, if enabled, will pick up ARP requests and replies that are passed between other hosts on the channel and add them to the arp cache. That avoids the initial arp exchange.

[4-3] An arp request is sent to an IP address with a host part that's set to all binary 1s — for an 8 bit host part, that's 255, and (on ham links) a hardware address of QST. These two special addresses indicate that the message is intended for all the hosts on the network.

[4-4] Unlike hostname/IP address maps, IP address/hardware matches are not stored in a disk file – change too often (for example, changing an Ethernet board automatically changes the associated hardware address) to make a disk-based cache worthwhile.

When a packet needs to go through a switch to reach a remote network, the process is similar, as shown in Figure 4-7. When *n8ur* wants to send a packet to a host on another network, say *k1lt* (a host in Columbus, Ohio), its default route tells it that all non-local packets should go through *w8apr*. If it doesn't already have an entry in its arp cache, *n8ur* will send an arp request looking for *w8apr*, will get a response, and will then send its packets for k1lt to the switch. *w8apr* will look at its routing table and find that the next hop on the backbone toward Columbus is *w8cqk*, and will send an arp request on the backbone to find the hardware address for that host. And so on and so forth, until the last switch is ready to deliver the packet to the destination host and does a final round of arp'ing to make the delivery.

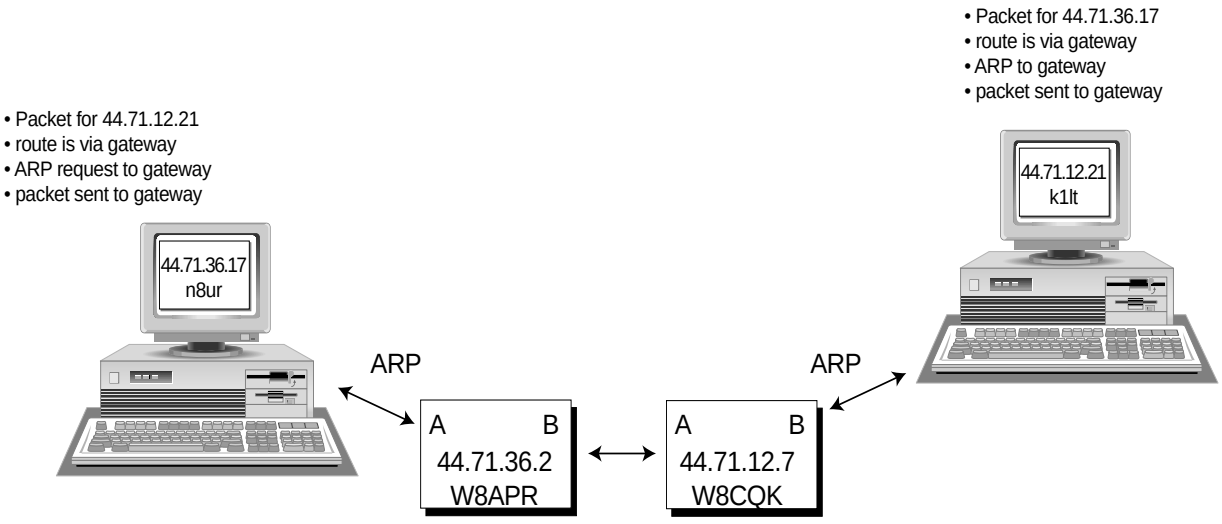


Figure 4-7 - ARP example

Remember this: ARP and hardware addresses are only important within a single network, among machines that talk to each other directly. Hardware addresses are a purely local concern on an IP network. IP addresses span the network end-to-end; hardware addresses only matter hop-by-hop.

Tricks with ARP

There are a couple of ways that ARP can be stretched to handle unique situations. First, think of two stations that are on the same subnet, but which can only talk to each other via a (yechh!) AX.25 digipeater. If W8APR sends out an arp request for K8GKH, the digipeater won't retransmit it (because the request is addressed to QST and isn't sent with a digi string), so K8GKH will never get the request, and will never respond.

To get around this problem, two things must happen: first, NOS must be told the hardware address of the distant host, and second, it must be told to use a digipeater to reach that hardware address. A manual entry to K8GKH's arp table takes care of the first problem:

```
arp add [44.71.36.32] ax25 W8APR vhf
```

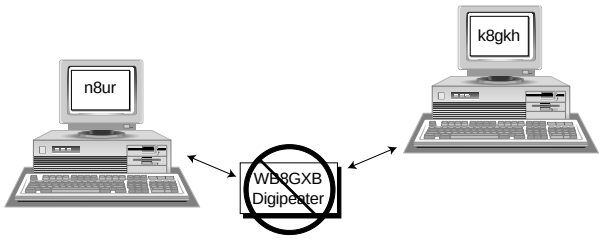
This command has the same effect as an arp reply; it adds the match between [44.71.36.32] and to the host's arp table. Next, the **ax route add** command

```
ax route add N8KZA vhf N8ACV
```

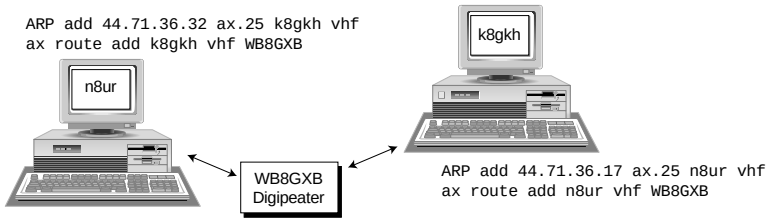
tells NOS that N8ACV is reached on the vhf interface via the N8ACV digipeater (K8GKH will have to add a similar pair of commands at his end). A similar concept works for IP routing over NET/ROM. Figure 4-8.

A second trick comes in handy when it's useful for one host to pretend that it has the hardware address of another. For example, let's say K8GKH has two NOS systems running at home, connected together via an Ethernet as shown in Figure 4-9. One system has radios hooked to it, while the other provides user services like ftp. As the figure shows, *rf-k8gkh* (44.71.36.33, hardware address k8gkh) serves as a gateway for *ftphost* (44.71.36.222), so to reach *ftphost* each user would normally have to add a specific route that identified *rf-k8gkh* as the gateway used to reach *k8gkh*:

```
route add 44.71.36.32 vhf 44.71.36.33
```



Problem: Digipeaters don't pass ARP requests



Solution: Manual ARP add AX Route statements at each end allowing connectivity through the digipeater

Figure 4-8 - Manual ARP

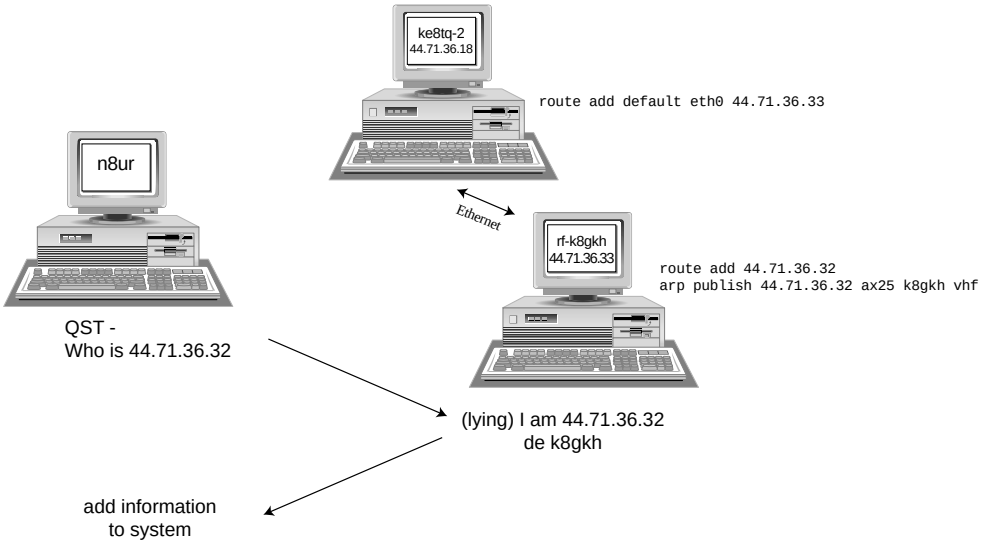


Figure 4-9 - Proxy ARP

It would be nice if users didn't have to add a route to be able to reach a single host who's hiding behind a "front end" like rf-k8gkh. What if *rf-k8gkh*could, for the purposes of arp, pretend that it was *k8gkh*, and respond to arp requests for k8gkh? Then, users wouldn't need a special routing command — they could arp *k8gkh*, and *rf-k8gkh* would reply. The user would then send packets to *rf-k8gkh*, and it would forward them on to *k8gkh* transparently.

The proxy arp (sometimes called **promiscuous arp**) function does this. Issuing the command to rf-k8gkh

```
arp publish 44.71.36.32 ax25 K8GKH vhf
```

instructs that station to respond to any arp request it hears on the vhf interface for k8gkh giving K8GKH as the hardware address. In other words, at the data link layer *rf-k8gkh* pretends to be *k8gkh* and tricks other systems into sending *k8gkh's* data to it. In this example that's a good thing, as it makes the Ethernet between the two systems invisible to the rest of the network. Proxy arp can be used for Evil Purposes (by capturing packets bound for a legitimate host), or it can inadvertently screw up a network (two hosts responding to the same arp request can cause quite a bit of confusion!), so it's best to be very careful when playing this trick.

A Routing Example

Well, that was all about as clear as mud. An example will help show how routing works in practice.

First, let's refresh some definitions. **Hosts** on a **local area network** can all communicate directly with each other. The hosts address each other locally using **hardware addresses**. One (or more) hosts on the network may act as **switches** or **gateways**, meaning that they know how to reach other networks. **IP addresses** contain network information that can help guide packets to the next hop on the way to their final destination. Hosts set up **routes** that use the **netid** and **netmask** to define where packets go.

Let's set up a simple network to use in this discussion. It's shown in Figure 4-10. Our local network is in Dayton, Ohio and has been assigned the network/subnetwork number 44.71.36.0/24. All the hosts connecting to this network have 44.71.36 as the first three bytes of their address; since the netmask is 24 bits (3 bytes) long, the last byte is used to identify each host, and that's the only byte that will be different in each IP address on the network.

However, local activities takes place on both 2 meters and UHF, so we've further divided our subnet by using one more bit to identify the LAN a host is on. Adding that bit means that internally (on the local side of the gateway) the netmask is actually 25 bits.

Let's take a deep breath while figuring out what that means. That 25th bit is the leftmost bit of the fourth byte — the byte we think of as the hostid. If you can count in binary (just pretend you only have two fingers), you'll see that this "subnetmask" bit represents "128." If one subnet is represented by this bit being turned off, that means the hostids on that net can range from 1 ("00000001") to 126 ("01111110"); the other subnet has the bit turned on, so its hosts can be in the range of 129 ("10000001") through 254 ("11111110")[4-5]. For this example, let's say that the first subnet is used on the VHF LAN, while the second subnet is used for UHF.

Our club station, *w8gps*, is on UHF with an IP address of 44.71.36.133. Fred's station, *ke8tq*, is on VHF with an IP address of 44.71.36.18. Our gateway, *w8apr*, is on both VHF and UHF, and has an address on each LAN—44.71.36.1 on 2M, and 44.71.36.129 on UHF. It also has a third interface to another network, the 44.71.1.0 network that connects together the various switches in Ohio on a backbone. Its address on that interface is 44.71.1.9. Let's assume that once a packet hits the 44.71.1.0 network, it can be routed to anywhere in the world by one or more gateway stations on that network (boy, I wish that were true!). For good measure, we'll also include a few other stations to illustrate concepts we've talked about.

[4-5] *Remember, hostids that have "0"s or "1"s in all their bit positions are reserved.*

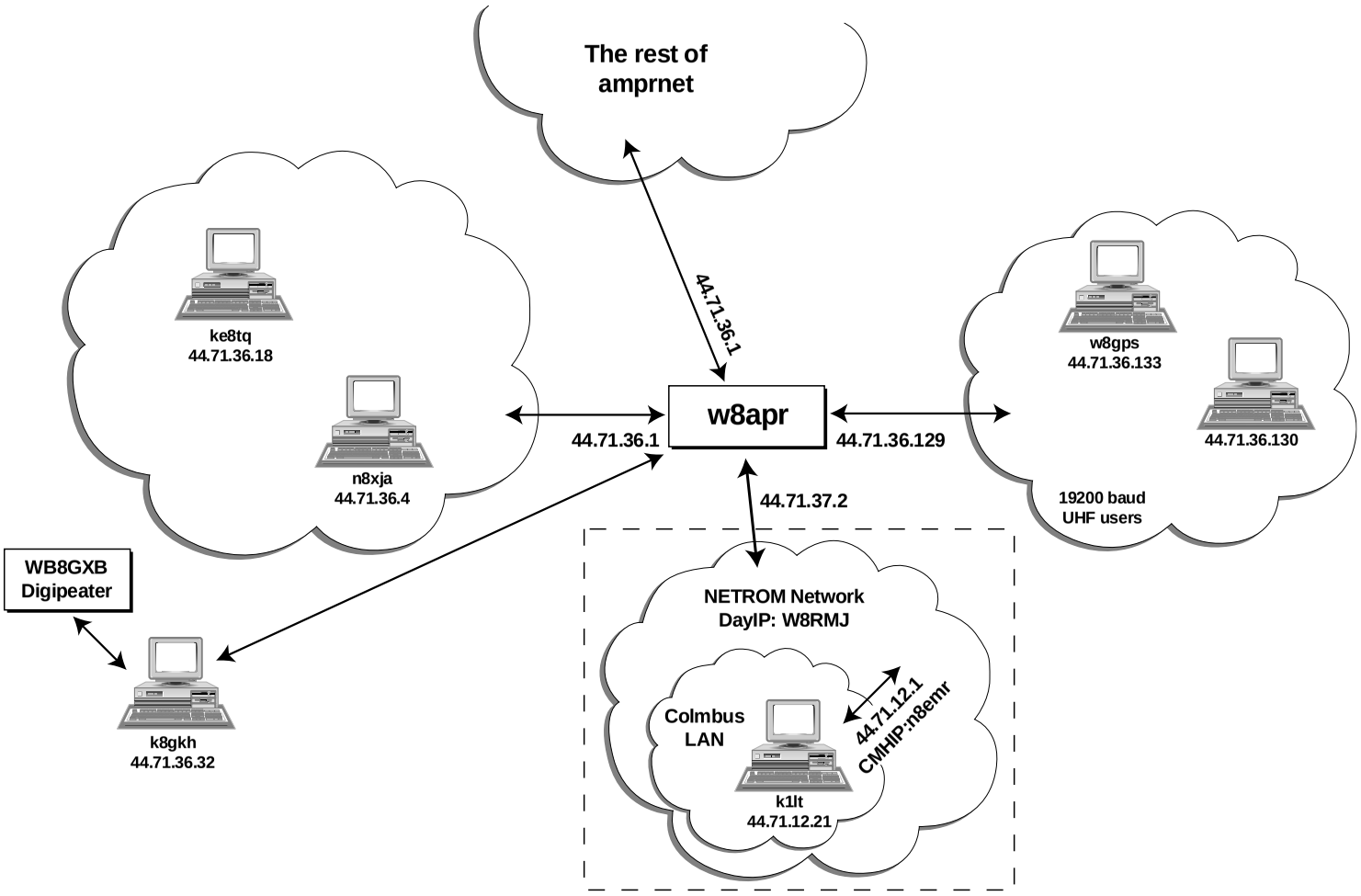


Figure 4-10 - Our Network

n8ur and *ke8tq* each need to have two routes defined: one handles packets going to local hosts, and the other packets going everywhere else. The first route says, "send any packet with a network address in the range of 44.71.36.1 through 44.71.36.127 out via the radio port, and assume that you can reach that host directly," and the second says "send everything else via the radio port to *w8apr*; he'll know what to do with it." This second route is the default route—it's what the system uses when no more specific route exists.

A switch is simply a host with two or more interfaces, and routing commands that ship packets from one interface to another based on IP addresses. So, *w8apr* has routes that are a bit more complex than a host with a single interface. It needs one route to handle traffic on the local network. It needs a route for each switch on the backbone network that feeds another local network. And, it needs one to reach the gateway on the backbone that connects to all those far away places.

Now you know how basic IP routing works. What I've described is called **static routing**, since it relies on manually entered routes that don't automatically change. There are a couple of protocols that do **dynamic routing** that is updated automatically as the network changes — hosts running these protocols send **routing broadcasts** to update each other with connectivity information.

The best known of these are **RIP** and **RSPF**. RIP (**R**outing **I**nformation **P**rotocol) is a standard IP protocol that's widely used in Ethernet LANs. It's included in all NOS implementations (though frequently it's not compiled into distribution binaries). If implemented properly, RIP can be useful in ham networks that span multiple channels. However, few amateur networks have used RIP on a wide scale.

RSPF (**R**adio **S**hortest **P**ath **F**irst) is a protocol designed specifically for the ham radio environment, and it's also available within NOS. Although it has great potential, the current implementation has some serious problems that make it impractical for real-world use. These problems are intimately connected with NOS' design philosophy, and it may be difficult to work around them.

I won't discuss RIP or RSPF further here, though if you're trying to design a TCP/IP network using "best practices," you should look at RIP as a potential part of your plans.

Chapter

5

Installing NOS: The Beginning

Frankly, there's no completely painless way to get NOS running on your computer. That's because there are infinitely many configurations of hardware, software, and services that hams put together for their systems. That means there are a number of custom parameters that you must set to teach the program about your environment and your network. In addition, NOS uses quite a few directories and files, and they all need to be in the right place

But don't give up! You, too, can master the secrets of NOS setup. It's mainly a matter of screwing up your courage to edit a configuration file called **AUTOEXEC.NOS** (older versions may call the file **AUTOEXEC.NET** instead; use whichever name is appropriate for your version). Getting the right commands in the right places in AUTOEXEC.NOS is most of the work; setting up a few other simple configuration files set up will take you the rest of the way to IPville.

Obtaining NOS

The best way to obtain a copy of NOS is via the Internet. The web page at http://www.tapr.org/tcpip has pointers to the homes of TNOS and JNOS, as well as other information[5-1]. When you get to the archive site, you'll probably find several versions to download. Which one(s) do you need ?

[5-1] What if you don't have Internet access ? TAPR offers versions of JNOS and TNOS on CDROM. Contact TAPR for details. http://www.tapr.org

JNOS changes version numbers relatively slowly. Version 1.10 lasted for several years, and 1.11A is current as I write this. Incremental updates are identified by a letter tacked on to the version number. There are also beta releases which include the letter "X" followed by a number (for example JN112X1.EXE is beta release 1 of the upcoming version 1.12). Look for the highest number and letter version at the site, but you should probably avoid the "X" beta versions until you've been around the block once or twice.

When you find the version you want, you will probably see three files with names like "JNOS11A.EXE", "JNOS11A.ZIP", and "DOCS11A.ZIP". Normally, the ".EXE" file is the executable JNOS program, while the ".ZIP" file is a PK-zipped archive of the source code. Documentation and other supporting materials are packaged separately in the "DOC111A.ZIP" archive; the documentation package also includes the files necessary to enable the on-line help system. You should get the ".EXE" and documentation files to begin with. There may be additional files available that contain auxiliary programs and tools; you might want to grab those as well.

TNOS follows a somewhat similar scheme, but beta versions aren't made generally available. Huge changes are signified by a change in the first digit of the version number, major changes by a change to the second digit, and interim releases (usually to fix a significant problem, but not adding functionality) by change in the third digit. There are separate directory trees at TNOS ftp sites for the UNIX and DOS versions of the program. The TNOS DOS directory contains source code in zipped file, and an executable version and supporting files in a self-extracting .EXE file. There will probably be a separate documentation file at the ftp site, but the TNOS230.EXE file includes basic documentation and sample setup files.

If you're working with JNOS, put the executable file somewhere in your DOS PATH and run the file (you'll probably want to remove it to something shorter, like JNOS.EXE). If you're using TNOS, extract the files from your root directory using a program like PKUNZIP with the "-d" command to create subdirectories. Either way, you'll end up

with all the necessary directories created for a default installation. TNOS will also install sample configuration files which you'll need to edit, while JNOS leaves you to create your own files.

To see if everything is properly installed, make sure the executable program is in your PATH, change to the root directory,and type the file name at the DOS prompt. You should see a startup screen and probably a bunch of error messages. That's okay for now. Finally, you should see a prompt like `nos>` on your screen. Congratulations; it works! To leave NOS and return to DOS, type `exit` and then hit the enter key.

Once you have the NOS program installed and the directory structure created, you're ready to start editing the configiration files.

Do I Need to be a Programmer to Configure NOS?

If you follow the messages on packet radio or in Usenet newsgroups about NOS, you may think that compiling NOS from source code is a necessary part of configuration. It's not. Although all the popular varieties of NOS are available in source code form and you can compile them yourself, you probably don't need to do that, because they are also available pre-compiled for download via Internet or BBS. Appendix C lists some of the places you can find current versions.

These executable files typically come in two or three flavors, each suited for a different set of user needs. One may be designed for an end-user station, while another may include the complete PBBS mailbox code. The chances are that one of them will work for you. If you're planning to run a typical NOS user station with an external TNC and no unique services, a precompiled NOS will almost certainly work just fine. If you want to "roll your own" after you get your feet wet, Chapter __ will show you how.

Files and Directories

NOS uses a bunch of files and directories for its configuration and working data. There used to be only a few, but with the addition of features, and particularly increased BBS functionality, the list of files has mushroomed.

Current versions of JNOS will create their own directory tree when you run them for the first time. If you want to set things up manually, you should create the following directories on your disk:

```
spool\              (holds NOS' working files)
spool\help\         (help files for the mailbox)
spool\mail\         (mail messages go here)
spool\mqueue\       (mail workfiles)
spool\rqueue\       (used by external mailer programs)
finger\             (home for finger info files)
public\             (file uploads/downloads)
```

By default, the JNOS directory tree must be under the root directory of your default disk (*e.g.*, under C:\), and NOS will look for its primary configuration files in the root directory (*e.g.*, C::\AUTOEXEC.NOS). You may want to put all your JNOS files and directories in a single directory (perhaps \nos) to keep you disk more orderly. You can do that in two ways.

If you start JNOS using the **-r** switch you can specify a new root directory: **nos -r c:\nos** will cause NOS to look for its configuration files and directory tree under the \nos directory. Under TNOS, the command line switch **-d** does the same thing (*e.g.*, **tnos -d c:\nos)**.

Alternatively, you can use the DOS **assign** command to assign a drive letter to the NOS directory, and then switch to that new drive letter to run the program. Either way works; I prefer to use the JNOS command line switch.

Recent versions of TNOS use a somewhat different directory structure than JNOS:

```
etc/              (config files)
etc/help/         (Mailbox help file directory)
finger/           (Finger info directory)
public/           (Public directory)
spool/            (Spool directory)
spool/log/        (log directory)
spool/mail/       (Incoming mail)
spool/mqueue/     (Outgoing mail spool)
spool/rqueue/     (queue for router)
spool/signatur/   (Mail signature file directory)
spool/stats/      (Statistics directory)
```

Appendix K includes a complete TNOS file list, including some additional directories that are required by advanced TNOS features. You don't need to manually create these directories; the TNOS installation program will create the required ones for you. In JNOS and earlier versions of TNOS, the main configuration files need to go in the top-level directory of your default disk (either the root directory, or the one you've identified with a command-line switch):

```
AUTOEXEC.NOS  (the NOS configuration file)
FTPUSERS[5-2]    (optional; user ftp/mbox access)
DOMAIN.TXT    (hostnames)
BM.RC         (optional; mailer configuration)
ALIAS         (optional; used by smtp and the BM mailer program)
```

With TNOS' recent change in directory structure, the ALIAS and FTPUSERS files have moved into the etc directory, and DOMAIN.TXT now goes in the spool directory.

[5-2] *In version 2.30, TNOS renamed the FTPUSERS file to SECURITY. The file format is unchanged. For this book, I'll continue to call it FTPUSERS.*

NOS uses one executable file, and optionally a standalone mail program (the most common one is called BM, and it's included with some NOS distributions. Another common mailer program is call **PCELM**. These files can be installed anywhere on your file path:

```
NET.EXE or NOS.EXE      (main executable file)
BM.EXE or PCELM.EXE     (mailer program)
```

An Introduction to AUTOEXEC.NOS

`AUTOEXEC.NOS` is the key to making NOS work. It can range from fairly simple to very complicated, depending on the combination of hardware and services you need. Although both JNOS and TNOS now have tools available (described later in this chapter) to automatically generate the `AUTOEXEC.NOS` file for you, understanding the structure of this master configuration file is worth the effort – the automated tools aren't perfect, and you'll probably need to dig into the file sooner or later to customize things for your installation. Bear with me while we walk through `AUTOEXEC.NOS`.

Commands in AUTOEXEC.NOS fall into seven broad categories:

- Basic commands to define internal parameters like memory usage.
- System wide commands.
- Global protocol-specific commands, subgrouped into commands for AX.25, IP, TCP, UDP, and NET/ROM.
- Hardware-specific commands that define and configure each interface.
- Server-related commands that start and configure the various services NOS has to offer.
- Local configuration commands
- Comments

The order of commands is important, and you may get unexpected results if things are out of order. It's generally best to put the NOS internal commands first, followed by basic information like your callsign, primary IP address, domain, etc. Then, set the

default protocol parameters. Next, start those servers that don't require info about specific interfaces (that's most of them) and set their parameters. After that, attach the hardware interfaces and set any protocol parameters on a per-interface basis. Finally, start the remaining servers and include commands that are tied to specific interfaces. Comments begin with the # character and may be included anywhere in any NOS configuration file; NOS will treat anything from the # character to the end of line as a comment. Although you can put comments on the same line as a command in `AUTOEXEC.NOS`, I don't recommend it – there have been reports that doing so can cause problems. It's best to put commands on separate lines.

Global versus Per-Interface Commands

It's the second-to-last rule that catches people unaware, and a brief digression here may save you lots of head-scratching later. NOS gives very precise control over things like packet size, retry interval, and all the other low-level stuff that makes packet radio work efficiently. These parameters all have built-in defaults that may or may not be reasonable in your case. You can override the defaults with a command like **`ax25 paclen 128`** (which sets the AX.25 "paclen" parameter to 128). This is a global command that affects all the interfaces on the system; if you enter the command as shown, it will update the global paclen setting for all the radio interfaces on the system.

You can also specify a different paclen for each interface. For example, the command **`ifconfig vhf ax25 paclen 256`** will set paclen for the interface named vhf to 256, while leaving the other ports' paclen settings alone.

The catch is that the global setting will only "stick" to interfaces configured after the global value is set. In other words, you need to set all global parameters before you attach interfaces that use those parameters. That means that your AUTOEXEC.NOS file should be structured with the global settings appearing ahead of the interface configuration commands, which should be as close to the end of the file as possible.[5-3]

[5-3] *It also means that issuing a global parameter change from the NOS command prompt won't work. If you want to change things on the fly, you need to make the change for each interface separately by using the ifconfig command.*

Command Structure

NOS has lots of commands, and many of them have several layers of subcommands. Most can be issued either in AUTOEXEC.NOS, or from the NOS command line. To see what commands are valid in your version of NOS, type a **?** at the NOS> prompt. To see the subcommands for a command type the command name followed by a question mark: **ax25** (you can use the same process to see subcommands of subcommands **ax25 timertype**). TNOS and JNOS also offer a help command that will (if the help files are installed) display a screenful of information about a command. To use it, enter **help <command name>** at the NOS prompt.

Some commands can be abbreviated to save typing; the degree of abbreviation allowed depends on the command set of the NOS version you're using. Experimentation is the best way to see what works and what doesn't. One minor annoyance in some older NOS versions is that command line input is case sensitive, and commands must be entered in lower case—"ip hostname n8ur" is fine, but "IP Hostname n8ur" isn't. That's a byproduct of TCP/IP and NOS having strong ties to the UNIX world, which is case-sensitive and where lower case command and file names are the norm. In fact, many of the commands used in NOS are the same as their UNIX counterparts, and by learning NOS you're starting on the way to learn UNIX! The bottom line is that it's safest to do all your NOS commands and keyboarding in lower case.

Another potential "gotcha" is that the format of commands that turn features on and off isn't consistent. In some older versions, a command like **ax25 hport <interface>** (don't worry about what that means!) toggles the state of the command—if it was on, entering the command will turn it off, and vice versa. In other versions, entering **ax25 hport** will display the current status of the command, but not change it. To do that, you'd enter **ax25 hport <iface> on** or **ax hport <iface> off**.

NOS Callsigns

Originally, NOS systems used and responded to a single amateur callsign. As services and protocols were added, a deficiency in the basic AX.25 protocol caused a problem: NOS had no way of knowing whether an incoming packet requesting a new session was an IP frame that should be passed to the IP layer, an AX.25 frame that should go directly to the mailbox, or a NetROM frame that should be passed to the NetROM subsystem. Only when a second packet was received could NOS tell what the data protocol was.

This meant that an AX.25 user connecting to a NOS system didn't get any response (other than her TNC's "connect" message) until she hit the return key to send another packet. Then, NOS could tell that the data was coming from an AX.25 user, and fire up the mailbox greeting and prompt. As JNOS became more commonly used as a PBBS system, this became a real pain, so WG7J introduced the idea of using a separate callsign (or the same callsign with a separate SSID number) for AX.25 connections.

In other words, I might use N8UR-5 as my "ax25 mycall" but N8UR as the "ax25 bbscall". Users connecting to N8UR would go right to the mailbox prompt without having to hit a carriage return first.

It turns out this idea works very well, and it has mushroomed in JNOS and TNOS to the point where you can identify many callsign/SSIDs for a single system. The "SSID" is the **S**econdary **S**tation **ID** that follows the callsign – in "N8UR-5", the SSID is 5. Values from 1 through 15 are valid.

The calls and aliases supported under JNOS (as of version 1.11a) are:

```
ax25 mycall        # link address for all ports
ax25 alias         # ax25 or netrom alias
ax25 bbscall       # like the alias; connects you
                   # to the mbox without hitting CR
ax25 ttycall       # immediate connect to
                   # split-screen session
ifconfig <iface> linkaddr    # unique address for a
                             # port to use for digipeating;
                             # by default, same as
                             # ax25 mycall
ifconfig <iface> cdigi  # unique address just for
                        # cross- port digi – no connects
                        # allowed to this call
netrom alias       # same as ax25 alias
netrom call        # callsign for netrom
```

TNOS offers even more callsigns to reach various servers. As of version 2.30, they are:

```
ax25 mycall        # link address for all ports
ax25 alias         # ax25 or netrom alias
ax25 ttycall       # immediate connect to
                   # split-screen session
ax25 infocall      # call for info server
ax25 newcall       # call for news server
ax25 tutorcall     # call for tutorial server
ax25 rosecall      # call for ROSE switch
ax25 user          # call used for outbound ax25 connections
convers call       # callsign for convers server
forward mycall     # callsign for bbs forwarding
ifconfig <iface> linkaddr    # unique address for a port
                             # to use for digipeating; by
                             # default, same as ax25 mycall
netrom alias       # same as ax25 alias
netrom call        # callsign for netrom
```

You don't need to explicitly set all these calls in your system, but there are a couple that are important. If you are using the PBBS (mailbox) system in NOS, you particularly need to make sure that the **`ax25 bbscall`** (JNOS) or **`forward mycall`** (TNOS) callsign/SSID is different than the **`ax25 mycall`**. Your callsign configuration is one of the first things to check if users report difficulty connecting to your mailbox or other non-IP service.

A Basic AUTOEXEC.NOS File

Now that we've gone through the theory, let's get our hands dirty and build an AUTOEXEC.NOS file. Appendix C includes a basic, but workable, file skeleton, and in this chapter we'll fill in the blanks. When we're finished, you'll have a minimal AUTOEXEC.NOS to get started with. These are the entries you'll have to customize.

Your hostname in FQDN format (assigned when you get an IP address from your local coordinator), your assigned IP address, and your callsign (optionally including an SSID, but local customs vary on this):

```
hostname n8ur.ampr.org.
ip address 44.71.36.17
ax25 mycall N8UR
```

If you have a domain name server, add a command near the beginning of your configuration file identifying its IP address:

```
domain addserver 44.71.36.133
```

NOS uses a command called **`attach`** to configure your serial port or other hardware and link it to a named **`interface`**. These commands include specific hardware configuration information like I/O ports and interrupt numbers and can get quite hairy; see the next chapter for many more details. For a TNC in standard KISS mode on COM 1 with a 4800 baud serial port speed, use:

```
attach asy 0x3f8 4 ax25 vhf 1024 256 4800
```

The "vhf" in the middle of the command is the interface name—you use it as a "handle" to identify this port to NOS when you set up routing commands and the like. You can use any (short) name you'd like. Many people use names like "ax0" and "ax1," which mimic the device names used by UNIX systems, but I prefer to use an interface name that has some relationship to the interface's use, like "vhf" or "uhf".

You'll also need at least one routing command, and possibly a second one. If you have no switches available to reach other subnets and all your local activity is on a single frequency, then all you need is this (assuming your interface is named "**vhf**"):

```
route add default vhf
```

If you have a switch or gateway that routes packets to other networks, you should add the IP address of the switch to the end of the default route command:

```
route add default vhf 44.71.36.2
```

This command tells NOS to route all packets to the switch at the indicated IP address[5-4], which is reachable via the vhf interface. If you have a gateway route like this, you'll need a second route for the local hosts you can directly reach:

```
route add 44.71.36.0/24 vhf
```

You'll recall that the "/24" refers to the leftmost 24 bits (3 bytes) of the address are the netid that should be checked for routing information. This route tells NOS to send all packets with a destination IP address that has "44.71.36" as its first three bytes (in other words, hosts that you can reach directly) straight out the vhf interface without going through a gateway.

Finally, start some servers; the list will be determined by the services you want to offer. You'll probably want to start the **smtp** server so you can receive mail, and you also need to set the **smtp timer** so that your system will send out mail that you create.

[5-4] *It's best to refer to other hosts in AUTOEXEC.NOS by IP address rather than hostname. That way, you'll be able to reach them even if you don't have a DOMAIN.TXT file or domain name server available at startup.*

The **smtp quiet** command determines whether the console bell will ring when new mail arrives:

```
start smtp
smtp timer 600
smtp quiet on
```

You'll probably want to let people connect to your system to leave you messages using the mailbox, so you'll want to set a callsign that hooks directly to the mailbox. As discussed above, you need a different call/SSID for this because the "ax25 mycall" is the general callsign for your system, and connections to it won't get the mailbox prompt until a second packet arrives. Since this is a TCP/IP system, after all, you'll also want people to be able to reach the mailbox via telnet, so start that server, too:

```
ax25 bbscall N8UR-3    # JNOS only
forward mycall N8UR-3  # TNOS only
start ax25
start telnet
```

If you want people to be able to "beep" you with the "O"perator paging command, issue the command:

```
mbox attend on[5-5]
```

If you want folks to be able to connect directly to you in a split-screen chat mode, start the **ttylink** server, and assign a callsign/SSID for users to use to make such a connection:

```
ax25 ttycall N8UR-4
start ttylink
```

[5-5] *In TNOS version 2.30 and later, the "mbox" command has been changed to "pbbs".*

If you've set up a `public/` directory for ftp file upload and download, and set up permissions in `FTPUSERS` (more on that later), you will want to start the ftp server:

```
start ftp
```

If you've configured a `finger/` directory and created finger files, start that server:

```
start finger
```

Starting the echo server will let people use the ping command to see if you're alive:

```
start echo
```

These are the basic commands to configure your NOS system. Appendix C is a very detailed `AUTOEXEC.NOS` template from N5KNX; it describes in detail how you might configure a full-blown system, and is full of comments to explain things.

Fast Relief for Headache Sufferers

Versions of TNOS 2.02 and later include a configuration utility (`MKCONFIG.TCL`) that will create a pretty reasonable `AUTOEXEC.NOS` file for you, based on your answers to a few configuration questions. It's included with the TNOS distribution package. You may still want to tweak the file, but it will give you a good starting point Generating a new configuration file with `MKCONFIG.TCL` when you get a new version of TNOS is a good idea to ensure that you've kept up wtih changes in command names or syntax. There are also "plug and play" kits for JNOS. There have been a couple of installation programs written for JNOS. One that is relatively current is `INSTJNOS` by David Fraser, ZL3AI. It is available from the ftp.tapr.org site, and works with JNOS version 1.11.

Chapter

6

Installing NOS: Hardware

NOS supports a number of versions of the **attach** command to deal with different communications hardware. I'm discussing them here because you use these commands almost exclusively in the AUTOEXEC.NOS file, and they are without a doubt the most cryptic commands in the NOS repertoire. Although NOS supports a number of different hardware devices, each of which has its own attach syntax, only a few of the most common ones are included in standard NOS compiles.

This chapter will describe the most common use of the attach command; configuring a serial port to talk to a TNC, as well as two other commonly used hardware drivers, and how to configure them. As usual, this discussion covers the basics; see the NOS reference manual for details on all the many options that the attach command can use.

The most common use of NOS is probably to attach to a standard TNC via a serial port. The **asy** version of the attach command does this. The syntax is:

```
attach asy <ioaddr> <vector> <mode> <if> <bufsize> <mtu> <speed>
```

In English, these parameters are:

`ioaddr` the address of the COM port being used. COM1 is usually **0x3f8** and COM2 is usually **0x2f8**. COM3 and COM4 usually use **0x3e8** and **0x2e8** respectively, but these aren't completely standardized; if you use one of these ports you should check the documentation for your serial card to be sure.[6-1]

`vector` the IRQ used by the hardware. COM1 and COM3 are usually **4**, and COM2 and COM4 are usually **3**. Again, COM3 and COM4 may vary, and you normally won't be able to use two COM ports that share the same interrupt at the same time. In other words, don't try putting TNCs on COM1 and COM3 if both ports use IRQ 4.

`mode` this specifies the nature of the interface. **ax25** is for a connection to a KISS TNC, **slip** is for a hardwired connection to another host, **ppp** is for a dial-up connection, and **nrs** is for attaching a NOS station to a NET/ROM node stack.

`if` the interface name. Any short name can be used. In the UNIX world, interface names are usually like **ax0** — two letters identifying the interface type, followed by one digit identifying the specific interface. I prefer something that has more meaning, like **vhf** for a radio interface on 2 meters.

`bufsize` the buffer for incoming data, in bytes. A value of **1024** is usually more than sufficient for a 1200 baud channel. The bufsize should always be at least as large as, or larger than, the **mtu** (discussed next).

`mtu` the maximum transmission unit size, in bytes. For 1200 baud channels, or any routes that may travel over a digipeater or NET/ROM, a value of 256 is appropriate. The mysteries of **mtu** are discussed in detail in Chapter 8.

`speed` the speed of the serial (not radio) link, in baud. The best setting for this will depend on the speed of your computer, but generally two to four times the radio speed is adequate.

[6-1] *If you are using JNOS or TNOS under Linux, you don't specify the physical hardware address or interrupt number. Instead, give the device name instead of the address, and use a "-" character instead of an interrupt. For example:* `attach asy /dev/ttyS1 - ax25 vhf 1024 256 4800`.

Some sample **attach asy** commands are:

```
# COM1, KISS TNC as vhf, MTU 256, 4800 BAUD
attach asy 0x3f8 4 ax25 vhf 1024 256 4800

# COM2, KISS TNC as uhf, MTU 128, 2400 BAUD
attach asy 0x2f8 3 ax25 uhf 1024 128 2400
```

The most common use for a serial port in NOS is to drive a TNC in KISS mode, but COM ports can be used for several other purposes as well. NOS can talk to NET/ROM stacks and multi-port TNCs via the serial port, can interface to a modem or dumb terminal, and can talk to two other Internet protocols, **SLIP** and **PPP**. The SLIP protocol is worth a few more words here. SLIP stands for **S**erial **L**ine **I**nternet **P**rotocol and it can be used to hook two TCP/IP hosts together via their serial ports. If you want to put another NOS computer on your home network, you can do it with SLIP. And, together with proxy arp (which we talked about in Chapter 4), you can build a radio "front end" to a computer that runs TCP/IP software under Windows or Linux.

To configure a serial port to use SLIP, use a command like this:

```
# SLIP link, COM1 as sl0, MTU 256, 9600 BAUD
attach asy 0x3f8 4 slip sl0 1024 256 9600
```

There's a "gotcha" to watch for when using SLIP links: to improve throughput over slow links, some systems use a technique called "Van Jacobson header compression" which reduces the header overhead. NOS supports this feature, and you enable it by adding the letter "**v**" to the end of the SLIP attach statement. If you aren't able to establish communication with the other end of a SLIP link, make sure that both ends either are, or aren't, using header compression – a mismatch will cause the link to fail.

Some TNC Issues

If you're using a TNC to interface NOS to your radio, there are a couple of things you should consider. Most of them relate to making the serial port link between the computer and NOS work at its best.

The big issue is **latency**. This is the time that it takes for data to travel over the serial link between the computer and the TNC. To minimize latency, the serial link should run at least twice as fast as the radio link, and preferably four times the radio speed. At 1200 baud, this is no problem, but at higher speeds it can be difficult to make the serial port keep up with baud rates you will want to use. Many TNCs only support serial rates up to 9600 baud. Although NOS can support serial ports at up to 115,200 baud, your system may not be able to reliably drive the port at this rate.

In particular, if your computer has common serial port hardware that uses an 8250 or 16450 UART chip (that's the device on your serial port card that takes parallel data from the computer's bus and turns it into a serial data stream, and vice versa), you may find NOS can't reliably run the serial port at a baud rate of more than 9600 or even 4800 baud. You can determine whether your serial ports use an 8250 or 16450 UART by running a diagnostic program such as MSD.EXE (which is included with Microsoft Windows). Or, you can use the **asystat** command in NOS, which will tell you information about your attached serial ports, including their UART type. If your PC has serial ports that use these older chips, you can't get any better bang-for-the-buck than replacing them with a serial board that uses the 16550 or other buffered UART chip.

The 16550 is a $10 drop-in replacement for the more common 8250 or 16450 UART, and it supports a first-in, first-out ("FIFO") buffer than makes a *dramatic* improvement in serial port performance under NOS. NOS will automatically detect and support the '550 UART. A '286 class PC with one of the other UART chips may have trouble talking to the TNC at more than 4800 baud. With a '550 installed, the same machine should handle a speed of 19,200 baud or greater.

Although NOS will automatically detect and support the ‘550 UART, you can force it to do so, or tune the performance (by adjusting the “high water” mark at which the chip sends an “I’ve got data” interrupt to the processor) by adding “**f <hiwater>”** to the end of the attach statement for that interface. **<hiwater>** can be a value of 1, 4, 8, or 14. The default value is 4; I’ve found that on a busy system a value of 8 may provide better throughput.

The original 16550 chip has a bug that keeps it from working properly. There aren't many of those chips around, but if you're buying a serial board or chip, you should make sure that the chip number has a letter suffix. In particular, seeing "16550AFN" stamped on the chip is a good sign. Many serial cards today use ASIC chips that include UART functionality. Surprisingly, many of them do not support the '550 buffers. It's best to check the box and documentation for reference to "high speed" or – better yet – "16550 emulation." Another popular UART chip is the 16650; it has even more buffering than the '550, and will work fine with NOS.

Next, if you’re running at a radio speed of 9600 baud or faster, consider modifying your TNC for better performance. TNC-2 clones normally operate at a CPU clock rate of just under 5Mhz; by doubling the crystal frequency and installing higher speed CPU components, the TNC can run at 10Mhz, offering better performance and an additional bonus—a doubling of the indicated baud rates of both the radio and the serial ports. This means that if your TNC could only operate at 9600 baud on the serial port before, after modification it will support 19,200 baud. As I write this, PacComm offers a 10Mhz CPU upgrade kit for about $25 that will work in PacComm Tiny-2 TNCs as well as other TNC-2 clones.

There’s also another trick to improve the performance of a TNC-2 clone for high speed operation. The GRAPES folks (developers of the WA4DSY 56kB modem) wrote a special KISS-only EPROM for TNC-2s that is optimized for high speed operation. It can (just barely) support a radio port speed of 56kB in a 10Mhz TNC-2, and it offers better timer resolution—for example, you can set the TXDelay in 2.5 millisecond increments, rather

than the 10ms that's normal. This is useful in high speed systems where TXD settings become critical to performance. We use KISS56 ROMs with TNC-2 clones in our local 19.2kB network, and they work very well. The KISS56 firmware code is available from several ftp sites.

One last problem with KISS TNCs is detecting errors on the serial link. These are usually rare, but the original KISS protocol doesn't support any sort of error checking over the link between the computer and the TNC. Such errors can cause strange problems that are hard to diagnose. There are extended versions of KISS available that address this problem (notably, one version written by G8BPQ and available with his software distribution), and at least some distributions of NOS have drivers to support them. You'll need to check your NOS version to see which, if any, of these extended KISS protocols it includes.

Setting TNC Parameters

KISS TNCs set their default values for TXDelay, Ppersist, and SlotTime at bootup, and the PI card driver (discussed later) does the same. These defaults may not be appropriate for your network, so xNOS lets you set your own values for them, as well as several other parameters.

The asy driver allows you to update these defaults through the **param** command.. The driver for the Ottawa PI card provides similar support as well. By using the command

```
param <iface> <param_id> value
```

you can set things like TXD, slottime, and ppersist. "Param_id" may be either a number, or in recent versions of xNOS a name like "txd." For example, issuing the command **`param vhf txd 30`** will tell the TNC attached to the vhf interface to set its TX delay to

a value of 30 (300 milliseconds). Remember that the update happens when the param command is issued; if the TNC isn't on-line then, it won't get the word to change its values. Since parameters are normally set in AUTOEXEC.NOS, it's best to make sure the TNC is powered up and initialized before you start NOS — that way you'll be sure NOS is able to update the TNC with the correct values at startup.

For KISS TNC's, JNOS and TNOS support these parameter types:

Name	Number	Name	Number
TxDelay	1	Group	12
Persist	2	Idle	13
Slottime	3	Min	14
TxTail	4	MaxKey	15
Fulldup	5	Wait	16
Hardware	6	Down	129
TxMute	7	Up	130
DTR	8	Blind	131
RTS	9	Return2	254
Speed	10	Return	255
EndDelay	11		

The **param** command can also be used to configure serial port handshaking via the RTS and DTR parameters. If you have trouble getting your TNC to communicate with xNOS, try issuing param commands with the RTS and DTR values set to either 1 (handshaking on) or 0 (handshaking off).

There's also another use for the param command: the command **param <interface> 255 0** will drop the TNC out of KISS mode.

NOS and Other Hardware

NOS supports other types of hardware. I'll describe the **attach** commands for two of them: the "packet driver" interface that's used with a variety of network cards and the Ottawa PI high-speed packet interface card.

The Packet Driver

The **packet** interface is used to connect to Ethernet cards and other hardware that supports the FTP, Inc. "packet driver" standard. This is a useful setup because packet drivers are available for nearly every commonly used network interface card.

The packet driver is a **T**erminate-and-**S**tay-**R**esident (**TSR**) program for DOS. When you run the program from the command line or a batch file, it loads into memory and then returns you to the command prompt.

The driver talks to a hardware device like an Ethernet card.[6-2] You use command line parameters to configure things like the hardware address and interrupt number that the card is using. These parameters are different for each driver, so you'll need to look at the packet driver documentation file for details specific to your hardware.

The DOS command invoking the packet driver must also set a **software interrupt** number that's used to identify the driver in the computer's memory map, and to establish the link between the driver and NOS. If you have more than one packet driver loaded at once, you need to assign a unique software interrupt to each. The full command line to load a packet driver can be lengthy, so avoid typing it all out each time you load the driver by including it in a batch file (the AUTOEXEC.BAT file is probably best because you will want to load the driver only once; you may as well do it when you boot the system).

Most of the networking cards on the market have packet drivers available. There are also packet drivers that provide a SLIP interface through a serial port, and a way to

[6-2] *A packet driver can also provide an interface to another software program.*

interface NOS via a packet driver with the G8BPQ packet switch software running on the same machine. There's also a packet driver for the PI card which we've found may work better than the built-in pi driver discussed later. In other words, the packet driver interface gives NOS great hardware flexibility. There's a complete set of drivers from Crynwyr Software that's available via ftp from many sites.

The syntax to attach a packet driver interface in NOS is:

```
attach packet <software interrupt> <if> <maxqueue> <mtu>
```

Remember that the software interrupt you supply here needs to match the one you assigned to the packet driver when you loaded it into memory. A common value for the software interrupt of the first (or only) packet driver is 0x60. **maxqueue** is the number of packets (not bytes) that may be outstanding; values of 2 or 3 are typical. For Ethernet, the standard **mtu** is 1500; for other types of hardware, use a value that makes sense for the speed and reliability of the channel.

If you use the NOS packet driver, you'll need to load the appropriate packet driver TSR before starting NOS. The syntax varies from driver to driver, but in all cases you need to make sure the software interrupt parameter for the driver matches the one set in the NOS attach statement. The PI card packet driver includes lots of parameters that set the card's operating conditions. The syntax for the driver is:

```
pi <packet_int> [hw int] io addr] [dma] [baud] [txd]
    [ persist ] [slot ] [tail] [clock mode] [bufsize]
    [ number of buffers ]
```

That's a lot of parameters, but most have reasonable default settings, and the PI card documentation provides more information about each. Note that the PI card's operating parameters are set when the driver initializes, and you can't change them except by exiting NOS, unloading the packet driver, and restarting it.

The Ottawa PI Card[6-3]

The Ottawa PI card is a plug-in board for PCs. It's designed for high-speed performance in conjunction with external modems like the GRAPES 56kb system. It has two ports, one a high-speed port that is DMA driven, which means it doesn't have to generate an interrupt to the computer for every few characters it receives or sends; and a lower speed port that is interrupt driven for 1200 to perhaps 9600 baud rates. The current version, the PI-2 card, has an optional built-in 1200 baud modem for the slow port. The DMA port is capable of handling speeds greater than 56kb, and can do this even in a fairly slow PC. The attach syntax is:

```
attach pi <ioaddr> <vector> <DMA chn> <mode> <name>
    <bufsize> <mtu> <speed a> <speed b>
```

A sample attach command (using the PI's default jumper settings) is:

```
# attach pi card at  port 380, IRQ 5, DMA 1
attach pi 380 5 1 ax25 uhf$vhf 1280 1024 0 1200
```

The PI card provides two interfaces, and this command names the primary interface "uhf" and the slow one "vhf". The "$" character separates the two names. If you're not using the second port you can detach it to save a bit of memory and overhead with the command **`detach <slow interface name>`** after the attach command. The PI attach syntax is explained in more detail in the manual provided with the card.

There's a relationship between the **`bufsize`** parameter in the pi card attach statement and the **`mem ibufsize`** parameter in AUTOEXEC.NOS. The setting of **`mem ibufsize`** must be about 128 bytes larger than **`bufsize`**, or the driver won't properly initialize. I haven't been able to figure out the exact size difference required, but a bufsize of 1280 with an ibufsize of 1440 seems to work fine.

[6-3] *PI Card kits can be purchased from TAPR.*

The ifconfig Command

The **attach** command provides the initial setup for an interface. The **ifconfig** command lets you set the AX.25 and TCP operating parameters separately for each interface. This is particularly useful where you have both 1200 baud and higher speed interfaces; the optimum parameters for one don't work too well with the other, but ifconfig lets you tune each interface for best performance. The basic syntax of the ifconfig command is:

```
ifconfig <interface> <command> <value>
```

The command argument can be nearly any of the ax25, ip, or tcp protocol commands, such as irtt, window, mss, etc., as well as several other options. The value argument is whatever is appropriate for the command given. The ifconfig command provides the flexibility to set operating parameters separately for each interface; this is very helpful when one system is trying to support link layers as different as Ethernet and 1200 baud NetRom.

In general, to set an interface-specific parameter, use this syntax:

```
ifconfig vhf ax25 irtt 3000
ifconfig vhf tcp irtt 4000
```

Another use for ifconfig is to set IP addresses and ax25 callsigns. To simplify routing, hosts normally have a separate IP address for each interface. If you are running more than one interface, you can set that interface's IP address by using the **ifconfig** command after the attach command (remember, if you use a global IP address, it must be assigned before the interfaces are attached):

```
ifconfig <iface> ip addr <aa.bb.cc.dd>
```

You can also use the ifconfig command to assign a unique callsign/SSID to an interface:

```
ifconfig <iface> linkaddr <callsign>
```

Finally, the ifconfig **description** command lets you add a descriptive identifier to the interface; this description will show up when users list available ports with the mailbox "**p**" command:

```
ifconfig <iface> <description>
```

If the description is more than one word long, it must be surrounded by quotation marks: **ifconfig vhf "1200 baud 144.99MHz"**

Chapter

7

Installing NOS: The Rest of the Story

There are a few more files, and a few more tricks, involved in getting NOS up and running. This chapter is a catch-all that will help you add the finishing touches to your NOS installation.

Storing Name/Address Matches in DOMAIN.TXT

Back in Chapter 3 we talked about the way the Domain Name Service matches machine-friendly IP addresses with human-friendly hostnames. You'll want to configure DNS on your system to provide that mapping. How you handle this will depend on whether your local network includes a system that acts as a domain name server.[7-1] If you're that lucky, you only need to create a minimal local domain file (called **DOMAIN.TXT**) in the NOS root directory to contain a couple of special addresses. If you don't have a local domain name server (**DNS**), you'll want to have a more detailed DOMAIN.TXT that includes the hostnames of all the systems with which you communicate

Remember, DNS is a convenience: if you don't have an entry for a host in the file (or the name server doesn't know about it), you can still talk to that host; you'll just need to use the IP address instead of the hostname in NOS commands.

[7-1] Any NOS system can be a DNS server, but there should be local coordination to make sure there's a single source of authoritative information on the network.

However, for NOS to work properly, it needs to have at least two entries in its local DOMAIN.TXT file. One is for the "loopback address" – the internal address every TCP/IP system uses to talk to itself, which is always 127.0.0.1. The other maps your own hostname to your IP address. If those aren't present, strange things may happen; in particular, smtp won't work properly and incoming mail may be lost in never-never land.

If you're using DNS, NOS will automatically save the hostname/address matches it gets from the nameserver in **DOMAIN.TXT**, so you'll find that file growing by itself.

Although the DOMAIN.TXT file created by NOS may look cryptic, the basic format is quite simple. For each IP address/hostname mapping, you need a line that includes the IP address, the keywords "IN" and "A", and the fully qualified hostname, including trailing period. An entry might look like this:

```
127.0.0.1     IN   A    localhost
44.71.36.17   IN   A    n8ur.ampr.org.
```

If you have included the command **domain suffix ampr.org.** in AUTOEXEC.NOS (and you should), you could simply use "n8ur" instead of "n8ur.ampr.org." in the last field. You'll avoid possible problems and confusion, though, by using only FQDNs in DOMAIN.TXT entries. And note that the trailing dot is critical if you use the FQDN — without it, NOS will add the domain suffix (if set), resulting in "n8ur.ampr.org.ampr.org.", which doesn't work too well!

If you are operating in a network with a nameserver, the updates that NOS automatically makes to your DOMAIN.TXT file will have an additional field that defines how long in seconds the entry should be treated as valid, but you don't need to include that field in hand-made entries. Appendix F shows a simple DOMAIN.TXT file.

Security, Permissions, and FTPUSERS

When you run the severs in NOS, you are letting other users into your system. NOS provides a mechanism to control who can do what with your machine. It's based on the FTPUSERS[7-2] file. The idea is that each user on your system can be assigned a login password, the top-level directory they can access on your disk drive, and a permission mask that defines just what commands they can perform.

FTPUSERS started out as a simple file that defined who could upload and download files via ftp (hence its name). As NOS added capabilities, and particularly as BBS and gateway funtionality grew, the number of possible permission settings has become huge. Appendix G shows a sample FTPUSERS file, and lists the premissions that can be enabled or disabled for users in JNOS and TNOS.

Each user defined in FTPUSERS gets a single permission "mask" that's created by adding together the decimal value of all the permissions you want to grant. For example, to give a user access to read, write, and create files in the ftp server, you'd add the values for each permission (1 for read, 2 for write, and 4 for create) to come up with a permission mask of 7. To add the ability to delete files (dangerous!) you'd add 8 more and the mask would be 15. To add additional permissions, just keep adding values. Since the individual values are all powers of 2 (each one is twice as large as the last), the numbers get big pretty quickly!

The entries in FTPUSERS are in this format:

```
<username>  <password>  <directory>  <permissions>
n8ur        zimbot      /pub                7
```

Generally, you'll want your FTPUSERS file to include an entry for you—with a password! — that gives broad permissions, so that you can manage the system. You may have a few trusted users who will also have passwords and perhaps less broad

[7-2] *In TNOS 2.30 and later, the file is called SECURITY.*

permissions. You will want an entry for anonymous ftp users (discussed below in Chapter 10) and, if you're running NOS as a PBBS, an entry establishing permissions for mailbox users and PBBS forwarding partners (that's discussed in more detail in Chapter 13). Finally, if you're running an Internet gateway, there are a whole additional set of permissions you may want to establish.

TNOS also adds several new security functions that operate in conjunction with the FTPUSERS file. They are especially useful in securing systems that serve as Internet gateways because they allow different permissions for users who access the system via the Internet than for those who come in over the radio ports. If you need these features, check the TNOS documentation and release notes for more details.

For added security, TNOS also encrypts the passwords in FTPUSERS each time the program starts. To set or change a password, edit it into the file in plain text. The next time you start TNOS, it will automatically encrypt the password. You can also force encryption while TNOS is running by entering the command **encrypt**.

Finger - an Information Server

finger is a tool that allows users at other hosts to obtain information about your system and your users. The first way to use the finger command is **finger @<hostname>**, which returns information about that host. Additionally, you may be able to get information about a single user by using it this way: **finger <user>@<hostname>**. Finally, the JNOS and TNOS versions each provide system information by substituting a command name for the "user" part of the command. For example, the command **finger users@<hostname>** will return a list of the users currently logged into the mailbox.

The information returned from a NOS system in response to a finger command(other than fingering system information) is read from files in the **finger** directory. Each user who wants to provide information in response to a finger request should have a file with a name that matches their login name in that directory. The data returned by the `finger @<hostname>` version of the command is the list of files contained in the finger directory; in TNOS and JNOS, the output also includes the built-in commands that may be fingered.

Fingering a single user – **finger fred@n8ur.ampr.org** – displays the contents of that user's finger file. You can use any ASCII text editor to create these files. Don't go overboard — one screen of text is plenty.

You can also create additional files with information about specific aspects of your system. For example, you might have a list of the files available for downloading on your system in a finger file called "filelist." A remote host who issues the command `finger filelist@<myhost>` will get that list.

The AT Command

If you keep your NOS system running all the time, you'll find the **at** command a lifesaver. It's a timed execution command that allows you to schedule events to occur at a given time. Using "at" commands in AUTOEXEC.NOS allows you to set up your system to take care of its own housekeeping. The syntax is simple:

```
at <time> <command>
```

The time can be specified in yymmddhhmm, hhmm, or mm form. If you want the event to occur at a time relative to the time the at command is run, you can specify the time as "now+hhmm". Let's say we want to exit NOS at 10:15 AM, and the time is currently 10:00 AM. Either **at 1015 exit** or **at now+0015 exit** would work.

If the command has any subcommands or options (*i.e.*, it consists of more than one word), you need to put the entire command string in quotes. For example, use **at now+0015 "attend off"** to turn off the attended flag fifteen minutes from now.

You can extend the power of the **at** command by using it with the **source** command to execute a series of commands stored in a separate file, rather than a single command. For example, the command **at 0015 "source cleanup.scr"** would execute all the commands contained in the text file "cleanup.scr".

The **at** command is particularly useful if you are running NOS as a PBBS, where there's regular maintenance to be done. But it can be handy even when you're operating an end-user system. TNOS also provides a more sophisticated timed-execution command called "cron"; see the TNOS documentation for information on using it.

Memory Matters

I've already mentioned (several times!) that NOS stretches the limits of an MS-DOS computer's memory. You can see how memory is being used in your system in two ways. First, there's a **mem stat** command that you can issue from the command line to display memory statistics. Second, if you are using a newer version of NOS and have enabled the status display with the **`statline on`** command, you will see a running display of the critical memory numbers on the top of your screen.

The display shows a bunch of statistics, but there is one that's critical, and another that's useful. The critical value is the **coreleft**. This is the amount of unused memory that's available for NOS to grab from DOS if it needs to. If this value falls below 16k or so, your system is likely to become unstable when new sessions are opened or new users sign on.

The useful value is **heap avail**. It tells you how much memory NOS has grabbed from DOS, but isn't using at the moment. Adding the values of coreleft and heap avail together yields the total amount of unused memory NOS has to work with.

You may have memory problems even though these values show a reasonable amount of free memory. Because of the way the DOS memory management system works, available memory can be fragmented into chunks that are too small to meet a request from NOS, and for various reasons small pieces of memory can't be put together to create larger ones. There's good news, however—recent versions of JNOS and TNOS have added memory management routines that "reclaim" core memory, and this greatly improves NOS stability, though memory remains the biggest challenge to running a DOS-based NOS system.

NOS typically starts with a relatively high **coreleft** value, which rapidly reduces and in theory should become fairly stable after the system has been running for a while. Rapid decreases in **coreleft** after the initial reduction at startup are signs that something is sucking up memory. Lots of unclosed sessions can cause this problem, as can a dead TNC with lots of data queued up for transmission. If you're using a PI card driver, or a packet driver, the setting of the **nibufs** and **ibufsize** commands have an impact on memory usage.

NOS is a good reason to have a memory manager like **EMM386** (included with recent versions of MS-DOS) or **QEMM** running on your system if you have a 386 or higher processor. NOS can make good use of the extra memory made available by tricks such as loading DOS in the high memory area, and loading TSRs (like packet drivers) in high memory. In addition, JNOS and TNOS have command line switches that can be used to load some of the NOS data structures into high memory. That won't buy you much, but every little bit helps.

If you have a monochrome video card, or a VGA card running with a monochrome display, you can steal an extra 64k for your DOS window by telling the memory manager to include in low memory the addresses from A000 through AFFF that are normally reserved for color displays.

Despite all these tricks, the bottom line still is that you should use a version of NOS compiled with only the feature set you really need. If you don't need NET/ROM support, but have it compiled into your program anyway, you've just burned up at least 40k of memory. Building the convers server into your system will cost 50k. The most stable NOS is the one that includes only the features you need.

Recent versions of KA9Q NOS and TNOS have completely shattered the DOS memory limitation. A freeware 32-bit C complier called "DJGPP" can build DOS executables that are capable of using all the memory an 80386 or better computer contains. Phil Karn rewrote NOS to work with that complier, and TNOS/DOS migrated to it as of verstion 2.20. As a result, a single executable TNOS/DOS program can include just about every feature imaginable, and still run reliably on a 386 machine with at least 4MB of RAM. There is one downside – the DJGPP version of TNOS doesn't seem to work with Crynwyr packet driver described in Chapter 6. That problem will hopefully be solved before too long.

NOS Versions: Build or Buy?

One of the great things about NOS is that all the popular versions are available in source code form, as well as in pre-compiled executable files. This makes it easy for people to mess around with the program to create their own vision of what NOS should be. Of course, this has the potential for disaster, but we've been lucky—NOS has grown and prospered as a result of having lots of folks working on the code.

Consider the pedigree of TNOS. KA9Q wrote the original program, and his versions remain the baseline (Phil is still generating a new release of his code once a year or so). In 1991, PA0GRI added his own work to create the GRI versions, to which N1BEE later contributed. Then, WG7J used the GRI code as the basis for JNOS, which was the starting point for KO4KS' DOS version of TNOS. Meanwhile, KF8NH was working on a Linux port of JNOS (which itself was based on other folk's earlier work on UNIX versions), and much of that code went into TNOS/Linux. Starting from this base, Brian added a tremendous amount of new work to create TNOS as it exists today.

Along the way, WG7J moved on to other things, so N5KNX took over development of JNOS and has sheparded that version of the program through several updates. KO4KS and N5KNX keep in touch with each other, so many of the improvements (and especially the bug fixes) added to one program find their way into the other.

We also need to note that lots of important code came from yet other folks. SM0RGV wrote code for the mailbox (BBS) subsystem, W9NK developed Net/ROM support, and additional features, design ideas, and bug fixes came from dozens of others. The full list of contributors to the various versions of NOS numbers several dozen; it's truly been a group project.

You can see that the versions of NOS that we're using today are built on the work of lots of people. You can probably guess that "lots of people" equals lots of different features and options, and you're right. People have added everything but the kitchen sink (and there are rumors about that) to the code base. NOS is configurable… extremely configurable.

The options in NOS affect both the software features and the hardware support in the program. Although it would be nice to just turn everything on and go to town, under the DOS memory limits there's simply no way that you can run, or even compile, a version of NOS that has all forty or fifty options built in. Even if memory isn't a problem, added features tend to reduce stability, so it's wise to only compile in the features you need. NOS uses a configuration file in the source code directory to define what features are included in the compiled program. You need to select a set of options that both meets your needs and fits in the available memory, and you may need to adjust your expectations and reduce the option list until you end up with a program that will run.

What is the impact of including lots of features in your custom configuration? Mike Murphree, N4CNW, has done some experiments with TNOS to see how much the various options add to NOS' memory requirements. Including NET/ROM support adds over 40k of code. The convers server adds another 50k. The POP3 client and server together add 15k. TIPMAIL and XMODEM support add 9.6k. It's easy to see how these bits of memory add up to create a monster.

You can compile your own version of NOS to get the features you need. You don't need to be an expert in the C language to do it. Editing the CONFIG.H configuration file is pretty straightforward, and all you need is a text editor[7-3]. Just remember that the more stuff you include in the executable program, the more difficult it will be to run in a stable way.

Compiling NOS under DOS

If you want to compile your own version, you'll need a compiler program for the C language. Borland C++ version 3.1 is currently the standard compiler for most DOS versions of NOS, though early TNOS versions were based on Borland version 3.0 and current versions of TNOS and KA9Q NOS use the freeware DJGPP complier. You want to get "Borland C", not "Turbo C". It may be possible to build NOS with other compiler brands and versions, but there's no guarantee that things will go smoothly, or that the resulting executable program will work. The steps to compile the program are pretty simple (though there may be additional steps required for some versions or some hardware environments; always check the "readme" and documentation files for the NOS version you're using to make sure you are doing all the right things):

- Install the compiler software, and then install the NOS source code files in their own directory.
- Make sure you have the directory where the compiler program lives (usually a "bin" directory under the main compiler directory) included in your PATH statement.
- Change to the NOS source code directory.
- For TNOS, run the "pre make" program included with the source code.

[7-3] *Current versions of TNOS include a program called "MKCONFIG.TCL" that builds CONFIG.H for you based on your response to a series of questions. That simplifies things, and helps ensure that incompatible options don't fight with each other.*

- Edit the file MAKEFILE to reflect your computer's environment (the comments will tell you what to do).

- Make a backup copy of the CONFIG.H and then use a text editor to edit it (if you're building TNOS, run the program MKCONFIG.TCL which will do all the work for you). All you'll be doing to CONFIG.H is defining or undefining options to turn them on or off. Each option is identified by a name, and is controlled by a line containing a definition keyword followed by the option name. For example, to include the BBS server, you would have a line that reads "**#define BBS**" (including the pound sign, but without the quotes). To leave the BBS out, use "**#undefine BBS**". All you should do is change existing lines in CONFIG.H from "#undefine" to "#define" or vice versa. The abbr eviations "#def" and "#undef" will also work. You shouldn't add or delete lines. The comments in CONFIG.H will tell you what does what, and will also show you what not to change. Save the edited file.

- Type "make" from the DOS prompt. The compile could take an hour or more (depending on the speed of your computer), so be patient. You'll see lots of messages whizzing across the screen, and you may see "warnings." It's safe to ignore these. If you see "error" messages, you probably have a problem. If you get errors at the very beginning of the process that complain about not being able to find files, the chances are that MAKEFILE doesn't have the proper directory information, or you don't have your path set to include the compiler programs.

- If all goes well, the DOS prompt will eventually return without an error message, and you will have a NOS.EXE or TNOS.EXE executable sitting in the source code directory. Copy it over to your NOS working directory, run it, and see what happens!

When you run your customized version of NOS, check the memory usage. In particular, take a look at the **coreleft** and **mem avail** values that are discussed earlier in this chapter. immediately upon startup, and after the system has been running for a while, to see how much working memory you have. If it's low, or if NOS won't even start without a memory error message, you'll need to undefine some features and try again. Remember: build the minimum set of features to do the job you need, and you'll have a much easier time getting – and keeping – things running.

NOS under Linux

To get around the DOS "RAM cram," and gain other advantages, more and more hams are running KO4KS' TNOS under the Linux operating system. Linux doesn't have the MS-DOS memory constraints, and a TNOS version with all the features compiled in will run quite reliably. This means that you don't need to compile TNOS/Linux yourself (although you can if you want – full source code is available). The executable file that you download has everything included, and even with all that code is stable.

The TNOS/Linux combination requires a 386 or better processor, and at least 4MB of RAM (although 8MB is a more resonable memory size); the hardware demands are greater than under DOS, but so is the functionality and reliability. The other tradeoff is that Linux, an offshoot of the UNIX operating system, has a learning curve of its own. There are lots of good books on Linux basics available, though, and it never hurts to add another operating system to your resume!

Linux itself is freeware, and is available on the Internet, though the most practical way to get a Linux system running is to buy one of the several inexpensive (some as little as $20) CD-ROM distributions that are available. Appendix A lists some of those distributions. There's also a Linux version of JNOS 1.11a available if you prefer that flavor of NOS. Like TNOS/UNIX, it requires a 386 or better processor and at least 4MB of memory. We'll talk more about using Linux for amateur TCP/IP in Part 4.

Chapter

8

Some Boring but Necessary Technical Stuff

Before we move on to the good stuff about how to make NOS do magic, we need to talk about some of the low-level parameters that, you may need to tweak depending on local custom and the quality of the RF paths you are using. These parameters define how NOS ships data out over the air, and they are similar to some of the TNC commands you may be familiar with.

AX.25 uses the **`paclen`** command to limit the size of packets, **`maxframe`** to set the maximum number of packets "in flight" at any time, and **`frack`** to set the time between retries. NOS has parameters to set these same values at the AX.25 level. In addition, parameters at the TCP layer provide additional control over the way data moves between hosts.

Packet Sizes and Windows

In theory, the larger the **datagram** (TCP/IP's term for a single block of data), the higher the efficiency, because the header information that is a necessary part of each protocol layer adds a fixed amount of overhead. In larger datagrams the overhead is a smaller percentage of the total data sent (see figure 8-1).

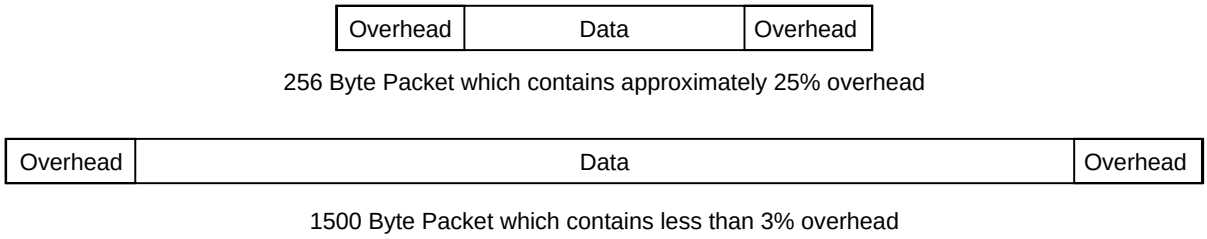


Figure 8-1 - example of overhead in datagrams

However, some networks (such as NET/ROM) can't handle large datagrams in one piece. More importantly, the larger the datagram, the longer it takes to transmit, and on a busy or marginal path, the greater the likelihood that something will corrupt it along the way. It only takes one bad bit to ruin an entire packet, whether the packet contains one byte of data, or one thousand. Since it takes longer to resend a large packet than a small one, the cost of retries is greater. These factors mean that a fast network with clear channels and solid paths can get away with using larger datagrams than a slow, unreliable one.

NOS provides three parameters that deal with datagram sizes. The most important one is the **MTU** (the sixth value in the **`attach asy`** command described above). It is similar to **paclen**; it sets the size of the largest packet, including any higher layer protocol headers, that can be sent on an interface. Datagrams larger than the MTU are **fragmented** into multiple pieces, which seriously reduces efficiency.

Fragmentation is a Bad Thing for several reasons. First, the high data-to-overhead ratio of large packets may be lost because the fragment containing the leftover data may be quite small. For example, Figure 8-2 shows how a packet containing 250 bytes of data may be fragmented when it passes through an interface that supports a smaller MTU.

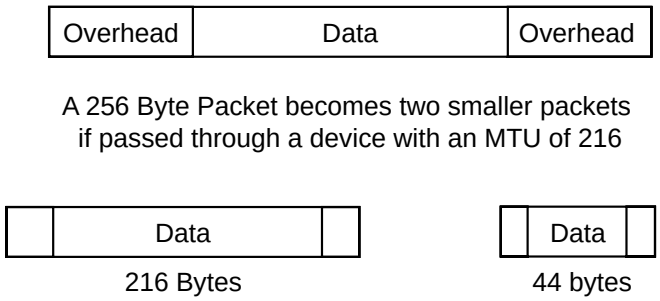


Figure 8-2 - Example of packet fragmentation

The second fragment contains only 44 bytes of data; there are more bytes of overhead in the AX.25, IP, and TCP layers than there is data! The fragments aren't reassembled until they reach their final destination, so even though subsequent links could handle the unfragmented datagram, they will instead will have to transmit the smaller, less efficient fragments.

One common cause of fragmentation in amateur networks is a route that travels over a NetROM circuit. NetROM has a maximum packet length of 256 bytes, so any larger IP frames that pass over it will be fragmented. Since NetROM adds protocol overhead of its own, fragmentation over NetROM circuits is especially painful.

The second problem is that if any fragments are lost in transit, the whole process will have to repeat – there's no way to request retransmission of only one fragment. The receiving host starts a **reassembly timer** when it receives an initial fragment, and if that timer expires before all the fragments have arrived, all the data will have to be resent and refragmented.

Each interface has its own MTU (maximum transmission unit) set as part of its attach command. For 1200 baud channels that are shared with other packet users, a value of **256** is reasonable; in fact, since that is the largest packet size most non-TCP/IP ham networks (like digipeaters and NET/ROM) are designed to handle, 256 is the largest MTU you should specify if any of your packets are going to travel via such networks.

Faster networks that don't have links using other protocols may take advantage of higher MTU values. For good-quality channels with fast data rates (9600 baud or above), it may be reasonable to use an MTU ranging from 512 to 1500 (which matches the standard value used by Ethernet systems).

The other two parameters that set datagram size are part of the TCP protocol. **tcp mss** (maximum segment size) is the largest chunk of data that TCP will send at a time. Because the TCP and IP headers attached to each datagram total 40 bytes, mss must be 40 bytes smaller than MTU; **216** is the correct value for an MTU of 256.

The **tcp window** parameter is like **maxframe** – it tells NOS how many segments can be outstanding at once. If the tcp window is set to twice the value of mss, NOS can receive two segments before sending an acknowledgment. A large window can improve efficiency because, among other things, multiple acknowledgments can be sent in a single packet.

Although using a large window has major benefits on full duplex networks, on typical ham networks the best performance comes from smaller windows that are one to three times the mss. In fact, Phil Karn has shown that under real-world conditions on our networks, a window that's exactly equal to the mss provides the best total throughput. You may or may not find that to be the case in your neighborhood. You won't be too far off if you start with a window equal to twice the value of mss (**432** for an mss of 216). Some experimentation from that point should show what works best in your environment.

In summary, good starting values are:

	1200 baud shared channel	9600 baud or higher clear channel
`mtu`	**256**	**1500**
`tcp mss`	**216**	**1460**
`tcp window`	**432**	**2920**

Timers

Another set of parameters that can have a major impact on performance are the settings of the AX25, NET/ROM (if you're using that capability in your system), and TCP timers and retry controls.

By default, NOS uses a retry mechanism that is very polite—it backs off (i.e., increases the time between retries) geometrically as the number of retries increase, until some quite-large maximum delay is reached. In other words, if the time until the first retry is one second; the second retry will occur two seconds later; the third four seconds later; and the fourth eight seconds later. This is the desired behavior on a network like Ethernet, where retries are almost always due to collisions between packets. But on ham networks, where retries might mean that the path is marginal, rather than that the channel is very busy, these friendly parameters can quickly take retry times toward infinity and dramatically cut down the amount of data you can move.

The command names and syntax vary between NOS versions, but the first parameters you might want to tweak are the **timertype** settings for the AX.25, NET/ROM, and TCP protocols, which often can be set to:

- The default exponential backoff that increases the delay by a factor of two, then four, then eight, etc.;
- A linear backoff that doubles, triples, quadruples, etc., the original delay time and therefore is a bit more aggressive than the default; or
- The same algorithm used in the original TNC code, which doubles the retry delay after the first retry, and keeps it constant thereafter—this is the most aggressive timer type.

You'll have to experiment to decide which timer type for each protocol is most efficient in your environment. Remember that more aggressive timers make NOS a less friendly inhabitant of the channel, so make sure your settings don't make you a bad neighbor!

You may also want to adjust the **IRTT** (**I**nitial **R**ound **T**rip **T**ime) values for the AX.25, NET/ROM, and TCP protocols. IRTT is similar to the FRACK value in a TNC; it's the system's first guess at how long to wait between retries. NOS is much smarter than a TNC, though, and it will adjust the round trip timer based on the actual time it takes for a round trip. Setting the IRTT to an appropriate value will ensure that NOS is close to the mark on the first few retries, will help the system learn the true RTT more quickly, and will result in an optimized retry strategy.

Some NOS versions have a **tcp maxwait** command that may be useful. It sets the longest time (in milliseconds) that the TCP protocol will wait for an acknowledgment before sending a retry. Used in conjunction with either the exponential or linear timertypes, this allows you to be a friendly neighbor without your retry times reaching infinity should the channel temporarily fall apart.

Datagrams vs. Virtual Circuits

There's one more performance tweak you might try. TCP/IP is designed to be a **datagram**-based protocol. That means that it doesn't establish a link-layer connection with the other station before sending data – it just shoots datagrams off and expects the TCP layer to handle acknowledgments and retries.

That's one of the reasons that AX.25 isn't an ideal link layer protocol to use under IP; IP frames are sent as UI (unnumbered information) packets that use only a part of the AX.25 feature set. In particular, IP doesn't use AX.25's mechanisms for sequencing or retries, but rather relies on the higher-level TCP protocol for that. Additionally, datagram networks don't do acknowledgments and retries hop-by-hop, but rather on an end-to-end basis. That means that the system can't react immediately to an error, but has to wait for the round-trip timer to expire before retrying. This can cause unnecessary delays and reduced throughput on routes that include several hops.

There's a near-religious war in the networking world between datagram proponents, and their opposites, the **virtual circuit** crowd. Networks that use virtual circuits establish a link-layer connection (as if they had issued an AX.25 "connect" command) between stations before data transfer begins, and they rely on the link level protocol (in our case, AX.25) to handle timers, error checking, and retries. Virtual circuit proponents argue that when links are unreliable, their system provides better throughput than datagram networks, largely because acknowledgments and retries happen each step along the way, rather than end-to-end.

Leaving the wars behind, there may be something to be said for virtual circuits in our world of marginal RF paths. Not to worry; NOS is nothing if not flexible. It provides a virtual circuit mode that you can specify for each interface by using an appropriate option with the **mode** command:

```
mode <interface> vc|datagram
```

If you're trying to make data flow over an unreliable path, experimenting with virtual circuit mode may be worthwhile (remember that it must be turned on at both ends of the path).

Before Leaving the Boring Stuff...

Even more than in other parts of this book, this discussion glosses over lots of subtleties. Much of what I've said here is a bit controversial; there are a wide range of views on how to tune these parameters for best performance. Changing them can drastically affect throughput for better or worse, and both experimentation and local consensus are necessary to come up with settings that work well without stomping on other users of the channel. One user with wrongly set parameters can bring a network to its knees, so please be considerate and ensure that your system is compatible with your neighbors.

Chapter

9

Using NOS

We're finally ready to begin using NOS, instead of just talking about it. This chapter will help you figure out what to do once you finally see the NOS prompt on your computer screen.

KISS Mode

Once NOS is installed and your configuration files are set, you need to do one more thing before you're ready to start: get your TNC talking to your computer in **KISS** mode. Nearly all TNCs support KISS in one way or another.

What is KISS? The acronym stands for the common expression, "Keep it Simple, Stupid." In this context, it's software that "dumbs down" your TNC and lets NOS manage the AX.25 protocol itself. A TNC operating in KISS mode deals only with the lowest-level activities of assembling and disassembling packets and managing things like TX delay; it lets NOS control retry timers, packet lengths, and all the other parameters that we've been talking about.

To put your TNC in KISS mode for the first time, you'll typically need to use a standard terminal program to issue commands to the TNC to set the serial line baud rate to the same speed as you've specified in the NOS attach command, to 8 bit data, and to no parity. Then, issue the KISS command (for a TNC2, "kiss on"; for a Kantronics, TNC, "interface kiss"), and the TNC's software restart command. After that, you won't be able to talk to your TNC via the terminal program, but NOS will (don't worry, you can easily return the TNC to normal mode if you want to). Now, you're set to run NOS.

One trick that NOS supports is the ability to send commands to the TNC during startup through entries in AUTOEXEC.NOS. The **comm** command will send a string of text to the named interface. For example, to force a Kantronics DataEngine or KAM into KISS mode, you could issue these commands:

```
comm vhf "interface kiss"
comm vhf "reset"
```

For a TNC-2 clone, the commands would be:

```
comm vhf "kiss on"
comm vhf "restart"
```

Note that surrounding the text with quote characters will preserve spaces embedded in the command. Once you know what commands are needed to put your TNC into KISS mode, you might want to put the appropriate **comm** statements into your AUTOEXEC.NOS file (*after* the attach commands, but before any parameters are set) to automatically set KISS mode every time you start NOS.

If you use your TNC with programs other than NOS, you'll want to return it to KISS mode before running those programs. Issuing a **param <interface> 255 0** command from the NOS prompt will return the TNC to normal command mode (and cause that interface to stop working in NOS). Some NOS versions have yet another trick available

to make your life easier: after you issue the exit command to quit NOS, they will read the contents of a text file, located in your NOS root directory, called ONEXIT.NOS, and execute the commands found there before shutting down. Putting the command to KISS OFF in that file will leave you with one fewer thing to remember.

If you're stuck and can't get your TNC out of KISS mode, there's another trick you can try: fire up your terminal program, and, while pressing down the shift and the ALT keys, use the keypad to type a 0, then a 255, then another 0. That should return the TNC to command mode.

If that doesn't work, as a last resort you can remove power from the TNC, open it up, and remove the jumper that connects the memory backup battery. After a few seconds, the TNC should lose its memory. When you reapply power, you'll be back in command mode, but the TNC will have all its parameters set to their factory defaults.

Some TNCs are known to drop out of KISS mode while in operation, which can be annoying. If you're using your TNC for nothing but NOS, you may want to replace the standard firmware with a new EPROM that is strictly for KISS. KISS EPROMs are available for the TNC-2 as well as several other TNC models.

The NOS Display

With that taken care of, we're finally ready to get NOS started. From the DOS prompt, type **NOS** (or whatever your executable program is named). In a few seconds, you should see a clear screen, with some startup information at the top and a prompt.[9-1] Error messages that appear first usually indicate a problem with one or more commands in your AUTOEXEC.NOS file.

Sessions

In NOS-speak, **sessions** are the various screens that are active in the program. NOS supports multiple simultaneous sessions, so you can have a lot going on at once. There's always a **command session** that shows you the NOS prompt and allows you to issue commands to the program. Each outbound connection you initiate, whether via AX.25, telnet, or ftp, is a session.

If you want to monitor channel activity, you can establish a **trace session**. When you issue commands to view a file on your local disk, or list a directory's contents, those are also sessions. You can also shell out to a DOS command line; that's a **DOS session**.

Whenever you see the NOS prompt, you are in the command session and what you type is sent to NOS, not to another station. All commands that you enter at the NOS prompt need to be followed by the return key.

When you've initiated a connection to another station, and see data on your screen, you are in a **converse session**; what you type is sent to the remote host. That's an important concept: all sessions except the command session pass your keyboard input through to the station or program at the other end of the session. You can't issue commands to NOS while you're in a converse session; you need to return to the command session to do that.

[9-1] *Recent versions of NOS allow you to modify the prompt similar to the way you can change the DOS prompt. Old versions used a simple "net>" prompt.*

Managing Sessions

You can list the current sessions from the command session by typing **session** (which may be abbreviated to **sess**). To get from command session to another session, you can either type **sess <session number>** or press the function key that corresponds to that session number—the F3 key will take you to session number 3, for example (if you have turned on tracing, as described below, the trace session is reached by pressing F9). You can also use the function keys to jump from one session directly to another, bypassing a stop at the command screen. As a shortcut, pressing the return key by itself from the command screen will move you to the most recently active session screen. To return to the command session, press F10.

The preferred way to close a session with a remote host is to use the exit command provided by the host. In other words, if you're logged into a remote host's ftp server, issue the **quit** command and let the remote station terminate the session gracefully. If that's not possible, you can issue the command **close <session number>** from the command session. If there's only one session open, you can just enter **close**. The close command does an orderly disconnection; it won't work if the other end has stopped responding. In that case, you can use the **reset <session number>** command to force a session to end. However, a forced disconnection can leave things in a disorderly state, and you should use the reset command only when you have to.

A Graceful Exit

You should never simply exit from the NOS program when you have an open session. Doing so can cause great unpleasantness at the remote host. Unless you're in some sort of software or hardware lockup, or you *know* that the station on the other end has gone away, always close sessions and wait for confirmation before exiting the program.

You should also be aware that your system may have started activity in the background, for example to transfer electronic mail, or someone else may have initiated a connection with your system. You may not even know these things are happening. Pulling the plug on them would be very impolite. Before exiting NOS, you should first use the **session** command to make sure there are no current sessions running, and then the **socket** command to see if there are any background connections established.

socket will show you a list of sockets (which are connections either in use or available on your system). If you see any entries in the "Remote Socket" column of the display, that means there's at least one remote host communicating with you, and you shouldn't exit until that socket has been closed. By the way, if a socket appears "stuck" and no data is flowing, you can try to bring it back to life with the **skick** command. Issuing **skick <socket number>** will force NOS to retransmit the data currently in its queue, regardless of the current timer state.

If you need to manually kill a socket, you can do so by entering the using the reset subcommand of the ax25, ip, or tcp commands with the "PCB" entry listed in the **socket** display for that socket. For example, an AX.25 connection to KE8TQ might show up in the **socket** listing like this:

```
S#  Type    PCB       Remote Socket  Owner
153 AX25 I  08190c00  KE8TQ on uhf   08116a00 connect
```

If you enter **ax25 reset 08190c00**, that will reset the socket and terminate the connection. If this socket was associated with a session (*i.e.*, it was a session that you initiated with a connect, ftp, or telnet command), the session will still be alive, though in "limbo." To clear it, go to that session screen with the appropriate function key or the **session <session number>** command and hit return. That will close out the session from your end. Remember, using the reset command doesn't generate a graceful exit; do this only as a last resort!

Help

Typing **?** in the command session will display a list of commands. Typing a command name followed by ? will display available subcommands. JNOS and TNOS both offer more extensive help with – surprise – the **help** command. Assuming that the help files are installed in the right place in your NOS directory (if you install the "help.zip" or similar packages they should be), typing **help <command>** will display a short description of that command and its options. You can create your own help files; they are text files named for the command, and are located in **spool/help** in JNOS, or **etc/help** in current versions of TNOS.

File Commands

You can issue several commands from within NOS to deal with local files and directories. **pwd** displays your current working directory, and **cd** allows you to change directories. **dir** displays files in the current directory. **mkdir <dirname>** creates a new directory, and **rmdir <dirname>** removes one. **delete <filename>** erases a file. **more <filename>** will display the contents of a text file using the NOS paging routine to display one screenful of text at a time. As at the DOS prompt, the "*" and "?" wildcards can be used in filenames.

You can also “shell out” to the operating system from within NOS by entering either an exclamation mark (**!**) or the command **shell**. To return to NOS, type **exit** at the OS prompt. The amount of memory available to run a DOS program from this shell may be very limited, so don't plan on launching a large program in the shell. Also, depending on your NOS version and its configuration, shelling out may bring all other NOS activity to a halt – including sending and receiving packets – so it's not a good idea to do this when there are active sessions at your system.

You can capture incoming data from the current session to a disk file by using the **record <filename>** command, and you can read in a data file from disk to the current session with the **upload <filename>** command.

Chapter

10

NOS Functions

Using NOS for AX.25 Contacts

Although NOS is designed primarily to use the TCP/IP protocols, it also lets you make standard AX.25 connections to other stations. AX.25 sessions work like other sessions, and are created with the command:

```
connect <interface> <callsign> [digi1 digi2 etc.]
```

from the command session. By the way, the first thing you should do after installing NOS is try out some AX.25 connections to make sure that your hardware configuration and basic parameters are working. Knowing that AX.25 is working properly will make troubleshooting TCP/IP problems much easier.

Establishing an AX.25 connection through NOS is like using the standard TNC commands, but with a few small differences. First, since NOS can support several interfaces, each with its own hardware, you need to tell NOS which one to use.

So, to connect to W8APR using the radio on interface "vhf", enter **connect vhf W8APR**. Once you get a "Connected" message, you'll be able to type to the station at the other end just as you would with normal packet. If you're connected to a BBS or other service, you should exit using the command at the remote end. If that isn't possible, you can exit an AX.25 session either by entering **close <session number>,** or **disconnect**, from the command session prompt. Just as with a TNC, these commands can be abbreviated; just how few of the letters are necessary will depend on each implementation of NOS and the commands it supports.

The other minor difference between the NOS connect command and a regular TNC is that the word **via** is not used when specifying digipeaters, and you can't use a comma to separate multiple digis—you use a space instead. To connect to W8APR through N8KZA on interface ax0, you would enter **connect vhf W8APR N8KZA.** Check your reference manual, though: the **via** keyword and commas in the digi list are likely to be supported in new versions of JNOS and TNOS.

Recent versions of JNOS and TNOS also offer a way to divide the AX.25 session display into two windows, one for received data and the other for data you enter. To enable this feature, use the command **split** instead of **connect** to start an AX.25 session: for example, **split uhf ke8tq**. This option must be compiled into the NOS executable file to work, so some precompiled versions may not support it.

Telnet

The **telnet** command logs you in to a remote TCP/IP host; depending on the capabilities of that host, you might find yourself chatting directly with the user at the other end, connecting to the NOS mailbox, which acts very much like a sophisticated personal PBBS, or getting a UNIX "login:" prompt. To establish a telnet session, enter **telnet <hostname>** at the command prompt.

JNOS and TNOS offer a new type of session that improves on telnet for real-time keyboard-to-keyboard chats. It's called **ttylink**, and it works just like telnet except that it connects you directly to the remote host's chat mode, and uses a split-screen format (like the ax25 **split** command)to make things less confusing as you type to each other. To start a ttylink session, use the command **ttylink <hostname>**.

Your ttylink session will result in a message like "Telnet session 1 failed: Reset/Refused errno 9" if the remote host doesn't support ttylink. If the operator at the other end isn't available to chat, you'll get a message like "The system is unattended." You'll still be able to type, but there won't be anyone there to reply. You can set whether or not your own system will give this message to other stations by setting the **attended** command to either **on** or **off**. You might want to put this command in your AUTOEXEC.NOS file to set your default status. You exit from ttylink by issuing the **close <session number>** command from the command session prompt.

Using Telnet to Connect to a UNIX Host

When you telnet from one NOS system to another, everything works very much like a traditional AX.25 connection. The screen display doesn't use any special formatting characters; it's treated as if it were a glass teletype without any way to move the cursor other than down or to the right. Data is sent line-by-line when you press the carriage return, and the echo of what you types is locally generated.

When you telnet to a UNIX host, things are different. Most UNIX systems think that they're talking to some sort of terminal (often a VT-100) that knows how to clear the screen and do cursor addressing. And, in UNIX telnet sessions data is sent character by character and is echoed from the other end. If you stop to think about it, that's really the only way you can use a program like a text editor over the network—the remote system has to have control what appears on your screen, and echoing each character allow use of cursor control keys, backspacing, underlining, and other features.

These two factors cause a lot of trouble for NOS users who telnet to a Linux or other UNIX system. The first problem is pretty easy to deal with. Assuming you're running under MS-DOS, just make sure that ANSI.SYS or one of its replacements is loaded as part of your CONFIG.SYS file. That will allow your system to respond properly to the VT-100 control codes (which are very similar to the codes supported by ANSI.SYS), and make most UNIX telnet sessions work properly, though you may need to play with terminal definitions at the UNIX end to get everything working just right.

The second problem—character-by-character echo—is a little bit tougher. When you telnet to another host, the telnet protocol exchanges information about the capabilities of the system at each end. The UNIX end will want to take over control of character echo, and will try to put NOS into character mode. That means that every character you type on the keyboard will get wrapped up in its own special TCP frame, complete with 60-something bytes of overhead. As you might have figured out, this doesn't work too well over a 1200 baud radio channel. Heck, it doesn't work too well over a 19.2kB radio channel (though at that speed it *is* usable). The same problem occurs in the other direction when a UNIX host telnets to a NOS system.

There is a way to at least partially solve this problem. By entering the command **echo refuse** prior to opening a telnet session (or as part of AUTOEXEC.NOS) you can tell NOS to refuse the remote echo, character-by-character option. That will make the remote host work the same way as a NOS system does. In this mode, you will be able to do command-line based things on the remote host, but trying to run programs that take over the screen (like most text editors and many mail programs) will result in disaster.

Telnet sessions initiated by a UNIX system to a NOS system are another matter. Getting the Unix telnet command to work in line-by-line, local echo, mode requires commands at the UNIX end that may be difficult or just plain unavailable. With many versions of the UNIX telnet program it's either difficult or impossible to come up with a configuration that makes the keyboard work right. At the moment, all we can say is that NOS running over a slow radio link does not make a very good remote terminal for a UNIX system, and it's a problem that can't be solved from the NOS end alone—we need a tweaked UNIX telnet program to deal with this.

Telnet is a very versatile tool. In Chapter 2, I mentioned that each TCP service (like telnet, ftp, and smtp) listens for connections on a specified IP port. For example, the telnet server listens on port 23, and smtp listens for connections on port 25. Most telnet clients can establish a session using any port number, not just 23 – that's just the default port that's used when one isn't specified. But if you'd like to chat with an smtp server, all you need to do is add the port number after the hostname in the telnet command:

```
telnet n8ur.ampr.org 25
```

and you will find yourself chatting with smtp. This is useful for debugging problems, and also allows the telnet client to be used for unplanned protocols like the **convers** conference bridge server (more about that in Chapter 19).

File Transfers with FTP

One of the most common uses of TCP/IP is file transfer. The file transfer protocol, **ftp**, is widely available and many ham hosts maintain directories of files that are available for download.

You initiate an ftp session by entering **ftp <hostname>** at the command prompt. An ftp session is a bit of a mixed bag. Although it looks like a standard converse session, some ftp session commands act locally while others act at the remote system.[10-1] For example, some commands rename or delete local files, or change the local directory. Others allow you to do the same thing on the remote system (if you have permission, of course).

Once the session is established, the ftp session will prompt you for a user name and a password. If your hostname and password have been added to the remote host's FTPUSERS file, you'll have the ability to download and perhaps upload files in the directories permitted you. If you haven't arranged with the remote host for your own account, you can try to login as **anonymous** or **guest**; many systems support these user names and grant limited (usually download-only) privileges to them. If you login under one of these accounts, you should enter your email address as the password to let the remote host to keep track of who's been using their system.

Once you've logged in, you'll see a new prompt: **ftp>**. This will remind you that you're actually issuing commands to the ftp session. From the ftp> prompt, you can list the files in a directory on the remote system, change directories, upload files, or download files.

To list files, enter **dir** or **ls** at the ftp> prompt. You will get a listing that shows subdirectories (if any) and files together with their dates and sizes. To show the current directory name, type **pwd**. To change directories, issue the **cd <directory>** command.

[10-1] ftp sessions actually use two separate data streams. One carries commands between the ftp client and the remote ftp server, while each time a file transfer begins a separate stream is started to move the file data.

Note that directories are displayed with a forward slash (/) instead of the usual MS-DOS backslash (\). That's because the UNIX operating system, which is TCP/IP's natural home, uses forward slashes. If the remote host is running NOS, you can use either character, but most other systems will recognize only the forward slash.

You can change the local directory where retrieved files will be put (and where files to be sent will be found) with the **lcd** command. **lmkdir** will make a local directory, and **lrmdir** will remove one. Local files may be listed with the **ldir** command. **lrename** will rename a local file. To monitor the progress of a file transfer, issue the command **hash on** *before* the transfer begins. Then, a "#" symbol will appear for every kilobyte of data transferred.

Once you've found a file you want to upload or download, you need to make a decision. ftp can transfer the file either as an **image** (or binary) file, byte for byte, or as an ASCII file, converting the line-end character as necessary to compensate for different operating systems (UNIX uses only a linefeed character at the end of lines; MS-DOS uses carriage return/linefeed). Before beginning a file transfer, enter **type i** for an image file, or **type a** for an ASCII file, at the ftp> prompt.

What are the consequences choosing the wrong transfer type? Transferring a binary file as type **a** will almost certainly fail. Transferring an ASCII file as type **i** will work, but you may find that the line-ends are screwed up. ASCII transfers are also quite a bit slower than image, because each line needs to be processed separately.

To start a file transfer, use the command **put <filename>** to send a file, or **get <filename>** to receive one. In both cases, the newly created file will have the same name as the original file. This can cause a problem if you're transferring from a system that supports long filenames (like UNIX) to one that doesn't (like DOS). If you need to change filenames, you can add a second filename to the command (such as **put <localfile> <remotefile>**), and the transferred file will be named <remotefile>.

ftp takes things quite literally and one source of frustration is the way it tries to force remote filenames into the local filesystem. In addition to the long filename problem, specifying a directory as part of the remote filename given to an ftp server can cause unexpected problems. For example, if your local directory is c:\tmp and you issue the ftp command `get pub/nos/docs/faq.txt`, ftp will try to create "faq.txt" not in c:\tmp, but rather in c:\tmp\pub\nos\docs\, and will complain loudly if that directory doesn't exist – and it won't even try to create the missing directory.

You can avoid ths problem by specifying a destination name in addition to the source name. The command `get pub/nos/docs/faq.txt faq.txt` will have the desired effect of creating faq.txt in the current local directory. Alternatively, you can change into the remote directory and then specify only the filename in the `get` command.

If you want to put or get multiple files that match a wildcard, the commands **mput <filespec>** and **mget <filespec>** will do the trick. **mget *** will grab all the files in the remote directory and transfer them to your system. File names in ftp transfers can include a full path if you desire; remember to use the proper path separator character for the remote host. By default, `mget` will ask you before it downloads each file. To avoid this, you can issue the `prompt off` command

If something goes wrong during a file transfer, you can terminate it by going to the command session via F10 and issuing the **abort <session number>** command, where "session number" is the ftp session (this will only work if the ftp session is still listening for your command). To end an ftp session, you can either type **quit** at the ftp> prompt (the preferred way), or you can close the session from the net> prompt.

If you want others to be able to access files on your system, you'll need to set up an FTPUSERS file in your root directory as described above, establish the file directory structure (most folks use /pub or /public as the directory to hold publicly accessible files), and make sure you've included the command **start ftp** in AUTOEXEC.NOS.

A note on channel courtesy: transferring files via ftp is reliable, but can be slooooow, particularly at 1200 baud. And, downloading a big file can keep the channel busy for a long time. Before you start downloading a 250 kilobyte file, consider how busy the channel is, and whether you want to tie things up for (perhaps) several hours by your download. NOS is polite and won't hog the channel, but don't doubt that a large file transfer will slow things down for everyone else.

PING

The **ping** command uses **ICMP** (**I**nternet **C**ontrol **M**essage **P**rotocol—a special protocol that IP systems use to manage traffic flow) packets to test a path or to see if a remote host is on the air. Just enter the command **ping <hostname>** at the NOS prompt. If the host is available, you will see a response indicating what the round-trip time was to that host. The time may be many seconds if you're going through gateways, so be patient.

You can also tell ping to send off multiple packets by specifying the time between pings, and the amount of data to include in each packet. The command **ping n8ur 128 5000** will send a packet of 128 bytes to host *n8ur* every 5000 milliseconds (5 seconds).[10-2] In this mode, you can tell ping to send packets as large as the interface's MTU, minus 40 bytes of IP and TCP headers. When you start a repetitive ping session, the results will be displayed in a session screen that shows the cumulative number of packets sent and received, as well as statistics about the round trip time. If you send just a one-shot ping, the result will appear in the command session.

Hop

hop is similar to ping, but it shows each system that an ICMP packet goes through to reach its destination. It's very useful for determining the route between two stations. The syntax is **hop <remote host>**. The **command hop maxttl <hops>** specifies how many hops to count before giving up; setting this to a reasonable value keeps the hop command from going in circles if it encounters a routing loop.

[10-2] *Be careful with repetitive pings. It's easy to flood a channel if you make the packet size too large or the repetition rate too fast.*

Finger

The **finger** protocol lets you see information about a remote host's users and services. Entering **finger @<hostname>** (note the slightly different syntax—the **@** symbol must immediately precede the remote hostname) will display a list of the finger files (described earlier) at that host. Entering **finger <user@hostname>** will display the text file for that user. Recent versions of NOS also allow you to obtain system statistics via finger. The **finger @<hostname>** command should show you the commands that you can finger.

Monitoring The Channel

NOS provides a trace session that allows you to watch traffic flowing on any interface. The syntax is **trace <interface> <options>**, where the option setting is a three digit number. The first digit indicates the trace mode; the second and third indicate whether to monitor incoming and outgoing packets, respectively (1 means monitor, 0 means don't monitor). The display formats are: for a first digit value of 0, decode headers but don't display data; for a value of 1, fully decode the whole packet in hex and ASCII; for 2, fully decode the headers but display text in ASCII only; for 3, display in a format similar to the monitor mode of a TNC.

I've found display format 3 to be the most useful for general monitoring, so to trace my UHF interface for packets sent by other stations, and packets I send, I issue the command **trace uhf 311**. The output of the trace command goes to a separate session screen; press F9 to switch to that session, and F10 to return to the command session. You can also send the output of the trace session to a file by adding a filename at the end of the command:

```
trace uhf 311 trace.txt
```

To stop tracing, issue the command:

```
trace <iface> 0000
```

Chapter

11

Electronic Mail

We've saved NOS' electronic mail capabilities until now because they are a bit more involved than some other parts of the program. NOS acts as a **mail transfer agent** ("MTA"), a tool that handles the delivery and receipt of messages, but it doesn't actually have the means to enter or read messages—there's no "mail" session. To do that you need a **mail user agent** ("MUA"), and you have a couple of choices. You can use an external mail program like **BM** or **PC-ELM** to handle mail, or you can use the NOS mailbox system as a crude MUA.

If you use the NOS mailbox, the tools you use for entering and reading messages will be essentially the same as on any PBBS system; there's not much more to say about it here. As an example of an external MUA, I'll talk about BM, which Bdale Garbee, N3EUA wrote for use with NOS. It's simple but it does the job, and it has the advantage of being small enough that you can usually use it from a DOS shell within NOS. It's also quite similar to the UNIX "mailx" command, so this is another place where learning NOS will help you learn UNIX. A discussion of BM demonstrates several important concepts that all MUAs have in common. *Even if you don't use BM, the information in this section is important if you're planning to use the email capabilities of NOS.*

BM Basics

BM.EXE is a program that reads and writes mail message in the format TCP/IP systems recognize. Contrary to popular belief, "BM" stands for "Bdale's Mailer" in honor of its creator. You can run BM from the DOS prompt just like any other program, or from within NOS by shelling to DOS with the **!** or **shell** commands; some versions also support a **mail** command that is the equivalent of shelling to DOS and running BM. Remember that when you shell out, NOS is usually in suspension and is not responding to packets. Staying in this mode too long can result in NOS crashing when it runs out of room to buffer unprocessed packets.

Before using BM, you need to create its configuration file, **BM.RC**, which must live in the root directory of your disk (note: changing your NOS root directory doesn't affect where BM looks for BM.RC — you'll need to make sure that file is always in your disk drive's root directory). Here's the layout of BM.RC.

First, you should set your hostname, which should match the hostname assigned to your NOS system:

```
host n8ur.ampr.org
```

You also need to set your user name, which may be a bit touchier. Some folks recommend using either your first name, or your initials (for example, my address would be "jra@n8ur.ampr.org") while others suggest using your callsign instead ("n8ur@n8ur.ampr.org"). I prefer using a callsign for the mail name because doing so has major advantages if you want to interface with the PBBS mail system. You can also set up an **alias** (described later), which will allow mail sent to any of a number of addresses to end up in your mailbox.

Whichever method you choose, it's important to be consistent within the local area, so that everyone knows how to address mail to everyone else. Even if there's local agreement, it's a good idea to use aliases so that mail addressed to your name, your call, "postmaster", "sysop", etc., are sent to the mailbox where you actually read your mail.

Set the user name with this line in BM.RC:

```
user n8ur
```

Next, tell BM your full name to use in the "From:" line of your messages:

```
fullname John Ackermann
```

If you want to have replies sent to another host because, for example, you are using a POP server, this line specifies where replies should go:

```
reply n8ur@w8apr.ampr.org
```

If you are using a mono video care in your PC, you can tell BM to use direct video writes for more speed:

```
screen direct
```

BM's built-in editor is pretty rudimentary. You can specify an external editor that BM should start when you want to compose a message:

```
edit  c:\dos\edit
```

will establish the DOS edit program as the editor that BM will use.

You can specify folder to store your saved mail; note the forward instead of back slashes in the file names:

```
folder c:/folder         # directory to hold folders
mbox c:/folder/mbox      # saved mail folder
record c:/folder/outmail # sent mail folder
```

When you start BM, you'll see a prompt such as **n8ur>** showing the default mailbox (based on the **user** entry in BM.RC). As in NOS, you enter commands at the prompt, following them with a carriage return. Most BM commands are single letters, optionally followed by a mail addressee or a message number (or numbers).

To send mail, use the command **m <addressee>**. The addressee will normally be a user at a remote host; for example, n8ur might send mail to k8gkh@k8gkh. The most common mistake people make with BM is *forgetting to include the hostname*—in other words, sending mail to <user> rather than <user>@<hostname>. Without the hostname, BM will think the user is on your local system, and the message will end up being stored in a mailbox under that user's name on your own system. That doesn't work too well, and BM doesn't give you any warning that it's happening.

The Alias File

One way to solve that problem, and do some other interesting things, is to create an **ALIAS** file in your root directory (again, note that BM won't find ALIAS in a \NOS root directory). When you send or receive a message, NOS will compare the addressee with the alias file, and if it finds a match will replace the alias with a full address from the file. An alias can point to a list of addresses, so it's possible to define an alias that will send a copy of the message to everyone in your local group. A sample alias file might look like:

```
greg     k8gkh@k8gkh.ampr.org
bill     n8kza@n8kza.ampr.org
club     k8gkh@k8gkh.ampr.org n8kza@n8kza.ampr.org
         n8acv@n8acv.ampr.org wb8gxb@wb8gxb.ampr.org
john     n8ur@n8ur.ampr.org  n8ur
```

The left side of the alias entry must be a local user name; you can't do this:

```
n8ur@w8apr.ampr.org     n8ur@n8ur.ampr.org
```

The alias for “club” demonstrates two things: a single alias can expand to several addresses, and you can continue a long address list on subsequent lines by indenting them with spaces or a tab character. With these alias entries, mail sent to “greg” will automatically be expanded to Greg’s full address, and sending a message to “club” will result in all four users getting a copy. That alias shows that a copy of any message for "john" will go to my home system, and a copy will also be left in a mailbox called "n8ur" on this computer. This setup is very useful for NOS systems providing PBBS services – you can arrange for a copy of each message to be delivered via smtp, as well as stored as a traditional BBS message.

By the way, you do not use a trailing dot after an FQDN (as discussed earlier) in email addressing; doing so will screw things up.

Aliases can cause big problems if you create a routing loop. For example, if n8ur.ampr.org has an entry like

```
daytcp   daytcp@w8apr.ampr.org
         n8ur@n8ur.ampr.org
         ke8tq@ke8tq.ampr.org
```

in its ALIAS file, and w8apr.ampr.org has one like

```
daytcp   daytcp@n8ur.ampr.org
```

in its file, problems are going to happen when a message addressed to “daytcp” is received at either machine. If a message is sent to “daytcp@n8ur.ampr.org,” n8ur will follow the instructions in its ALIAS file and send a copy to daytcp@w8apr.ampr.org, which will follow the instructions in its own ALIAS file and immediately copy the message back to daytcp@n8ur.ampr.org, and so on until one of the machines crashes. It’s not a pretty sight.

The simplest way to avoid this problem is to use only one host on your local network to handle mailing list aliases. The other systems then use their alias files only to handle local needs. Since only one host is using aliases to send mail to other users, the risk of creating mail loops goes way down.

Using BM

If you use BM's built-in editor to compose messages, remember that it doesn't wrap lines; you have to hit the carriage return at the end of each line. To stop editing a message, enter a period by itself at the beginning of a line. You'll be returned to the BM prompt. You can read an existing file into the BM editor with the **-r <filename>** command.

Use the **l** command to list outbound mail; you can kill an outbound message with the **k <msg#>** command, using the message number obtained from the **l** command.

Several commands are used to deal with incoming mail. **h** displays the headers (summary info) about messages in your mailbox. It is the basic command you should use to check your incoming mail. Each header displayed includes a message number to use with the other message manipulation commands. Commands given without a message number act on the current message (the one marked with an ">" in the display from the **h** command); if there's only one message, it is always the current one.

NOS and BM can support multiple users at a single host; a separate mailbox is created for each user. Unfortunately, NOS has no way of knowing if incoming mail addressed to <someuser>@<yourhost> is valid, so it will happily accept such mail and create a new mailbox for <someuser>. You may never know it's there, unless you use the **n** command to display the list of mailboxes. You can also use **n** to change to a different mailbox: **n <mbox>**.

The commonly used commands (which may be followed by one or more message numbers if appropriate) are:

```
msg#  message number by itself will display that message and set
      it as the current message.
r     reply to a message.
d     delete a message.
s     save a message; if a file name follows the message number(s),
      the message(s) will be saved in that file.  Otherwise,
      they'll be saved in the default mbox file.
u     undelete a message previously marked for deletion.
p     print a message on the local printer.
w     save a message to a file without including headers.
f     forward a message to another recipient.
b     bounce a message.  Like forward, but keeps the original
      sender information intact (i.e., the message will not
      appear to have been sent by you).
$     update the mailbox.  This deletes messages marked for
      deletion and reads in any new mail that may have arrived
      since you started BM.
```

There are two commands that exit from BM: "**x**" will exit without updating the mailbox. In other words, the same messages will be there the next time you run the program. "**q**" updates the mailbox (like "**$**") and then exits.

Mailbox files are stored in spool\mail\ with names in the form <username>.txt. Outbound mail created by BM is stored in the \spool\mqueue directory, where it waits patiently until the SMTP server attempts to send it to its destination.[11-1] Each outbound messafe creates two files: one with a .txt extension that contains the message itself, and another with a .wrk extension that contains delivery information. The filename is based on the system message number.

[11-1] *If you're acting as a POP server, the mail will be stored in the end user's mailbox, not in the mqueue directory.*

Moving Mail With NOS

Now, to the mechanics of getting mail into and out of your system. All mail that you create is sent to its destination (or at least to the next stop on the way) by the **smtp** server in NOS. The **smtp timer** command tells smtp how often (in seconds) to scan the spool/mqueue/ directory for outgoing mail. When it finds some, it attempts to open an smtp session to the remote host in the address and send the mail there. There's no default for the smtp timer value, so your AUTOEXEC.NOS file should include something like **smtp timer 600** (which scans for mail every ten minutes). You can manually force smtp to scan the queue by issuing the **smtp kick** command from the command session.

If you have a local mail server with connections to the outside world, you can use it to route mail for hosts that aren't in your domain file with the **smtp gateway <hostid>** command. As mentioned above, it makes sense to appoint one system on your LAN (preferably one that's up full time) as the local mail server, which handles mailing lists, acts as a POP server, and perhaps has an email gateway to the Internet. Without a gateway, mail bound for hosts which you cannot reach will simply remain queued up indefinitely; by pointing outbound mail to mail gayeway that operates full time, and using POP (described on the next page) to receive mail, your mail won't depend on your machine being up all the time.

Using POP to Process Mail

Incoming mail arrives at your station when a remote host starts an smtp session with you. If you don't keep your station up 24 hours a day, the other host will be trying, and trying, and trying, to connect with you until you finally show up. This wastes channel resources, and memory on the remote host. A far better approach is to use **POP**—the **P**ost **O**ffice **P**rotocol. If your system runs POP, and someone in the area has agreed to be a POP server, NOS will automatically contact that server when you come on the air; the server will respond by sending the mail waiting in your mailbox. You can then read it with BM just as if it had arrived via smtp.

To use POP, the server must establish a username and password for you in its FTPUSERS file, and needs to have its POP server started. You need to add the appropriate command to your AUTOEXEC.NOS file:

```
popmail addserver <server> <timer> <protocol>
<mailbox> <username> <password>
```

The "server" is the hostname or IP address of the POP server. The "timer" protocol is how often, in seconds, to poll for mail. If this value is not set, you'll have to manually request mail using the "pop kick" command. The "protocol" parameter is either "pop2" or "pop3" depending on the version of the protocol running on the POP server. POP3 supersedes POP2, and it's recommended that servers run that version. The "mailbox" parameter is the name of the local mailbox (the file in spool/mail/) into which mail will go; "username" is your mailbox name on the POP server; and "password" is your password on the server.

For mail addressed to you to be sent to the POP server and delivered to you, your correspondents must address their messages to your user name at the POP server, *not directly to your host*.

Remember that smtp or POP sessions may be running in the background without your knowing about it. Always check for activity with the **sockets** or **tcp status** command before pulling the plug!

Additionally, smtp creates lock files in spool/mqueue when it tries to send outgoing mail. If NOS is killed before the mail transfer has succeeded, these files (with the extension ".LCK") will be left behind and if they are not manually removed, they will prevent smtp from trying again to send those messages. To prevent this, you should always issue the command "erase spool/mqueue/*.LCK" before starting NOS. It's a good idea to launch NOS using a batch file that removes the locks before executing the program.

Chapter

12

Using NET/ROM

NOS includes support for NET/ROM, probably the most popular packet radio networking protocol. You can use an existing NET/ROM network to carry TCP/IP traffic if a native IP network isn't available, but you don't need NET/ROM to make NOS work, and you shouldn't even configure it unless you need to.

Why not? There are at least four reasons. First, IP is a complete networking protocol on its own. It doesn't need something like NET/ROM to route its packets. Second, running TCP/IP over NET/ROM isn't very efficient. It adds additional overhead, and because NET/ROM is limited to packet lengths of less than 256 bytes, you can't use larger MTUs to increase throughput. Third, IP routing over NET/ROM isn't automatic. You need to manually configure routing commands and other parameters in AUTOEXEC.NOS for each IP gateway you want to reach via NET/ROM (and the station at the other end needs to do the same). Finally, NET/ROM networks—especially large ones—are somewhat fragile, and the willy-nilly addition of new nodes can cause problems, particularly if they aren't configured consistently with other nodes on the network. It's also worth noting that if you compile your own NOS.EXE and exclude the NET/ROM code, you can save about 40k of memory overhead.

When *should* you use NET/ROM? The classic example is two local IP networks that can't reach each other except via an existing NET/ROM backbone. In that case, each IP network should establish one host to act as a gateway to the NET/ROM network. This host – and this host only – turns on its NET/ROM server and provides access to the NET/ROM backbone for all hosts on the local network. Those hosts route their outbound packets to the gateway as described above. They don't even need to know that the gateway is using NET/ROM to route its packets.

Each gateway needs routing statements and manual arp entries for each of the other gateways on the NET/ROM backbone.

Configuring NET/ROM

NET/ROM runs as a server in NOS. It creates a "virtual interface" that operates on top of one or more of the physical hardware interfaces. You must configure a callsign and a NET/ROM alias for the system, as well as specific parameters that control NET/ROM's behavior. Finally, for each IP system at the other end of a NET/ROM link (hopefully, only a single gateway) you must establish a route and add an entry to the arp table.[12-1] Here are commands that might be used in AUTOEXEC.NOS to configure a TNOS NET/ROM switch (beware, JNOS has some minor differences in syntax):

```
# NET/ROM setup commands

# Start the NET/ROM server
start NET/ROM

# Attach the pseudo-interface
attach NET/ROM

# Set the callsign and alias
NET/ROM mycall W8APR-2
NET/ROM alias #DAYIP
```

[12-1] *The manual arp entry is required because arp request messages won't propagate over a NET/ROM path. NOS needs to be explicitly told the callsign of the NET/ROM-to-IP gateway at the other end.*

```
# Enable NET/ROM on the “uhf” physical interface with
# a quality of 192.  The “n” means that this station
# will broadcast only its own node information, and not
# the full contents of its routing table.  An IP-to-
# NET/ROM gateway that doesn’t extend the NET/ROM
# network should set the “n” flag to avoid unnecessary
# routing broadcasts and possible routing problems.

netrom interface uhf 192 n

# Set the minimum quality required to add an entry to
# the node table.  The value you use will depend on how
# many nodes are in the network, and how many of them
# are usable.

netrom minqual 140

# Set various protocol parameters
netrom timertype linear
netrom acktime 8000
netrom obsotimer 1800
netrom nodetimer 1800
netrom window 3
netrom irtt 3000
netrom ttl 16
netrom tdisc 600
netrom retries 5

# Miscellaneous commands
# Display only nodes with aliases that <don’t> begin
# with “#” in node lists.  Nodes named this way are
# generally not intended for end-user access.

netrom hidden ON

# Configure IP routes.  Each route needs a manual
# arp entry to map from the IP address of the remote
# NET/ROM-to-IP gateway to its callsign.

route add [44.71.36.0]/24 netrom 44.71.36.1
arp add 44.71.36.1 netrom w8cqk-2

# Start things off by broadcasting our presence.
netrom bcnodes
```

Some NET/ROM Details

There's a common misconception that NET/ROM routes have to be manually defined in NOS. That's not true, though you may want to hard code them into your AUTOEXEC.NOS file for other reasons (and I'll describe how to do that in a minute). NET/ROM is an automatic routing protocol, and NOS takes advantage of that fact. All you need to do is use the "netrom" keyword in your routing and arp commands, and NOS will figure out the rest.

However, one minor problem with NET/ROM's automatic routing is that your station won't know what the available routes are until it has received nodes broadcasts that include the remote gateway's routing information. JNOS and TNOS automatically request neighbors to broadcast their routing tables when the NET/ROM interface is started, but not all varieties of NET/ROM switches will respond to that request, and it may take a while to receive routing information from one of these uncooperative nodes. There are two ways around this problem.

First, as I hinted above, you can manually define a NET/ROM route with a command in AUTOEXEC.NOS like this:

```
netrom route add #cmhip w8cqk-2 uhf 192 w8rmj-1
```

This creates a route for the remote node #cmhip[12-2] with callsign w8cqk-2, using a route quality of 192. The last entry – the "relay" call – is the callsign of the NET/ROM neighbor who knows how to reach #cmhip; it must be reachable via the specified interface (here, "uhf"). By the way, a manual NET/ROM route is also useful if you need to reach your NET/ROM neighbor via a digipeater. You can add the digi's callsign after that of the neighbor (of course, the NET/ROM switch at the other end will also have to be configured to reach you through that digi as well):

```
netrom route add #cmhip w8cqk-2 uhf 192 w8rmj-1 n8jxa
```

Here, we'll use n8jxa as a digipeater to reach w8rmj-1, the relay station.

[12-2] *Node aliases that begin with a "#" sign are normally hidden from node listings. It's a good idea to use this feature to hide nodes, like most NET/ROM-to-IP gateways, that aren't intended to be connected to by end users.*

The other way to get routing information more quickly is to save the NET/ROM routing table each time you exit NOS, and reload it when you restart the system. That's easy to do with the **netrom save** command, which will save the routing table in a file called **spool/netrom.sav**. The **netrom load** command will reload that file and recreate the routing table as it existed when it was saved. You can insert the netrom save command in an **onexit.nos** file (remember, that's the command file that NOS executes just prior to shutting down), and you can put the **netrom load** command in your autoexec.nos file after you've configured the NET/ROM system.

The danger of reloading a saved routing file is that some of the routes may have changed since the last time the file was saved. Another risk is disabling the save command without realizing it. This caught us once, when our onexit.nos file mysteriously disappeared, but the old netrom.sav file was still hanging around. Every time we started NOS, the old file was reloaded, and was the source of all sorts of obsolete routing information that caused havoc on the network.

I mentioned above that NET/ROM imposes a limit on the size of TCP/IP frames. The largest packet a standard NET/ROM system can pass is 256 bytes; anything larger is fragmented, and the nature of that fragmenting results in great inefficiency for TCP traffic. Because of the 40 bytes of overhead added by the IP and TCP headers, the largest TCP segment that can travel over a NET/ROM link is 216 bytes. Current versions JNOS and TNOS automatically set the MSS for any route that travels via NET/ROM to this value, so you don't need to expressly address this limitation in the gateway's configuration files.

However, the NET/ROM MSS limitation spreads beyond the gateway system. Any station that is routing traffic through the gateway should also limit its MSS to 216 to avoid fragmentation. As a consequence, LANs that rely on NET/ROM for more than occasional routing should consider standardizing on a 216 byte MSS for all hosts. That will maximize efficiency for traffic that travels via NET/ROM, at the cost of efficiency that could be gained on the local network through a larger MSS value. You'll have to weigh the tradeoffs to see which is more important in your environment.

More on Getting Along

As you can see from the example configuration file, NET/ROM provides plenty of opportunities for parameter tweaking. You can control the frequency of nodelist broadcasts, how quickly nodes are aged out of the nodelist, and the minimum node quality the system will accept. You can lock in routes and neighbors, and can adjust the netrom irtt to adjust retry rates. But don't go wild playing with the NET/ROM settings. It's critical that you coordinate closely with the folks managing the NET/ROM nodes with which you'll be connecting, because one improperly configured node can bring a network to its knees.

Unfortunately, NOS is looked at with suspicion by lots of other packet operators, and NET/ROM node operators are no exception. If any problem shows up on the network after a NOS system appears, it's NOS that will be blamed, rightly or wrongly. So, work with the local NET/ROM folks to make sure your configuration is consistent with the way they do things.

TheNET X1 Nodes

Just as a note, there's a freeware clone of NET/ROM called TheNET which runs in TNC-2 clones. Recent versions (X1J and higher) support IP routing. If you have these nodes as part of your local NET/ROM network, you have a head start toward building an IP network—all you need to do is convince the nodeop to turn on and configure the IP functionality. With a TheNET node to act as a gateway, local NOS users don't have to worry about running NET/ROM at all. There aren't any X1 nodes in our area, so I can't describe how to interface to them, but the reports I've received indicate that they work well as IP routers.

Chapter

13

Using NOS as a PBBS

From the beginning, NOS has had a simple mailbox that allowed AX.25 users to connect to the system and read messages, both private and public. Long ago, SM0GRV upgraded the mailbox so that an AX.25 user connecting to it sees something very much resembling a W0RLI-style PBBS, and added rudimentary message forwarding capabilities.

JNOS and, especially, TNOS go further and bring to their versions of NOS all the functionality of a complete PBBS system. Although NOS was never envisioned as "Yet Another Packet BBS," it now has the capacity to be just that. At the extreme, you can argue that NOS PBBSs perpetuate an inefficient kludge of a messaging system. Moving a bit closer to the middle of the road, I prefer to think of the NOS PBBS as a link between the PBBS and TCP/IP worlds, and a gentle path to the greener pastures of a TCP/IP world. I'll leave the philosophical arguments to others and just point out that quite a few full-service PBBSs are running today using these NOS versions.[13-1]

Configuring and running a NOS PBBS is not a simple undertaking; it's adding all the complexity of a sophisticated program to an already complex and sophisticated program! All I can hope to do here is describe the basics. Please look to the reference manuals and an Elmer for more details.

[13-1] The F6FBB PBBS software has now been ported to Linux, and it provides another way to integrate a BBS with a TCP/IP system. Configuring FBB is beyond the scope of this book, but we'll mention it again when we talk about Linux AX.25 capabilities.

The NOS mailbox subsystem handles three separate sets of functions, all of which are highly configurable. First, it establishes the capabilities, such as reading and sending messages, downloading files, etc., that are made available to users, and it provides security functions via the FTPUSERS file and security commands to control what users can and can't do with the BBS. Second, it manages incoming and outgoing messages. This includes organizing messages into public and private message areas via the AREAS and REWRITE files, and expiring old messages using the EXPIRE.DAT file. Finally, it handles the incoming and outgoing forwarding of messages to other PBBSs through the FORWARD.BBS file, and preventing the spread of duplicate messages via the HISTORY file that it maintains.

NOS PBBS Hardware Requirements

Although NOS itself doesn't require too much hardware horsepower (other than plenty of DOS RAM), using it as a PBBS requires more resources, and if you're thinking of running a NOS PBBS, you should pay some attention to your hardware selection. Though NOS will run on pretty slow computers, PBBS service puts heavy requirements on the machine, and particularly on the hard disk system—although in fairness, most other PBBS programs probably have similar real-world requirements for good performance. My suggestion, based on lots of experience with JNOS and TNOS, is: invest your money in a fast hard disk, and if you're running NOS under DOS, buy a few megabytes of memory for RAM disk to store temporary files.

The CPU horsepower isn't much of an issue, and even an ancient 386/40 or 486/20 motherboard will work fine. You don't need a big hard disk (40MB is more than adequate), but you should try to find a drive with a fast access time and a fairly speedy disk controller. You should also use 16550 buffered UARTs on your serial ports to reduce the interrupt load on the system. As discussed earlier, they are a cheap way to greatly improve your NOS box's I/O performance.

JNOS creates lots of temporary disk files when users change from one message area to another, and during forwarding and expiration cycles. Most of the performance problems come from reading and writing these files. At W8APR, we originally tried a disk cache of several megabytes, but found that using the memory for a RAM disk provided better performance. We told JNOS to store its temporary files there (by setting a DOS environment string of **TEMP=D:** in **autoexec.bat**), and that gave much better performance. Our users' biggest complaint—the time to change from one message area to another—disappeared.

The trick is making sure you have a large enough RAM disk to hold all the temporary files that will ever be needed at any one time. We found that a 3MB RAM disk worked, but was sometimes dangerously close to full. With the price of memory today, you should use at least 8MB of RAM (which allows a 7MB RAMDISK) to allow plenty of room for temporary files; in fact, with memory at less than $1.00/MB, you could probably run the whole system from 32MB of ramdisk.

Another point to remember in a DOS-based NOS PBBS is that the hard disk may get very fragmented after the system has been running for a while. It's well worthwhile to run a program like the DOS "DEFRAG" utility every now and then to clean up the hard disk. Since NOS beats on the disk pretty heavily, it's not a bad idea to check the disk integrity with something like SCANDISK at the same time.

TNOS for DOS uses a somewhat different process for handling mailbox messages; it provides better performance but still benefits from a RAM disk and regular hard disk maintenance. In my opinion, though, the best alternative right now for an NOS-based PBBS is TNOS or JNOS running under the Linux operating system, as described in the next chapter. Performance is better, and the biggest problem with any DOS version of NOS – memory cram – disappears.

Configuring the BBS

To work properly as a PBBS, NOS requires you to set several commands and parameters within AUTOEXEC.NOS, and to create and edit several configuration files.

It's time for another warning: when you set up your NOS system as part of the PBBS forwarding network, you are interfacing with other systems using other software. The forwarding network is a fragile thing, and misconfigured PBBS systems can muck things up very quickly. Unfortunately, NOS has gotten a bad (and undeserved) rap as a problem child in the PBBS world, and any time there are problems with duplicate bulletins or other screwups, any NOS system in the forwarding path is going to be blamed.

It's important, not only for your reputation but for the reputation of NOS, that your system be properly configured before you start doing live forwarding. Test with a local system first, and be aware that compatibility with some other PBBS software may take configuration tweaking on your part. Take heart, though: properly configured, current versions of JNOS and TNOS are perfectly capable of being good neighbors to all the other PBBS programs out there.

That said, let's move on to the NOS configuration files that you'll need to build for PBBS service.

AUTOEXEC.NOS

The basic settings that tell the PBBS subsystem who it is, and how it should work, are contained in AUTOEXEC.NOS. Here's a sample of the PBBS configuration commands for TNOS (by the way, this sample is based on the one generated by the mksetup.tcl program included with TNOS version 2.10):

```
# Start the AX.25 server for incoming connects
start ax25

# *** Set the basic PBBS functions ***
# This lets people page the operator
pbbs attend on

# Prevent people from sending bulletins without
# including an "@host" part
pbbs enforce on

# The message of the day that all users get.
motd "Welcome to w8amp.ampr.org!"

# The message AX.25 users see.
pbbs motd "Welcome to TNOS at w8apr.ampr.org!"

# The message telnet users see.
pbbs tmsg "Welcome! Please login with your call sign!
Use any password!"

# The jumpstart command tells TNOS to bring up the
# pbbs menu immediately upon receiving a connection
# to the AX.25 mycall.  If this is turned off, the
# user will have to hit a carriage return before
# getting the menu.
pbbs jumpstart on

# Time out users who don't do anything in 10 minutes.
pbbs tdisc 600

# An alias callsign for the BBS.
pbbs alias MVFMA

# Password for remote sysop acceptance.
pbbs password averylongpasswordwithmanyletters

# Log PBBS activity.
pbbs log on
```

```
# Complain if a message is entered without a subject.
pbbs nosubjbell on

# Hold all locally generated mail for review.
pbbs holdall on

# Hold all locally generated BULLETINS for review.
bulletin hold on

# Don't send messages about permission infringements
# and system errors to "sysop" – if left on, this can
# generate a lot of messages!
errors off

# This is a full-service BBS (not a PMS).
pbbs fullservice on

# Allow third-party traffic.
third on

# *** Bulletin Handling ***
# "On" means date associated with bulletin is date
# originated; "off" means it's the date received.
bulletin date on

# Optimize forwarding by scanning the R: lines so we
# don't forward to systems that have already received
# the message.
bulletin check on

# Bulletin return address is from last R: line.
bulletin return on

# *** Mailfor beacon options ***
# Calls listed here will be excluded from "mailfor".
pbbs mailfor exclude w8apr sysop n8ur

# Send mailfor beacon every 20 minutes.
pbbs mailfor timer 1200

# Display a "new mail" message when stuff arrives for
# these areas/users.
pbbs mailfor alert mode on
pbbs mailfor alert mbox N8UR

# Tell the TNOS status line to display a "!" to the
# left of the date line when mail is in the hold file,
# and a "@" when "alert" mail is waiting.
pbbs mailfor hold mode on
```

```
# *** PBBS forwarding parameters ***
# Your hierarchical address, minus callsign.
forward haddress #DAY.OH.USA.NOAM

# Use bulletin IDs in forwarding.
forward bid on

# Use FBB-style forwarding.
forward fbb-style on

# Use FBB compressed forwarding.
forward fbb-compression on

# Forward mode allows different forwarding strategies
# to be set in FORWARD.BBS.  If set to “0”, forwarding
# will be disabled.
forward mode 1

# “On” includes smtp headers in forwarded messages.
forward smtp on

# Callsign for outbound forwarding sessions.
forward mycall W8APR-7

# Local time is this many hours from UTC (i.e. +1, -4).
forward utc -4

# *** White Page commands.***
# Enable white pages
wpage client on
wpage server on

# Keep WP entries for this many days.
wpage age 75

# The last thing to do is start the forwarding
# timer and kick off an initial forwarding run.
forward timer 300
forward kick
```

There are a few dozen other subcommands in the pbbs, bulletin, and forward menu trees, but their default values are generally acceptable. The commands shown above are those that are either particularly important, or offer interesting customization possibilities.

Managing the Mailbox with the REWRITE and AREAS Files

One of the unique features of NOS as a PBBS is its ability to sort messages into different **areas** based on addresses. This means that all messages addressed, for example, to "TCPIP@USA" can be stored in a single area, and users can see all these messages grouped together when they view this area. Two files manage this process: **REWRITE** defines the rules by which messages are dumped into areas, and the **AREAS** file defines the "public" areas that users may access.

Areas

Let's talk about areas first. An **area** is a single mail file that contains all the messages addressed to a single user. NOS puts messages in areas following rules contained in the REWRITE file. If REWRITE doesn't have a rule that matches a specific address, NOS will get confused.

Areas for user messages are private—only the user who matches the area name (or the sysop) can access them. Private areas aren't very useful for bulletins and other messages of general interest, though, so NOS allows for public areas as well. Any user can read the messages in a public area. You can make an area public by putting its name, optionally followed by a short description, in the AREAS file. A simple AREAS file might look like:

```
help      Help messages for this system
tcpip     Messages about ham tcp/ip
packet    Messages about general packet radio
junk      Messages addressed to "ALLUSA"
```

Non-sysop users have access to their private area, where mail addressed to them is placed, as well as to all the areas listed in AREAS.[13-2] When connecting to the mailbox, a user starts out in his private area where he can list and read his personal messages.

[13-2] *There's another file called AREAS.SYS that controls the areas available to users with sysop privileges. This is useful because the message expiration process only works on areas that are listed in one of the AREAS files. AREAS.SYS allows you to create an area, for example, to hold all the "SYSOP@" quasi-bulletins and keep them private, yet automatically expire them after they age.*

The "A" command in the mailbox is used to change areas (entering **a tcpip** at the mailbox prompt changes the user to the tcpip area). The "AF" command lists all the public areas and their descriptions.

The Role of the Rewrite File

How do messages get routed into the area files? The REWRITE file handles that process; it's the map that tells NOS where to put incoming messages, whether they arrive via smtp or via the packet BBS network (actually, the NOS smtp system processes all mail, regardless of where it originates). Though REWRITE can become a long and cryptic looking file, the basic concept is simple. Each line in REWRITE has at least two parts:

```
<template>                <rewritten address>
```

Every time a message comes into the system, the NOS smtp process matches its address against the rules in the rewrite file, starting at the top; when an address matches a template, NOS replaces (or rewrites) that address with the one on the right side, and stores it in the area that matches the rewritten address.[13-3] For example, if NOS receives a message addressed to "TCPIP@ALLUSA" and the rewrite file includes only the line:

```
tcpip@allusa   tcpip
```

NOS will deposit that message in an area named "tcpip" (an area is actually nothing more than a disk file named after the rewritten address, plus a ".txt" extension). If there's an entry for "tcpip" in the AREAS file, that area will be publicly available, and any user who changes to it will be able to read the message.

[13-3] *The rewrite process doesn't physically change the address contained in the message – it only changes the* ***envelope address****, or the address NOS is using when it processes the message. After rewriting, the message headers remain just the same as they were before.*

The goal of the rewrite process is *to ensure that every incoming message address matches a rule*. No message should "fall through" the rewrite file without finding a match, and every match should result in the message being put in the proper area.[13-4]

Wildcards, Variables, and Recursion

Three features make REWRITE a powerful tool. First, it supports wildcards (the "*" character in a template matches a string of zero or more characters), so it can match lots of different addresses with relatively few rules. For example, consider this rewrite rule:

```
*@allusa*       junk
```

This template can be broken into three parts. The first "*" is a wildcard that matches all the characters in the incoming address up to "@allusa" (from now on, I'll refer to the non-wildcard strings in a template as **separators**). The separator is the second part. Finally, the second "*" matches any characters to the right of the separator, forming the third part of the template.

To score a match in REWRITE, the incoming address must match the pattern set by the template. Any character, or no character at all, will match a wildcard. The separator must be matched exactly in the incoming address. In the example, "tcpip@allusa.usa" would match – "tcp" satisfies the first wildcard, and ".usa" matches the second. "@allusa" matches the separator.

However, "tcpip@allus.usa" would not match, because the separator requires an exact match on "@allusa". If you wanted to make both "@allusa" and "@allus" match, just shorten the template by one character to "*@allus* — now both versions will match the shortened separator.

[13-4] *That's not quite true. The rewrite rules need to deal with internet-style addresses that need to be sent on to another TCP/IP host via smtp. Those messages go into the smtp spool directory (mqueue) and not into an area.*

Wildcards by themselves allow flexible matches, but another tool makes the wildcard concept even more powerful. That tool is **variable substitution**. When an address matches a template containing wildcards, the characters matched by each wildcard can be represented in the rewritten address by variables. The first wildcard string is accessed with the variable "$1", the second with $2, and so on up to $9. Here's an example of how variable substitution works:

```
*@W8APR*   $1    r
```

This rule takes any mail that's addressed to W8APR and rewrites it to the user name alone, stripping off the host part. The "$1" represents the string matched by the first wildcard in the template, and forms the entire rewritten address. If we feed the address "KE8TQ@W8APR.#DAY.OH.USA.NOAM" through this rule, the $1 variable will contain the part of the address matched by the first wildcard – "KE8TQ", and that will be the rewritten address; the rest of the original address is discarded. The second wildcard is there to ensure that the rule ignores anything after the "W8APR" hostname – without it, "KE8TQ@W8APR" would match, but "KE8TQ@W8APR.#DAY.OH.USA.NOAM" wouldn't.

By the way, did you notice the "r" hanging out beyond the rewrite address of the rule? That's the key to REWRITE's third tool – recursion. Including an "r" at the end of a rule tells smtp to start over again at the top of the REWRITE file, using the rewritten address as its input. That's very useful if you need to "preprocess" an address.

The example we just worked through is a very important one. You need a rule like this at the top of your rewrite file to capture addresses that are local to your system (I'm in a creative mood, so I'll call this **localization**.) A localizing rule is necessary to reformat local addresses into a format that's valid for an area name. Without it, a message addressed to "KE8TQ@W8APR.#DAY.OH.USA.NOAM" that doesn't match any other rule will be put into an area file called "KE8TQ@W8APR.#DAY.OH.USA.NOAM.TXT", which isn't what anyone (particularly MS-DOS) wants![13-5]

[13-5] NOS versions that support the NNTP news transfer protocol handle this a bit differently. Instead of a long filename with lots of "." characters, they break the name into subdirectories. The example would result in this file: spool/mail/ke8tq@N8UR/#day/oh/usa/noam.txt. For normal mail, this still isn't the result you want!

Designing a REWRITE File

You've survived the mind-numbing theory of how the rewrite process works. Now, how do you use it in practice? I've included a complete example of a REWRITE file in Appendix H; you may be able to use it by changing our local rules to match yours.

Probably the most important idea to keep in mind when designing the REWRITE file is that the rewrite process is linear, and stops after the first match. This means that the order of rules is very important. Once an address matches a rule, that's the end of the story (unless you're using a recursive rule). Here's a gross example:

```
*          junk
*@tcpip*   tcpip
```

You're never going to get any messages in the tcpip area, because the first rule catches *everything*. This example is very obvious, but the problem can be very subtle as well. For example:

```
*@allus*   allusa
tcpip@*    tcpip
```

Here, the first rule will catch messages addressed to "tcpip@allusa" and put them in the allusa area, instead of the tcpip area that was intended. You can solve this problem by reversing the order of the rules.

Here's an order for the REWRITE file that I've found works well:

1. A recursive localization rule.
2. Capture internet-style addresses that should be sent on via smtp. If you have any sort of Internet email connectivity, you'll probably end up with a rule for each of the top-level internet domains, and perhaps several of the country domain designators as well.
3. Rules that deal with local names. This produces results similar to the alias file. You may want to have a rule that puts messages for "sysop" in your personal area. There are really two kinds of sysop messages: those addressed to "sysop" or sysop@<your station>, which are probably private messages you should pay attention to, while those

to “sysop@<flood designator>” which are really bulletins. You may want to put the sysop bulletins in a public or semi-public (using the AREAS.SYS file in TNOS) area.

4. Rules for “subject matter” bulletin types such as “arrl”, “amsat”, “packet”, “tcpip”, etc., that you want to assign individual areas. You’ll probably want two rules for each of these areas: one like “*@<area>*” and one in the form “<area>@*” to make sure you catch the inconsistent address schemes used in the PBBS world.

5. Rules for geographic flood designators like “allusa” and “ww”.

6. Forwarding rules. Depending on your forwarding partners, you may be able to use a few rules to send all outbound PBBS messages to a single BBS, or you may need a separate entry for each state, for each continent, and for the local distribution designators in your own and neighboring states. The area(s) into which you put messages for forwarding should be private and not included in the AREAS or AREAS.SYS files. We use the suffix of the forwarding partner’s call as the area name – for example, messages we forward to WA8ZWJ go into a private area called ZWJ. That way, Keith can still receive mail (and access his mailbox) as WA8ZWJ@W8APR if he needs to.

7. Finally, you need a “check” rule that catches anything that’s left for syop review. Something like:

```
*@*          check
```

will do that. This will put oddball messages into the check area, which the sysop should check regularly. The contents of this area will give you clues about changes you may need to make to the rewrite file, or education you may need to provide your users.

Moving Messages with FORWARD.BBS

PBBSs are built around the concept of mail forwarding—they use a protocol to automatically connect to other PBBSs and forward both private messages and bulletins. NOS uses the FORWARD.BBS file to define what PBBSs to forward to, how often to do it, and what messages should be forwarded.

The structure of FORWARD.BBS is straightforward. It consists of one set of entries for each PBBS to which the system forwards. The entries specify the callsign of the remote PBBS, how often to connect (and optionally the hours during which connects are permitted), any special commands necessary to make the connection, and, finally, a list of all the message areas that should be forwarded. So, carrying on the example I started above, the FORWARD.BBS entry at our system for WA8ZWJ might look like this:

```
# forward to WA8ZWJ only between 01 and 07 hours
WA8ZWJ 0107
# make the connection with this command
c uhf WA8ZWJ
# forward messages in the following areas
tcpip
packet
ohio
zwj
# end the wa8zwj forwarding info with a line of dashes
------------
```

The first line specifies the name of the remote PBBS, and the starting and ending hours that we will forward to it. The second line specifies the NOS command string that's used to actually establish the connection. The next lines, up to the dashes, are the areas that we will forward to this PBBS. A line containing several dashes indicates the end of the section for each forwarding partner.

This list includes several public areas, as well as the private "zwj" area described above. When this system connects to WA8ZWJ, these areas will be scanned and any messages not yet sent to WA8ZWJ will be forwarded to him. Once messages in a public area have been forwarded, NOS adds a header line to the message text to show that fact; messages in private areas like zwj are deleted after forwarding.

Recent JNOS and TNOS versions have added a lot of flexibility to FORWARD.BBS. It can now handle complex connect scripts that might be needed to establish a connection with a PBBS several network hops away, and you can control reverse forwarding and polling behavior for each PBBS in the list.

A more complete FORWARD.BBS file, though by no means the most complicated one possible, is listed in Appendix I.

Cleaning Up With the EXPIRE.DAT File

A busy PBBS may have several thousand active messages in its files. It's necessary to clean out the old messages every now and then to keep disk space under control. EXPIRE.DAT defines the maximum age (in days) that messages will be stored in each area. There are two ways to expire the messages: There's an external program called EXPIRY.EXE that will run outside of JNOS (but not TNOS), and there is an internal **expire** function that can be set to fire off at regular intervals. I recommend you use the internal expire command. Here is a sample AUTOEXEC.NOS section that you could use to set up internal expiration:

```
at 0700 "expire 24"
at 0701 "expire now"
at 0800 "oldbid 24 21"
```

The first two commands set the expire timer to kick every 24 hours and start the first expire process a minute later. The last command starts the bid expiration process an hour later, telling it to run every 24 hours, keeping old bids for 21 days. This trims the HISTORY file by deleting old entries; if you don't do this it can grow huge. A very large HISTORY file not only uses disk space but also slows down operation, since NOS must scan that file for every incoming forwarded message.

At W8APR we once found that hundreds of messages were queuing up in the mqueue file and never being delivered; the system was bogging down and finally crashing. We discovered that the expire process had inadvertently been turned off, the HISTORY file

was many megabytes in size, and the time to process each message was so long that the smtp process was timing out. Expiring out the old bids reduced the file back to a normal size, and everything worked fine after that.

Revisiting the FTPUSERS File

I spoke briefly about the FTPUSERS file earlier. Both JNOS and TNOS have greatly expanded the permissions available to users, and both use a common entry called univperm to set default permissions for mailbox users. The value of the univperm entry will vary depending on the configuration of your system; see the sample FTPUSERS file in Appendix G. To calculate the value for univperm, add up the values of the permissions you want to give your mailbox users.

If you don't define univperm in the FTPUSERS file, users will be prompted for a password when they connect, and you will have to specifically set up a line in FTPUSERS for each user. That's not very practical, since PBBS users tend to come and go.

You will also need to add a line in FTPUSERS for each remote BBS you forward to, or receive messages from. The premissions value may vary depending on what other NOS services the remote system uses, but _____ is a good starting point.

Other PBBS Configuration Files

As the PBBS functions in JNOS and TNOS have grown more complete, several additional files are now part of the picture. For example, each user on the PBBS can have a signature file that is automatically appended at the end of each message they send. TNOS adds to the AREAS file an AREAS.SYS file that allows additional control over areas. TNOS also has several other features, including custom command scripts and a help/tutorial system, that use configuration files. See the documentation for the programs to learn more about these features.

Chapter

14

An Introduction to Unix TCP/IP

Almost since the beginning, NOS running under MS-DOS has been starved for memory. Most of the stability problems that people have reported come back to the fact that NOS is a complex, sophisticated program—in many respects, it's a complete operating system—that strains DOS to its absolute limits. You can't configure a DOS-based NOS system to be a BBS, Convers bridge, and switch all at once, and expect it to stay up for weeks on end. It can do any one of these things very well, but you can't put 10 pounds of potatoes in a five pound bag.

One answer to the problem is to move NOS to another operating system that can better handle its demands. For years there have been versions of NOS that run on UNIX operating systems, but they've never been too popular since UNIX itself was unwieldy, expensive, and required more computer resources than most hams had available. That's changed in the last couple of years, though, as two "freeware" versions of UNIX have become available. These are **Linux** and **FreeBSD**. Both are fully functional operating systems that put some of their commercial counterparts to shame. Both come with compete source code, and both will run on computers that many hams have available today—a 386 or better with at least 4MB of RAM.

The rest of this chapter focuses on Linux because there's been more ham-related development for it than there has been for FreeBSD, but the chances are things will even out in the future. In particular, TNOS now runs under Linux, FreeBSD, and SunOS.

Linux currently offers two ways to get on the air with TCP/IP. Like most UNIX implementations, it includes TCP/IP networking as part of its basic functionality; a Linux system includes telnet, ftp, smtp, and other servers and clients. However, unlike any other operating system in the world, Linux also supports the AX.25 protocol – and therefore amateur TCP/IP – as a built-in feature. Alan Cox, GW4PTS, Terry Dawson, VK2KTJ, Jonathan Naylor, G4KLX, and several others have developed drivers and utilities for Linux that support AX.25 and allow you to put these services directly on the air. If you want to put a system on the air that looks as much as possible like a "real" network server, this may be the way to go.

The other approach is to use a version of NOS that's been ported to Unix. Both TNOS and JNOS currently have Linux versions. One of these systems running on top of Linux or FreeBSD provides all the features of NOS with stability that's at least an order of magnitude greater than you'll ever see under DOS. As I write this, the W8APR system, which runs TNOS/Linux and serves as a full function PBBS, has been up for over 23 days and has processed over 17,000 messages since its last reboot. This may not be unusual for some of the other PBBS programs, but for a NOS system under DOS it's just about unheard of.

The next few chapters describe how Linux can be used as the base for powerful amateur radio TCP/IP systems. I want to provide you with enough information to get your hands dirty, and that means I'll unavoidably be "speaking Unix", and specifically the Linux dialect. If you're unfamiliar with Unix, you will probably find a lot of the following pages pretty strange; the next chapter provides a very brief Unix tutorial to help you through the gobbledygook. If you're interested in learning how to actually use Linux, there are now quite a few books available that will help you get started. One very good one is Running Linux by Matt Welsh and Lar Kaufman, published by O'Reilly (ISBN 1-56592-100-3).

Chapter

15

A Linux Survival Guide

There are a few common rules that govern all Unix systems. Keeping these in mind will make Unix seem a bit less foreign to you.

Unix includes lots of documentation online; how useful it is, is a matter of debate. You can find the manual page for standard commands by typing **man <commandname>** at the prompt. Linux systems also include a documentation system called **info** and it provides more detailed documentation for some programs. To use it, type **info <commandname>**. Finally, many Linux distributions include a directory with documentation for the various programs they provide. On a Debian Linux system, for example, you can look under /usr/doc to find lots of "readme" files and other useful stuff.

Unix is a multi-user system, and you have to create a user account with a login name and (if you're smart) password to get into the system. There's a special user, called "root", who is all-powerful and can do anything on the system. You'll need to be logged in a root to do many of the activities described here; when you are, remember that you have the power to wipe out the system if you're not careful. The best rule is only to login as root when you really need to.

Unix has no "undelete" command. When you delete a file or a directory it's gone forever. "rm –rf *" (described below) are the most dangerous characters you can type on a Unix system; will wipe out everything in the current directory and every subdirectory below it – and if you issue that command from the root directory, you'll be reinstalling the system from scracth. Think twice and type once.

Unix uses the forward slash ("/"), instead of MS-DOS' backslash ("\"), as a directory separator. Unix uses the backslash character as an escape character.

Unix is case-sensitive. The file "autoexec.nos" is different then "AUTOEXEC.NOS" or "AutoExec.nos". Nearly all Unix command names, and most file names, are lower case; mixed case is sometimes used to make long names easier to parse (like "TheLatestVersion.txt"). All upper case either means you're shouting, or the file is imported from an MS-DOS machine.

Unix systems support long file names, so you may see some real monsters, particularly in software packages that include lots of version information. For example, this is the full name of the current version of Netscape Communicator for Linux: communicator-v45-export.x86-unknown-linux2.0.tar.gz File names can include spaces and other odd characters. To access a file with spaces in its name, put the name in quotes, or add a backslash ("\") before each space.

Although Unix supports long file names, that doesn't mean it *uses* them. Many Unix command names are short and cryptic. The most important are **ls** (list directory – by itself, ls provides a short directory list; to see details like file modification date and size, type "ls –l"); **cp** (copy file); **mv** (move or rename file); and **rm** (delete file). Most systems have a program call **less** or **more** (less is derived from more; Unix programmers think they have a sense of humor) that's used to display text files a pageful at a time. To copy two or more files into a single new file, the **cat** command works:

```
cat file1 file 2 > newfile
```

Unix commands are stored in several directories, which may include /bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin, and /usr/local/sbin. Your default path may not include all these directories. To see what your path is, type "**env**" at the prompt. If the file you need is not in one of the directories in your path, you can still run it by using the full pathname. There's a useful tool on Linux systems to help find commands. Type "**where <filename>**" to see all the directories where the file is located. "**whereis**" searches the most common directories, but not the entire system. To search for a file in every nook and cranny, use the "**find**" command, like this:

```
fine / -name <filename> -print
```

<filename> can include wildcards. This search is exhaustive and may take some time.

The most common Unix text editors are called **vi** and **emacs**. vi is small, efficient, and difficult to use, while emacs is large, bloated, and difficult to use. That's not really fair to either program; they both work well once you know how to use them, but unfortunately, they both work differently from each other, and from any MS-DOS-based editor you've likely used.

Unix software packages are usually packaged using a program called **tar**, which collects a bunch of programs into a single archive. A program called **gzip** is used to compress tar achives; the end result is roughly equivalent to a DOS "zip" archive. tar achives have a default extension of ".tar" while gzipped files usually end with ".gz"; tarred archives that have been gzipped often have a combined ".tgz" extension instead of something like ".tar.gz" (though you may see that, too). To extract a tar file, you can use the command **tar xvf <filename>**. To extract a gzipped tar file, add a "z" to the tar options: **tar zxvf <filename>**. Unlike most Unix commands, there's no dash ("-") before the option letters in tar. By the way, "gz" files aren't compatible with DOS ".zip" files. To create .zip files, you can use the zip and unzip programs included with most Linux distributions. Each of the Linux distributions has it own, more sophisticated packaging systems. Debian Linux packages end with the suffix ".deb", while RedHat Linux uses ".rpm". Each distribution includes its own package management program to work with these files.

Unix "programs" may be compiled binary files, like MS-DOS ".exe" files, but they often are scripts that are run either by the shell (the shell is a program that provides the command line interface to a user, much like COMMAND.COM in MS-DOS, and shell scripts are much like DOS batch files) or an interpreter like **perl**. Unlike DOS, where files you can run from the command line must end with ".COM", ".EXE", or ".BAT", most Unix programs don't have an extension.

Some Unix command lines can be very long and may not fit on a single line without wrapping. To make it easier to read such commands, you can break them into multiple lines by using the backslash ("\") character at the end of each line except the last. You can also use tabs or spaces to indent the subsequent lines.[15-1] Here's a single command split into multiple lines for readability:

```
/sbin/route add –net 44.71.36.0 \
    netmask 44.71.36.255 \
    mss 236 \
    window 512 \
    irtt 3000 \
    ax0
```

Unix text files use the linefeed character at the end of each line, unlike MS-DOS, which uses the carriage return/linefeed combination. Text files created under MS-DOS will display in Unix as if they're double-spaced, or with "^m" characters appearing at the end of each line. Linux includes programs called **todos** and **fromdos** that will convert files between the two end-of-line conventions.

If you need to read or write to an MS-DOS floppy disk, Linux includes a set of programs that allow this. They include **mcopy**, **mdel**, and **mmove**.

[15-1] Although Linux shell scripts and configuration files nearly always accept multiple backslashed lines, tabs and spaces, and blank lines, there are some exceptions, so if you have unexpected errors try removing the formatting.

Chapter

16

NOS and Unix

Why are JNOS or TNOS under Linux so much more stable than the same program running under DOS? It's very simple: Linux doesn't have the 640k memory barrier imposed by MS-DOS. NOS can use as much memory as it needs (within the memory available on the system, of course), and this solves all sorts of problems. Also, since Linux is a multitasking operating system that's designed to do more than one thing at once, it offers a much more hospitable environment for NOS, which itself is frequently doing several things at the same time. It also offers better disk performance through its disk buffering system.

OK, I admit it. The preceding paragraphs have been a blatant plug for the Linux approach to NOS. But switching the W8APR BBS from JNOS under DOS to TNOS under Linux has made our life much easier; system restarts now occur quarterly rather than daily. This chapter focuses on TNOS since that's what I'm most familiar with from its use at W8APR; JNOS configuration is quite similar.

If you're willing to climb the Unix learning curve, you can have the best of both worlds. You can run NOS/Linux as the "radio front end" of a TCP/IP system where it handles the radios and routes IP packets to the Linux networking code. This allows a blend of

commercial strength IP services through Linux and full AX.25 services through the NOS mailbox to lure folks in. In the past, to accomplish this requried a dedicated NOS/DOS box to serve as the front end, connected via ethernet or serial cable to a separate Unix system. NOS/Linux lets both pieces reside in the same machine. The rest of this chapter describes how to implement this model.

Configuring TNOS under Linux

TNOS/Linux[16-1] is available from several sources; the official site is ftp.lantz.com. The package is available in both a compiled version, and as source code. Unlike DOS versions, the precompiled TNOS/Linux binary has nearly all features compiled in – without a 640k memory limitation, there's no need to scrimp on functionality. This means you don't need to worry about compiling a customized version.

Installing the TNOS Package

When you grab TNOS/Linux from a BBS or ftp site, it will be in a compressed "tar" file with a name like "Tnos-2.30.tar.gz" or "TNOS230.tgz". Login to your Linux system as the root user, and create a directory to hold TNOS (the recommended directory name is /nos and I'll assume that's what you're using). Change directory to /nos and issue the command

```
tar zxvf /tmp/Tnos-2.30.tar.gz
```

(substituting the full directory and filename of your TNOS tar file). You'll see a progress report on the screen as files are extracted and installed.

This process creates the directories that TNOS needs, and installs the programs and sample versions of the configuration files.

[16-1] The correct name is actually TNOS/Unix, because TNOS can now be compiled and run under several Unix-like operating systems, including SunOS, FreeBSD, and Linux. I'll call it TNOS/Linux here because I'm talking about the Linux version, and specifically the pre-compiled Linux binary that's available.

Configuration Files

Once you've installed TNOS, you should run the configuration program **mksetup.tcl** which will create a customized autoexec.nos file based on your answers to a series of questions. Once you've done that, you can fire up the program with the **startnos** command to see if everything's working. You're not quite finished, though.

You'll need to manually edit the other TNOS configuration files like **alias**, **areas**, **ftpusers**, **rewrite**, and **forward** to match your requirements – if you don't, you'll be running sample config files, and things won't work properly. These files are in the same format as their DOS counterparts, so the previous chapters on setting up DOS-based TNOS systems will show you what to do. You may also want to manually edit the **autoexec.nos** file that mksetup.tcl created for you; the setup tool does a good job, but there are a number of things you may want to tweak.

Starting (and Restarting) TNOS Automatically

You can configure Linux to start TNOS automatically when the computer comes up, and to restart it automatically if for some reason it exits. You can also tell TNOS to use a specific **virtual terminal** (*i.e.*, have it start in one of the other login windows that Linux supports from the console). Virtual terminals are named /dev/ttyX, where X is usually from 1 through 8, and you can switch to them from the local monitor and keyboard by pressing function keys F1 through F8).

The first step is to write a startup script that starts TNOS using a specific virtual terminal:

```
#!/bin/bash
# /nos/nos-start
# See if TNOS is already running.  If it is, stop it.
# Then clean up the lock files and restart, redirecting
# input and output to the device specified on the
# command line.  For example, "/nos/startnos /dev/tty5"
# will start TNOS running on vt5; to get to it, hit
# the F5 key from the local console.
#

# kill running process
/bin/ps -aux | /usr/bin/grep " ./tnos" |\
while read user pid rest
do
    /bin/kill -STOP ${pid} 2>/dev/null
    /usr/bin/sleep 2
    /usr/kill -KILL ${pid} 2>/dev/null
done

# remove lock files
/bin/rm /nos/spool/mqueue/*.lck 2>/dev/null
/bin/rm /nos/spool/mail/*.lck 2>/dev/null

# get ready to start
cd /nos

# set local timezone; change as appropriate
export TZ=EST5

# set the terminal type
export TERM=console

# pause to let everything catch up
/usr/bin/sleep 1

# now start the program
exec ./tnos < $1 > $1 2>&1

# Done.
```

This is a complicated script, but it provides clean startup and shutdown; thanks to Mike Dent, G6PHF, for providing it.

After you create this file, make sure root owns it by issuing the command **chown root /nos/nos-start**, and make it executable with the command **chmod u+x /nos/nos-start**. Now, when you run this script from the command line as **/nos/nos-start/dev/tty5** TNOS will start on that virtual terminal; to shift to tty5 and see what's going on, press F5.

The second step is to run this script as part of the Linux initialization process, so that TNOS will automatically fire up when you boot the system. There's a file called **/etc/inittab** that controls how Linux starts. It runs the processes that generate login prompts, as well as triggering the preliminary activities needed to get all the system's resources running. You can include a line in this file to run TNOS as part of the startup sequence, and to restart (the technical term is "respawn") TNOS if it should die. /etc/inittab has a series of lines that define each of the virtual terminals. The line for tty5 might look like this:

```
c5:23456:respawn:/sbin/agetty 38400 tty5
```

That line causes Linux to put a login prompt on tty5. We'll modify it to start TNOS instead by changing it to this:

```
c5:23456:respawn:/nos/nos-start /dev/tty5
```

Now, TNOS will run on tty5, and if it should die, the init program that reads the /etc/inittab file will automatically restart it (that's what the "respawn" keyword in the third field signifies). To make this change take effect without rebooting the computer, type **telinit q** at the command line. After a few seconds, press F5 (or whatever function key corresponds to the virtual terminal you've chosen) and you should see the TNOS startup screen. If you type "exit" at the TNOS command prompt, you should see it exit and start again.

If TNOS doesn't start, keep cycling through the startup screen, or if you see an error from inittan about "respawninf too fast", something is wrong. The most likely culpritys are file permission problems, or incorrect pathnames.

Connecting TNOS to the Linux TCP/IP Subsystem

I mentioned above that Linux includes its own complete TCP/IP implementation. It's possible to run TNOS under Linux as a front-end that routes packets from the radio world to and from Linux' TCP/IP services. It's pretty easy to do, using a UNIX concept called a **pseudo-tty**. A pseudo-tty (abbreviated as "**pty**") lets two programs communicate with each other as if each controlled a serial port, and the serial ports were cabled together – it creates a logical pipe between them. Since a pty looks like a serial port, we can build a slip[16-2] link over it, and we can route data between TNOS and the Linux kernel's TCP/IP over that link.

Using a pty is much like hooking two programs together via a pair of physical serial ports – that's what the pty system devices look and act like to the programs using them. ptys come in pairs. One is called the **master** and its name will be something like **/dev/ptyp1**. The other is the **slave** and its name will be something like **/dev/ttyp1**. The master and slave pair share the same last two characters in their names. In a typical Linux system, something between 16 and 64 pty pairs are defined, with the first set of 16 identified as p0 through pf, the second as q0 through qf, etc. (the last character is hexadecimal). The practical significance of the master/slave designation is that the device using the master (ptyp) device must start first, and only then can the slave end start using a ttyp device.

What IP addresses should the pty link use? Should both ends use addresses in the 44.x.x.x amprnet, or in some other network, or should the networks on each end be different? Some may disagree, but I think the Linux end of the link should use a 44.x.x.x address in the subnet that TNOS is using for its RF ports, and the TNOS end of the link should use an address in the subnet that Linux uses for its network connection. If the Linux system doesn't have an Internet connection, the address used on the TNOS end doesn't really matter, and you can use any non-routed address you'd like,[16-3] though you should still use a 44.x.x.x address on the Linux end.

[16-2] ***slip*** *is the Serial Line Internet Protocol, described in Chapter 6.*

[16-3] *There are IP networks set aside for "private network" use that you can use for this purpose. In amprnet, 44.128.0.0 is reserved for testing and private use, while addresses in the 192.168.0.0 range are used for private networks in the Internet world. If you use an address in these networks, you* ***must*** *make sure that it is not routed to the outside world.*

Setting the addresses up this way makes routing simpler and the slip link transparent because each side of the link appears to be local to hosts on the other side (for example, radio users don't need to deal with routing to a non-amprnet address). The Linux machine has an amprnet address for users who want to reach it via the TNOS front end, and TNOS has an Internet address so it is reachable from the outside world through its Linux host. Using addresses in the same subnet on both ends of the link makes routing more difficult in one direction or the other.

With appropriate routing statements at each end, and the proxy arp function discussed earlier, users can connect use TNOS as a gateway to reach the Linux networking services, and vice versa, without even knowing they're going through a TNOS front-end. Here's how to configure both sides of your system to set TNOS up as a "front end processor" for the Linux networking system.

Configuring the Linux Side

The Linux configuration file you need to modify will vary depending on your distribution – you should try to configure the TNOS slip link in the file that contains the other networking setup commands; unfortunately, various distributions set up networking in very different ways. Appropriate places might be **/etc/rc.d/rc.inet1** in Slackware, **/etc/rc.d/rc.local** in Red Hat, or **/etc/init.d/network** in Debian.

The commands you enter will attach a slip network device to the master of a pty pair, and then add appropriate routing commands. Here's an example:

```
# Attach the first slip device to pty master ptyp9.
# We start with a high number because telnet and other
# services may use the lower pty devices.  The speed
# setting is a placeholder.  The "&" tells Linux to
# run the command and then return – without it the
# system will hang!  Assuming this is the first slip
# link established, ptyp9 will bind to "sl0".

/sbin/slattach -p slip -s 38400 /dev/ptyp9 &

# Delay for a second to let things get set up.

/usr/bin/sleep 1

# Configure the interface.  The netmask, etc., are set
# here to route one class C subnet via the slip link,
# assuming there's a default route that reaches every-
# thing else.  The pointopoint address is the address
# that TNOS will assign to its end of the slip link.
# The MTU should match what you're using on the
# radio channels.

/sbin/ifconfig sl0 44.71.36.2 \
    netmask 255.255.255.0 \
    broadcast 44.71.36.255 \
    pointopoint 192.168.1.3 \
    mtu 256

# Set the route.  On a slip link, the gateway at the
# other end is implied and it doesn't need to be
# expressly set in this command.

/sbin/route add -net 44.71.36.0 sl0
```

If you have a home ethernet, or Internet access to your Linux machine, you need to ensure that the Linux kernel on the TNOS machine is compiled with IP forwarding enabled. You do this by answering yes to the appropriate question in the kernel configuration script.

Proxy Arp Revisited

You also need to ensure that each machine on the local network, and your Internet gateway if you have one, know that they need to route through the Linux machine to get to TNOS (in effect, Linux acts as a gateway to reach TNOS). That usually means adding a routing statement on every system. But there's an easier way that makes the slip link invisible to other users and makes the TNOS system look as though it is sitting directly on the ethernet. You'll remember that earlier we talked about "proxy arp" and its ability to let one system pretend to be another. Now it's time to use it.

Let's assume for a minute that the local network is 192.168.1.0/24. Our gateway to the Internet is 192.168.1.1. The Linux machine that hosts TNOS – call it "tnos-host" – is 192.168.1.2, and the TNOS end of the pty slip link is 192.168.1.3.

If the gateway machine wants to communicate with TNOS, it first sends an arp request to 192.168.1.3. Since arp requests only travel on the local network, and don't go through routers, TNOS won't hear the request and won't respond. This is a problem, and it's why we need to have explicit routes through gateways.

But tnos-host knows that it can route packets to TNOS (it has the right routing command in place). By using proxy arp, it will respond to the Internet gateway's arp request and pretend to be 192.168.1.3. The gateway believes that response and sends the packet to tnos-host, which then forwards it via the pty slip link to TNOS. The packet gets where it's supposed to be, and no one's the wiser. The value of proxy arp is that it eliminates the need for additional routing statements to handle one-off situations such as our hidden TNOS problem. Figure 16-1 shows how this works.

It's easier to implement proxy arp than to describe it. You need to know two pieces of information – the IP address of the system that you are impersonating, and the hardware address of the interface on which you're going to do the impersonation. The IP address is the address assigned to the TNOS end of the slip link. The hardware address is probably the address of your ethernet card.

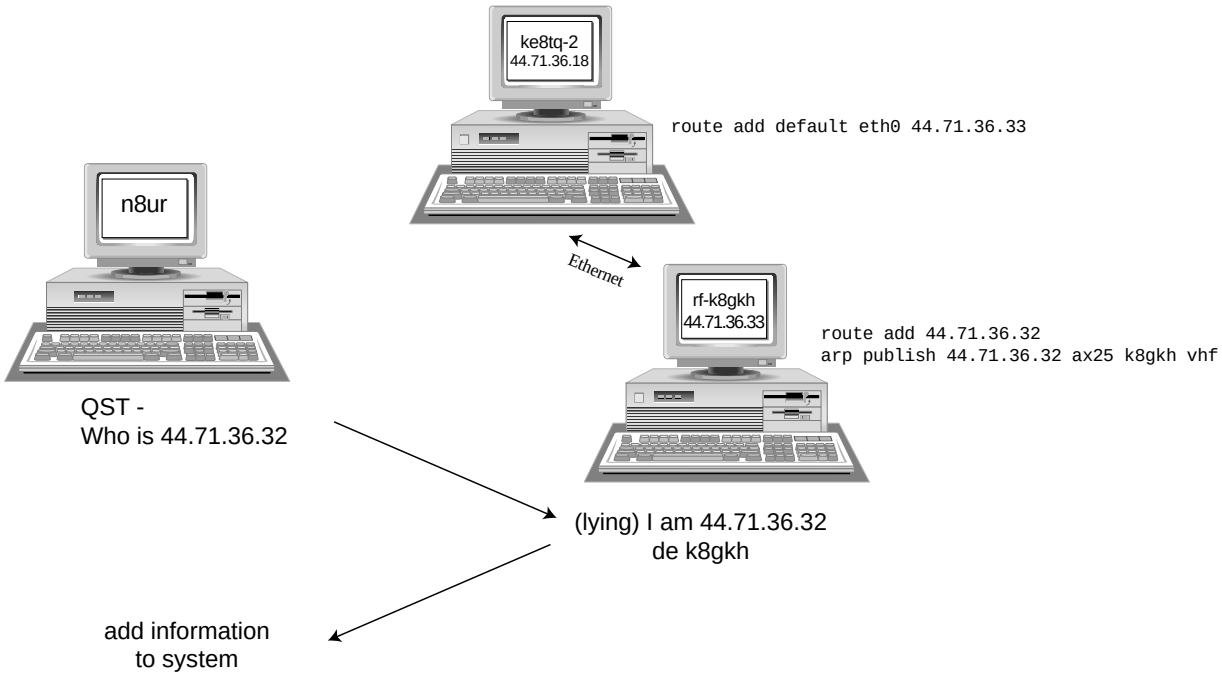


Figure 16-1 - Proxy Arp

To find the hardware address, issue the **ifconfig** command and look at the "Hwaddr" field relating to your network card (probably interface "eth0"). If it's an ethernet card (a likely bet on the Linux side), the address will be six two-digit hexadecimal numbers separated by colons. Add this command at the end of the pty slip link configuration (after the /sbin/route/statement), using the address of the TNOS end of the slip link, and the hardware address:

```
/sbin/arp -s 192.168.1.3 00:20:18:10:8e:70 pub
```

Remember that if you change network cards in the computer, you'll have to find the new hardware address and change the command to match it.

By running these commands as part of your network initialization at boot time, the master end of the pty slip link will be ready and waiting when you bring up TNOS.

Configuring TNOS

To set up the TNOS end of the slip link, you need to include commands like these in AUTOEXEC.NOS (probably located near your other attach and routing commands):

```
# Attach the interface to the Unix kernel.  Use
# the slave end of the pty pair set up in the
# Linux network configuration
attach asy ttyp9 - slip sl0 2128 1064 38400 c

# These commands set buffer sizes.  Adjusting them
# may improve performance on your system; see the TNOS
# docs for more details.
asyconfig sl0 rxqueue 30
asyconfig sl0 txqueue 10

# The following ifconfig commands set the
# interface parameters.
ifconfig sl0 description “Slip Link to Linux”
ifconfig sl0 ipaddress 192.168.1.3
ifconfig sl0 broadcast 192.168.1.255
ifconfig sl0 netmask 255.255.255.0
ifconfig sl0 tcp irtt 100

# The following is in addition to your other
# routing commands.  It is the route to the Linux
# end of the slip link.
route add [44.71.36.2] sl0
```

With a couple of minor differences, these commands are the inverse of the ones we use to set up the Linux side.

It's also possible to use proxy arp in TNOS to make the Linux system available on the air as if it had radios directly connected:

```
# This command uses proxy arp to make TNOS respond
# to arp requests for the Linux address.  See the
# discussion of proxy arp earlier in the book for
# more details.
arp publish [44.71.36.2] ax25 W8APR vhf
arp publish [44.71.36.2] ax25 W8APR uhf
```

TNOS requires an "arp publish" command for each interface. The hardware address is the TNOS callsign.

Summary

By setting up a slip link using a pseudo-tty pair, we can link TNOS to its Linux host and allow data to flow between them as if they were two independent systems. Enabling proxy arp on the Linux side makes TNOS visible to the Linux network (which may be another machine linked via ethernet, or a gateway to the Internet). Enabling proxy arp on the TNOS side makes all the Linux TCP/IP services available to the RF network via TNOS. Using "opposite" addresses on the slip link (amprnet on the Linux side and Internet on the TNOS side) simplifies routing and makes the configuration transparent to users.

Chapter

17

Using Linux Native AX.25

The previous chapter described how to use TNOS as a "front end" to Linux' native TCP/IP system. Thanks to work by Alan Cox, GW4PTS, Jonathon Naylor, G4KLX, Terry Dawson, VK2KTJ, and others, Linux also has built-in support for AX.25. This means it can directly control KISS TNCs and other AX.25 devices, and you can put a Linux system directly on the air for AX.25, NET/ROM, ROSE, and TCP/IP without running NOS. Until recently, this feature was best considered as experimental code, but with the release of version 2.0 of the Linux operating system kernel the AX.25 support has reached a level of real usefulness.

Linux is the first operating system in the world to have built in support for the AX.25 protocol, and that opens up a world of applications for the Linux packet radio user. What does that mean, and how can we use it?

Like other UNIX systems, Linux has TCP/IP networking built in and very tightly integrated with the rest of the system. Unlike MS-DOS, networking is not an "add on" to Linux. However, most UNIX systems are designed around Ethernet or other high-speed network hardware (though most have support for serial-port networking as well). What Linux adds is the ability to communicate using AX.25 as well as Ethernet. You can hook a KISS TNC to the serial port on a suitably-configured Linux system and

provide access via "straight" AX.25 as well as TCP/IP and NET/ROM. Linux includes not only the basic hardware support for AX.25, but also the ability to adjust parameters to make the networking code work efficiently on slow radio channels.

Linux' AX.25 support is very flexible. A user connecting via AX.25 or NET/ROM can be given a login prompt that allows direct access to any program on the computer as if they were logged in at a local console, sent to a program (like a BBS package), or receive a NET/ROM-like node prompt. A user connecting via AX.25 TCP/IP can do anything they could do via an Ethernet connection: remote login, file transfer, email transfer, etc.

The really exciting thing about Linux AX.25 is that it provides a platform that makes it very easy to develop applications for users. It's easy, using one of several programming languages and tools, to set up all sorts of useful programs that can serve your local network. Unlike JNOS or TNOS, a Linux application doesn't need to recreate the complete hardware interface, AX.25 layer, IP layer, and TCP layer – the programmer can concentrate on the application code. Using tools like perl, you can write a TCP server in just a few lines of code. A Linux system also has the potential of making a very good router that can handle NET/ROM and TCP/IP switching duties with good performance and high reliability.

Configuring Linux AX.25

Adding AX.25 capability to a Linux system requires compiling a Linux kernel that includes AX.25 support, obtaining and compiling the utility and application programs that interface with the kernel, and setting up a number of configuration files. I'll describe the process here, but before you try it yourself, you should look at the AX25-HOWTO document that's available by ftp or http from a number of sites, including http://sunsite.unc.edu/mdw/HOWTO/AX25-HOWTO.html. The AX.25 programs are still maturing, and the HOWTO is the best guide to obtaining and installing the most recent version.

A Note About Versions

Linux continues to evolve, and as it does the AX.25 support evolves with it. As I write this in early 1999, there have been two recent changes to Linux that will impact your installation.

First, the AX.25 support built in to the kernel has been updated. Users of kernel version 2.0.35, 2.0.36, or any kernel greater than 2.2.0 will have the new support. If you're using an older kernel, follow the steps in the section entitled "Applying AX.25 Updates to the Kernel Source." Even if you are using a new kernel, you will need to configure it to turn on AX.25 support.

The second change is an update to the standard Linux library – a file called **libc** that is used in compiling programs. During 1998 Linux distributions such as Red Hat and Debian started to switch from version 5 of libc to version 6. The change requires that some programs be modified before they can be compiled. If you are using a "libc6" system, you need to make sure that you have a version of the AX.25 programs that supports the new library. The correct program versions are available from TAPR and other sources.

Configuring and Building the Kernel

Most of the pre-compiled kernels included with Linux distributions do not have AX.25 turned on, so you'll need to compile your own kernel. Don't worry – it's not difficult to do. First, you'll need to make sure the "gcc" compiler and its associated tools are installed on your system. If you find a file called "/usr/bin/gcc" you're in business. Next, get a copy of the kernel source code file. The latest version is available via ftp from a number of sites on the Internet, including ftp://sunsite.unc.edu in the /pub/Linux/kernel directory. The kernel file is several MB in size these days, and can't be put onto a single floppy; keep that in mind if you need to transfer it from another machine.

Log in to your system as root and unpack the source code archive into the /usr/src/ directory using commands like this:

```
cd /usr/src
tar zxvf linux-2.2.0.tar.gz
cd /usr/src/linux
```

Next, you need to install links for some "include" files (these steps are included in the "README" file in the kernel source directory; you should always read that file for any changes that may have occurred):

```
cd /usr/include
rm -rf linux
rm -rf asm
ln -s /usr/src/linux/include/linux linux
ln -s /usr/src/linux/include/asm asm
ln -s /usr/src/linux/include/scsi scsi
cd /usr/src/linux
```

Finally, you need to create a configuration file that specifies the hardware and software features you want to include in the kernel. Do this is by entering the command "make config" or, for a slower but more user-friendly approach, "make menuconfig". Either way, you will be guided through a series of questions that require yes or no answers. The default answer will be capitalized, and will be used if you hit the return key in response to the question.

There may be all sorts of things you want to turn on and off, but the critical ones for AX.25 are:

```
Networking support (CONFIG_NET)
Amateur Radio AX.25 Level 2 (CONFIG_AX25)        YES
# The following provides G8BPQ-over-ethernet capability
AX.25 over Ethernet (CONFIG_BPQETHER)            OPTIONAL
# The following enables the NET/ROM protocol
Amateur Radio NET/ROM (CONFIG_NET/ROM)           OPTIONAL
```

Applying AX.25 Updates to the Kernel Source

There has been built-in support for AX.25 available in the Linux kernel for a long time, but during 1997 and most of 1998 the AX.25 development effort resulted in changes that greatly improved the quality of the support. As noted above, that code made it into kernel version 2.0.35 and later. Although the AX.25 code included in the kernel distribution for versions 2.0.34 and earlier works, it is not nearly as solid as the later code. So, if you're wise, you'll make sure that you're using a kernel that includes the latest version of the AX.25 code.

What that means is that if you're using a kernel version from 2.0.0 through 2.0.34, you need to follow these steps; if you're on version 2.0.35 or later (including any of the 2.2.x series [17-1]), you can skip them. If you're using a kernel version lower than 2.0.0, you need to update your system!

After installing the Linux kernel sources but before running the configuration program, you'll need to obtain the ax25-module patch file from TAPR, zone.psft.fi/~jsn, or some other source. The current, and probably last, version is **ax25-module-14f.tar.gz**. Extract the patch file into the Linux kernel source directory (usually **/usr/src/linux**): **tar zxvf ax25-module-14f.tar.gz**. You'll end up with a "readme" file or two, and another file with a ridiculously long name, like "ax25-2.0.31-2.1.47-3.diff" which is a set of changes that you'll apply to the kernel source code tree with the **patch** command: **patch –p0 ax25-2.0.31-2.1.47-3.diff 2> patch.error** (if you get an error, or **patch** stops to ask you for a file name, you'll need to change the –p parameter to another value, usually 1). Any error messages will be sent to the **patch.error** file. You should scan through that file to see if there are any messages indicating failed or rejected patches; if there aren't any, the patch was applied successfully and you're ready to proceed.

In addition to KISS over a standard serial port (which will be automatically enabled), Linux supports several packet radio interface cards. In the "Drivers" section of the configuration program, you'll be allowed to include the following drivers if you answer "yes" to the question "Radio network interfaces (CONFIG_NET_RADIO)" (note that you may need to request "experimental" drivers to see all of these):

```
BAYCOM ser12 and par96 kiss emulation driver for AX.25 (CONFIG_BAYCOM)
Gracilis PackeTwin support (CONFIG_PT)
Ottawa PI and PI/2 support (CONFIG_PI)
Z8530 SCC kiss emulation driver for AX.25 (CONFIG_SCC)
```

[17-1] *Kernels in the 2.1.x and 2.3.x series are development kernels, and I don't recommend that you use them in a production system.*

The config program allows you to define many of the options as modules. Doing so results in the driver being compiled as a separate chunk of code that's not included in the main kernel, but which can be installed when needed either manually, with a program called **modprobe**, or automatically if you have configured your system to use a daemon called **kerneld**. If you're using modules, you need to carefully read the file to learn the necessary tricks (**/usr/src/linux/Documentation/modules.txt)**.

Although modules are a really neat idea, and work very well in systems where the options in use change frequently, their advantages aren't as great if the drivers are in use all the time, which is likely if you're using AX.25 services. I generally compile the AX.25 options as kernel built-ins, rather than as modules.

Once you've finished the configuration script, type the following commands to compile the kernel:

```
make dep
make clean
make zlilo[17-2]
```

If you've chosen to build any options as modules, you'll also need to issue these two commands after the compilation process is complete:

```
make modules
make modules_install
```

[17-2] The "make zlilo" command may not work with older kernels. If it doesn't, you can use "make zImage" ***instead and then issue the following set of commands when compilation is complete:***

```
cd /
mv vmlinuz vmlinuz.old
cp /usr/src/linux/arch/i386/boot/zImage /vmlinuz
/sbin/lilo
/usr/sbin/rdev -R /vmlinuz 1
```

Go have a cup of coffee (or several), because the compile process could take from fifteen minutes to a couple of hours, depending on how fast your system is. After you reboot the computer with the "/usr/sbin/shutdown -r now" command, you should be running the new kernel with AX.25 support.

Installing the AX.25 utilities and user programs

In addition to including AX.25 support in the kernel, there are several utility and application programs necessary to make everything work. If you have a recent Linux distribution (particularly one that includes a version 2.0.x or 2.2.x kernel), versions of the standard networking utility programs that support AX.25 are probably already installed. If you don't have them, you'll need to get the "net-tools" package and install it (since most recent systems include versions of these programs, I won't go into the installation instructions here).

You'll also need the AX.25 user programs, which again may be included in recent Linux distributions, or are available on the Internet (from http://zone.pspt.fi/~jsn, although they are also available from the TAPR site). As I write this, the current version is "ax25-utils-2.1.42a-tar.gz", which will work with kernel versions 2.0.35, 2.0.36, or 2.2.x, or with versions 2.0.33 or 2.0.34 that have been patched as desribed above.

There's also a version of the ax25-utils package with "2.0." in its name instead of "2.1." That version is for use with unpatched kernels that still use the old AX.25 code, and *should not* be used if you are using a newer kernel, or have followed the directions I gave above to update the kernel AX.25 code.

Grab the appropriate ax25-utils archive file and do the following as root (after installing the ax25-module kernel patches, if your kernel version requires them):

```
cd /usr/src
tar zxvf ax25-utils-2.1.42a.tar.gz
cd ax25-utils-2.1.42a
make config
make install
```

The "make config" command will configure the programs for your system, and "make install" will compile and install the programs. It will also install a set of man pages (Unix documentation) for just about every ax25 program and file. You can view these with the **man** command.

Configuring Linux AX.25

Now that you've installed the software, you need to set up several configuration files that define your AX.25 setup, and probably issue some commands to configure AX.25 parameters. All the configuration files are located in **/etc/ax25** (note that in earlier versions they were put in /usr/local/etc; this changed with the 2.0.12 release and matching utilities), and the configuration commands are all in **/usr/sbin**.[17-3] By the way, since many Linux/AX.25 users will be running applications such as the F6FBB PBBS software on their systems, and the configuration process touches on – and is affected by – some of these additional services, in the sections that follow I'll describe overall AX.25 configuration, and not limit myself strictly to TCP/IP issues.

In Linux systems, network devices like ethernet cards or TNCs must have an **interface name** that uniquely identifies that device. Some interface names, like those for network cards, are created automatically by the kernel. If you have an Ethernet card in your system, you have an interface named **eth0** that comes into being when the driver for the card is loaded.

[17-3] *In this chapter, I'll usually refer to programs by their full pathname. If your system has a PATH variable that's fairly normal, you may be able to type just the command name, without the path. It's always a good idea, though, to use the full pathname in any script, because you can't guarantee what the PATH variable will say when the script is executed.*

On the other hand, the kernel can't create some types of network interfaces automatically, because the hardware may be used for different purposes. For example, a serial port may be used to talk to a KISS TNC, instead of the more traditional use with a modem or a terminal, or for a slip or ppp link. Special programs are used to turn such hardware into network interfaces. A program called **slattach** can be used to configure a serial port to work as a slip link. When slattach is run, it creates an interface name for that port (the name will be something like **sl0**).

AX.25 network devices work the same way. Since it's a single-use device, the kernel driver for a board like the Ottawa PI card will automatically create the interface and name it. But to use a serial port to drive a KISS TNC, you need to use a program called kissattach to configure the port and give it a name. The syntax for axattach is simple:

```
/usr/sbin/kissattach -i 44.71.36.8 \
-m 256 /dev/ttyS1 vhf
```

This command turns serial port "/dev/ttyS1" into an AX.25 port named "vhf", gives it an IP address of 44.71.36.8 (the "-i" flag, and assigns an mtu of 256 to the port (the "-m" flag). It will also create a network interface called "ax0" (for the first TNC attached). Further parameters for this device, like its speed, are set in the **axports** file described a bit later.

The port name is used by most of the ax25 utility programs, while the network interface name is used by Linux network programs like ifconfig.

Callsign Confusion

Simply creating a network interface isn't enough. You also need to assign a callsign to the interface and set basic operating parameters. One aspect of Linux AX.25 that's different from NOS is that you can't get away with just one call (in this discussion, when I talk about "call" I mean a unique callsign/SSID combination) for the whole system. You'll need at least one call for each physical port, plus perhaps several others

for specific services. Turning this requirement into practice is one of the most confusing things you'll have to deal with if you're building a system with more than one radio interface, or that uses NET/ROM as a protocol. Here's the scoop.

You need to have a unique call assigned to each physical port (i.e., a TNC or other device hooked to a radio) on the system. These calls will be used as the "from" callsign for outbound TCP/IP packets. Although you can use these calls for other AX.25 programs, you don't have to. And, as long as you send a legal ID according to the requirements of your regulatory authorities, the calls used for physical ports don't have to be real – you can use made-up calls for these ports if you want.[17-5] In a system configured the way I describe here, these calls will only be used for outbound IP traffic and won't be used for AX.25 or NET/ROM connections; users don't need to know about them.

Next, you need to have calls associated with "virtual" ports such as NET/ROM or AX/IP. Under some circumstances, these calls can be shared with other services. In the configuration described here, there is one unique NET/ROM call that's used on the air; two other NET/ROM ports (one for the BBS and one for the node software) share calls used by other services. The AX/IP daemon, if you use one, will require one call of its own.

Finally, there are the calls used by servers. In our system, there are two services that listen directly for, and respond based on, calls. One is the **ax25d** daemon that parcels most incoming AX.25 connect requests to the appropriate place (like the LinuxNode software that provides a G8BPQ-like user interface), and the other is the F6FBB PBBS software that listens for its own callsign. The rule here is that the FBB call (configured in FBB's init.srv file) must not be used by any other service. More generally, servers that listen directly to interfaces cannot share calls with one another. The only exception is that the FBB call can (but need not) be shared with the NET/ROM port it is listening to.

[17-5] *Some countries view fake calls in the AX.25 "FROM" field to be a violation of the rules, so you'll need to determine for yourself whether you can get away with this.*

The ax25d configuration file allows great flexibility — it can be set up to listen and respond to any call on any port. Unfortunately, FBB doesn't use it. But for other applications, like the LinuxNode software, you can configure the system to use the same or different callsigns on each port, have different programs run depending on the call used, and have different actions occur based on the user's call. This is all described below.

One suggestion from Linux/AX.25 gurus, and one that I follow on the systems I run, is to start with a high SSID for the hardware ports, and work your way down, while starting with low SSIDs on the service ports. Here's a list of call allocations for a hypothetical and moderately complex system that illustrates this:

```
# Hardware and virtual ports
W8APR-14        2m 1200 baud port with KISS TNC
W8APR-13        UHF 9600 baud port with KISS TNC
W8APR-12        19.2kb UHF port with Ottawa PI card
W8APR-11        NET/ROM default port
W8APR-10        AX/IP port

# Services ports
W8APR-0         F6FBB User Port
W8APR-1         Node Front End
W8APR-2         DX Cluster
W8APR-3         Callbook Server
W8APR-4         APRS Digi
```

The AXPORTS File

The minimal (and mandatory) configuration of AX.25 ports is done through a text file called **/etc/ax25/axports** which has one line for each physical AX.25 interface. The call set here is the one used for TCP/IP connections, and by default for other services unless overridden. The file should look something like this:

```
# axports file; the format is:
# portname, call, speed, paclen, window, description

# portnames for KISS TNCs are those assigned
# with axattach
vhf W8APR-14  4800      256  3    2M 1200 baud radio port
uhf W8APR-13  19200     256  3    UHF 9600 baud radio port

# PI card uses a portname that's the same as auto-
# assigned interface name.  Setting the speed to 0
# means that the card gets clock rate externally
pi0a     W8APR-12  0         256  3    UHF 19.2 radio port
```

The NRPORTS and NRBROADCAST Files

If you're going to configure NET/ROM in your system, you'll need to configure these two files and at boot time, start the netromd daemon that manages the NET/ROM routing broadcasts. Although NET/ROM can work adequately for IP routing with only a single defined port, if you want to offer user access I suggest you set up two, and possibly three or four, NET/ROM logical interfaces. One is the mandatory default interface, using W8APR-11 in our example. The second is listened for by ax25d and runs the LinuxNode software when used to make a connection request, and the third does the same thing for the F6FBB program if you use it. A fourth could be used to allow direct access to a convers node; we'll describe that in a later chapter.

The **/etc/ax25/nrports** file defines these logical interfaces. Here's an example:

```
# /etc/ax25/nrports
#
# The format of this file is:
#
# name callsign alias paclen description
netrom   W8APR-11  #MVFMA    235  Switch
netbbs   W8APR-0   MVFMA     235  FBB Port
netnod   W8APR-1   BBROOK    235  Node Port
```

With this configuration, users who connect to either W8APR-0 or MVFMA will be connected immediately to the F6FBB program. Those connecting to W8APR-1 or BBROOK will get to the LinuxNode front end. Users never need to use, and will never see, the W8APR-11/#MVFMA call, but that is what will appear on NET/ROM network-level packets (for example, level 2 connections between two nodes, or routing messages). Note that the default port is tied to an alias that begins with a "#" sign. That will keep it from showing up in standard node lists; users listing nodes will only see the MVFMA and BBROOK aliases.

The **/etc/ax25/nrbroadcast** file sets up the NET/ROM network-layer parameters. Here's a sample nrbroadcast file:

```
# /etc/ax25/nrbroadcast
#
# The format of this file is:
#
# ax25_name min_obs def_qual worst_qual verbose
pi0a     5    250  150  0
```

In this case, we are broadcasting NET/ROM info on one port. It's a high-quality, fast link, so the default quality is set to 250. We only want fairly high quality nodes to show up via this port, so the minimum quality is set to 150. Setting verbose to 0 means that we'll broadcast only our own node information on this port, and not the whole routing table. We could, of course, broadcast routing on multiple ports if we wanted to. There's no provision in this file to set the NET/ROM timers. You can set the broadcast interval, which defaults to 60 minutes, by passing an appropriate command to the netromd daemon when it is started: **/usr/sbin/netromd –t 15** to set the broadcast interval to 15 minutes, for example.

Other Parameter Setting

In the chapters on NOS, I went into painful detail about setting low-level parameters such as TXDelay, paclen, mtu, etc. How are those things set in a Linux/AX.25 system? There are several different tools that you'll use to tune your system.

First, the lowest level hardware settings are configured by a device-unique configuration program. For example, **/usr/sbin/kissparms** will set the TXDelay, slottime, ppersist, and other values for a KISS TNC, and **piconfig** and **sccinit** do the same thing for PI cards and generic 8530 cards (like the DRSI card) respectively. A typical kissparms command might look like this:

```
kissparms -p vhf -t 300 -s 300 -r 32
```

This sets the TNC attached to port vhf to a TXDelay of 300 milliseconds, slottime of 300 milliseconds, and a ppersist value of 32. **kissparms** can also be used to take a TNC out of KISS mode and back into normal command mode by issuing the command:

```
kissparms –p vhf x
```

The second group of parameters, those which operate at the AX.25 protocol level, may be set for each interface by accessing a file in the special directory (actually, filesystem) called **/proc**. If you cd into the directory **/proc/sys/net/ax25**, you'll see a directory for each of the physical network interfaces on the system, like ax0 or pi0a.

This is one place where the distinction between a network device or interface and a port is important. Even though you assign a name like "vhf" to your KISS TNC with the axattach command, to the system, the first attached TNC gets the device name **ax0**, the second **ax1**, etc.

If you go into one of these directories, you'll see a bunch of files, each of which contains only one line with a single value. Here's what they are, and what they mean:

FileName	Meaning	Values	Default
ip_default_mode	IP Default Mode	0=DG 1=VC	0
ax25_default_mode	AX.25 Default Mode	0=Normal 1=Extended	0
backoff_type	Backoff	0=Linear 1=Exponential	1
connect_mode	Connected (VC) Mode	0=No 1=Yes	1
standard_window_size	(MAXFRAME)	1 <= N <= 7	2
extended_window_size	Extended Window	1 <= N <= 63	32
t1_timeout	T1 Timeout (FRACK)	1s <= N <= 30s	10s
t2_timeout	T2 Timeout (RESPTIME)	1s <= N <= 20s	3s
t3_timeout	T3 Timeout (CHECK)	0s <= N <= 3600s	300s
idle_timeout	Idle Timeout	0m <= N	20m
maximum_retry_count	N2	1 <= N <= 31	10
maximum_packet_length	AX.25 Frame Length	1 <= N <= 512	256

Most of these should be familiar to you by now. The parallel "extended" settings are for a new mode of AX.25 that allows more frames to be in transit at any time. It's supported (at present) only by Linux, so it isn't very useful in mixed environments and you can probably ignore those settings.

You can change these values by changing the values in the files. However, these aren't real disk files that you can use an editor on; they're actually a way to look into the kernel's data tables stored in memory. To change them, you need to use the **echo** program, which dumps text to its standard output. Here's an example of how to change

the value of the standard_window_size, or MAXFRAME, parameter from the default of 2 to 4 for the ax0 interface:

```
echo 4 >/proc/sys/net/ax0/standard_window_size
```

If you're changing many of these values, you'll want to write a script that runs at boot time to set these values automatically. Alternatively, I've written a little script called **axsetparms** that, together with a couple of helper scripts, lets you easily edit and reset these values. axsetparms and its helpers are available from the TAPR site.

Next, you'll want to set IP and TCP parameters. These are controlled by the ifconfig and route commands. The ifconfig command allows you to set the maximum transmission unit (mtu) as well as the IP address, network mask, broadcast address, and other values of an interface. Here's an example:

```
/sbin/ifconfig ax0 44.71.36.2
/sbin/ifconfig ax0 hw ax25 w8apr-14
/sbin/ifconfig ax0 broadcast 44.71.36.127
/sbin/ifconfig ax0 netmask 255.255.255.128
/sbin/ifconfig ax0 arp # enables arp protocol
/sbin/ifconfig ax0 mtu 256 \
/sbin/ifconfig ax0 up   # turns the interface on
```

You can supply multiple options to a single ifconfig command, but this example breaks each option into a separate command for clarity. ifconfig uses the interface name, rather than the port name.

You can specify the initial round-trip-time in milliseconds, the maximum segment size (mss) and the window separately for each route on the system using the route command:

```
/sbin/route add -net 44.71.36.128 irtt 1500 \
mss 216 window 256 pi0a
```

Since the Linux networking code assumes that it's working over an ethernet, the default values for these parameters are far from optimal for a radio channel. For example, the MTU defaults to 1500 bytes, the window and mss values may be several kilobytes, and the irtt is very short. You'll definitely want to use ifconfig and route to optimize your system for radio use.

These are the main tools you will use to configure your system, but there are others available. The arp command allows you to manipulate the arp tables and configure a proxy arp if you need one. The axparms program can be used to manipulate the AX.25 routing table, to change the callsign associated with the port, or to associate specific incoming callsigns with system usernames (this allows AX.25 users to appear to the system as if they are logged in).

Once you have the AX.25 configuration complete, you can test it using the **call** and **listen** programs (both found in /usr/bin since they're end user, and not system, programs). call can be used to establish an outbound AX.25 connection:

```
/usr/bin/call vhf KE8TQ
```

The two parameters are the interface name, and the call of the station you're trying to reach (you can also include digipeaters). Once you've established a connection, you'll be in a line-oriented program that operates much like the Unix **cu** program. What you type is sent to the other station, and what the other station sends is displayed on the screen. Commands for the local system begin with a tilde ("~") character. To close the connect, issue the command "~." (tilde, followed by a period) at the beginning of the line.

The **listen** program lets you monitor traffic on an AX.25 interface. You can invoke it by typing

```
/usr/bin/listen
```

and your screen will display the incoming traffic from all AX.25 interfaces. listen

offers several options to tune the display; the most useful is probably the "-p <portname>" command that lets you choose a single port to monitor. To end a monitoring session, type control-C at the console.

Configuring TCP/IP over AX.25

Once you've set up the AX.25 interfaces and verified that they work, you need to configure IP routing for your radio interfaces. To do this, use the **route** command just as you would for an ethernet or slip route. For example, to send all ampr network traffic out via interface vhf, use the following command:

```
/sbin/route add -net 44.0.0.0 vhf
```

As noted above, you can include mss, window, and irtt values in the route command, and to optimize performance you should do so.

Congratulations! You've just put your Linux system on the air. Amateurs accessing it via TCP/IP will be able to use any of the network facilities on your machine – ftp, telnet, smtp, even the Worldwide Web. [17-6]

Automating the AX.25 Configuration

You probably noticed that several different programs are used to configure the AX.25 networking system. It would be a real pain to have to manually type all those commands each time you fired up the computer. You can avoid that by adding those commands to the Linux startup process.

Although different distributions of Linux do it in slightly different ways, using different file and directory structures, all Linux systems execute a series of runtime scripts (roughly equivalent to MS-DOS batch files on steroids) at startup. One of these scripts is usually called **rc.local** and it's typically located in a subdirectory of the /etc directory (often **/etc/rc.d**). You can insert the AX.25 configuration commands in this file, and they'll be

[17-6] *Of course, there's much more tweaking you can do to your network configuration; see a Linux "Network Administration" book for details.*

executed at startup.[17-7] Here's a sample rc.local file that sets up the W8APR system:

```
#!/bin/sh
# /etc/rc.d/rc.local

echo Starting AX.25 networking services:

# ax ports are:
#   vhf – 145.61 1200 baud (KISS TNC)
#   pi0a – UHF 19.2 repeater (PI Card)

# set up the TNC
/usr/sbin/kissattach /dev/ttyS0 vhf
/usr/sbin/kissparms -p vhf -t 250 -s 250 -r 32
/sbin/ifconfig vhf 44.71.36.2 hw ax25 w8apr-14
/sbin/ifconfig vhf broadcast 44.71.36.127
/sbin/ifconfig vhf netmask 255.255.255.128
/sbin/ifconfig vhf arp mtu 256 up
/sbin/route add -net 44.71.36.0 \
    irtt 4000 mss 216 window 256 vhf

# configure the Ottawa PI card
/usr/sbin/piconfig pi0a speed 19200 txdelay 30 \
    persist 32 slot 30 squelch 1
/sbin/ifconfig pi0a 44.71.36.130 hw ax25 w8apr-13
/sbin/ifconfig pi0a broadcast 44.71.36.255
/sbin/ifconfig pi0a netmask 255.255.255.128
/sbin/ifconfig pi0a arp mtu 256 up
/sbin/route add -net 44.71.36.128 \
    irtt 1500 mss 216 window 256 pi0a

# use axsetparms script to copy device
# parameters from /etc/ax25/parms into
# /proc/sys/net/ax25
/usr/sbin/axsetparms

# attach the NET/ROM device
/usr/sbin/nrattach -i 44.71.36.1 netrom
/usr/sbin/nrattach -i 44.71.36.1 netbbs
/usr/sbin/nrattach -i 44.71.36.1 netnod
```

[17-7] *The rc.local script is usually the last of the runtime scripts to be executed, so all the other system and networking daemons will be running by that time. That's normally not a problem, but if it is, there are usually one or more runtime scripts that configure the traditional network services, and you could inclde the AX.25 commands there.*

```
# set up the AX/IP port
/usr/sbin/kissattach -i 44.71.36.1 /dev/ptyqf axip

# set associations so a few of us can login
# to the node locally
/usr/sbin/axparms -assoc ke8tq fmp
/usr/sbin/axparms -assoc n8ur jra
/usr/sbin/axparms -assoc n8bjq n8bjq

# start the daemons
/usr/sbin/ax25d
/usr/sbin/netromd
/usr/sbin/mheardd
/usr/sbin/ax25ipd

# note that the following two commands require
# the "&" to put them in the background.  Without
# it, the script will hang forever.
/usr/sbin/axdigi &
/var/fbb/xfbb.sh -d &   # starts fbb
```

We haven't talked yet about some of these commands. Don't worry; we will.

Other Linux AX.25 Capabilities

In addition to TCP/IP, the Linux AX.25 toolkit includes several other goodies. Because our focus is on TCP, I won't provide much detail on them, but currently available features include the ability to map incoming AX.25 connections to specific programs using a daemon called **ax25d**. Several programs take advantage of this inbound-connection capability: **pms** is a personal message system that allows users to login via AX.25 and leave messages for system users; **axspawn** can can automatically create user accounts for AX.25 users when they connect to the system; the LinuxNode program (called **node** for short and described later) provides a NET/ROM-like user interface that allows a Linux system to act as an AX.25 and NET/ROM switch; and **axconv** (described in more detail later) allows users to be dropped into a convers program. These programs are probably just a sample of what we're going to see, as the ax25d daemon allows you to let users execute any program that runs under Linux when they connect to your station.

Configuring ax25d

The **ax25d** program is a central part of the Linux AX.25 suite, and it's worth a few minutes of our time. ax25d is a daemon program (one that runs in the background on the system, waiting for input) that monitors all the AX.25 interfaces for incoming connections. When a connection arrives, ax25d determines, based on the origin call, destination call, and the interface, what to do with it. For example, you can configure ax25d to route any connection arriving on the NET/ROM interface to the node program, or any connection arriving on the 2m port addressed to callsign W8APR-2 to the DX Cluster software. You can also direct connections based on source callsign; there are lots of possibilities here, not the least of which is locking out troublesome neighbors.

The ax25d program gets its configuration from the file **/etc/ax25/ax25d.conf** (not much of a surprise, really). Here's a sample (and simple) ax25d.conf file:

```
# /etc/ax25/ax25d.conf
#
# ax25d Configuration File.
#
# AX.25 Ports begin with a '['.
# NET/ROM Ports begin with a '<'.
#
[BBROOK VIA vhf]
NOCALL * * * * * * L
default * * * * * * - root /usr/sbin/node node
#
[W8APR-1 VIA vhf]
NOCALL * * * * * * L
default * * * * * * - root /usr/sbin/node node
#
[BBROOK VIA pi0a]
NOCALL * * * * * * L
default * * * * * * - root /usr/sbin/node node
#
[W8APR-1 VIA pi0a]
NOCALL * * * * * * L
default * * * * * * - root /usr/sbin/node node
#
<netnod>
NOCALL * * * * * * L
default * * * * * * - root /usr/sbin/node node
```

Note the structure of this file. First, at least some versions of ax25d complain if there are blank lines in the file, so it's best to make sure each line is either a command, or begins with a "#" character. Each destination call/interface pair has its own section, which continues until the next pair begins. Call and interface are delimited with "[" and "]" characters. The NET/ROM interfaces use "<" and ">" as delimiters, and only the interface, and not an associated call, is used.

Within each call/interface section, a separate line identifies what to do for each source call. The two lines in each section are the standard ones – the first locks out any station with a call of "NOCALL" while the second is a default that sends the connection off to the node software unless there's another match (this should always be the last line of the section). If you want to treat incoming stations differently based on their call, you can add additional lines for each specially-handled call.

The asterisks are option fields that aren't used in this example; see the ax25d.conf man page for details on what they represent. The FBB software, if present, does not use ax25d; it listens for a callsign defined in its own configuration file. Don't use the FBB callsign for any purpose other than FBB!

Configuring LinuxNode

The node program provides a NET/ROM node user interface much like the familiar G8BPQ software. In our example, node is fired up for any user-level (as opposed to network) connection via the netnod NET/ROM interface, and for any connection addressed to either W8APR-1 or BBROOK on any interface. The behavior and appearance of node is controlled by three files: **/etc/ax25/node.conf**, **/etc/ax25/node.info**, and **/etc/ax25/node.perms**.

node.conf is the primary configuration file; here is a simple version:

```
# /etc/ax25/node.conf - LinuxNode configuration file
#
# see node.conf(5)

# Idle timeout (seconds).
IdleTimeout   900

# Timeout when gatewaying (seconds).
ConnTimeout   3600

# Visible hostname. Will be shown at telnet login.
HostName mvfma.ampr.org

# ReConnect flag.
ReConnect     on

# “Local” network.
LocalNet 44.71.36.0/24

# Command aliases. See node.conf(5) for the meaning
# of the uppercase
# letters in the name of the alias.
#Alias
CONVers  “telnet %{2:mvfma} 3600 \”/n %u %{1:139}\””

# Node ID.
NodeId        BBROOK:W8APR-1

# NET/ROM port name. This port is used for outgoing
# NET/ROM connects.
NrPort        netrom

# Logging level
LogLevel 3

# The escape character (CTRL-T)
EscapeChar      20
```

The ability to define aliases (such as the one above that connects to the convers server) and to run external commands give the LinuxNode program great flexibility.

The node.info file is the text that is displayed when someone issues the "i" command from the node prompt. Here's ours:

```
This is a multi-protocol switch that supports AX.25,
NET/ROM, and TCP/IP using the Linux operating system.
It provides access to the W8APR PBBS System.  This
system is sponsored by the Miami Valley FM
Association and is located in Bellbrook, Ohio.

For more information, contact jra@febo.com or
N8UR@W8APR.#DAY.OH.USA.NOAM.
```

The node.perms file defines the privileges that node users have, based on their callsign:

```
# /etc/ax25/node.perms - LinuxNode permissions file
#
# user   type port passwd    perms

# User n8ur can login without password from
# anywhere else but 'inet'.
#
n8ur     inet      *    qwerty    95
n8ur     *         *    *         95

# WA8ZWJ is a bbs so it needs escape disabled.
#
wa8zwj   *    *    *         287

# Default permissions per connection type.
#
*   ax25      *    *         31
*   NET/ROM   *    *         31
*   local     *    *         31
*   ampr      *    *         31
*   inet      *    *         0
*   host      *    *         31
```

The connection type (second field may be one of the following:

```
*         Wildcard; matches any type of connection.

ax25      Matches users coming in with AX.25.

NET/ROM   Matches users coming in with NET/ROM.

rose      Matches users coming in with ROSE.

local     Matches TCP/IP connections where
          user's host is in "local" network as
          defined in node.conf.

ampr      Matches TCP/IP connections where
          user's host is in amprnet (44.0.0.0/8).

inet      Matches TCP/IP connections where
          user's host is neither in "local" net_work
          nor in amprnet.

host      Matches users starting LinuxNode from shell.
```

The permissions field works the same way as the NOS ftpusers file; the number is generated by adding together the values for the various permissions granted to that user. Here are the values you can use:

```
1       Permits logging in even if no other
        permissions are given.

2       Permits outgoing AX.25 connects.

4       Permits outgoing NET/ROM connects.

8       Permits telnetting to hosts in the
        “local” network as defined in node.conf(5).

16      Permits telnetting to hosts in amprnet.

32      Permits telnetting to hosts neither in
        the “local” network nor in amprnet.

64      Permits using hidden ports in outgoing
        AX.25 connections.  (See Hidden_Ports
        command in node.conf(5).)

128     Permits outgoing ROSE connects.

25656   The no-escape flag.  Disables the escape mechanism
        for this user.
```

Unfortunately, these values **don’t** match the ones NOS uses in ftpusers.

Chapter

18

Internet Gateways

Introduction – and Warning!

Gateways aren't technically too complicated, but there are significant security and legal considerations that affect every gateway, and coordination among gateway stations is probably more important than in any other part of packet radio. Before tackling a gateway, you should have a solid working knowledge of the software you're using, the security and legal issues involved, and the need for coordination with other gateway operators.

Why Can t We Directly Connect to the Net?

NOS is tuned for amateur radio applications, but because TCP/IP is the universal language of the Internet, there's no technical reason why a NOS system can't route packets between ham radio networks and the Internet. And, of course, a Linux/AX.25 system is designed with the Internet in mind. There are some good reasons, though, why we don't want to set up an amateur IP system to blindly route between the local ham network and the Internet.

Technically, such a gateway would cause problems because each IP network, including the 44.0.0.0 amateur network, should have only a single point of connection to the Internet. Remember that our ham network isn't fully connected; we are lots of isolated

subnets who can't talk directly to each other – if we could, we wouldn't *need* gateways! Multiple connection points linking pieces of the ham network directly to the Internet both violates Internet standards and would require complex routing tables among the commercial gateways on the Internet's backbone. In today's environment, that's not practical.

Legally, mixing the amprnet and the Internet causes several problems. First, when data originated by a non-ham (such as an Internet web server) is transmitted over the air, it's considered third-party traffic, and most countries have stringent rules that limit when, and how, hams can transmit third-party messages. Second, it would be very difficult to ensure that traffic being sent from the Internet over a radio network would comply with the restrictions on commercial traffic, obscenity, etc., imposed by our ham licenses. Finally, there's a reverse security issue – Internet traffic that's sent by ham radio is percieved by the Internet folks as insecure, and sensitive data such as passwords might be grabbed by the Bad Guys. The organizations who we might ask to give us connectivity — typically a university computer science department or a corporate IS organization — are concerned about what a wide-open connection to a bunch of hams might do to their security.

How Gateways Work

Despite all the problems I've outlined, we *do* have things called Internet gateways, so there must be some way to solve the security and legality issues. What's the secret?

There are two one-word answers: **encapsulation** and **addressing**. If you recall those long, boring chapters at the beginning of the book, the whole TCP/IP protocol stack is built on the concept of encapsulating one protocol within another. TCP is encapsulated within IP, which is encapsulated within AX.25. Encapsulation isn't a one way street. It's possible to encapsulate lower level protocol frames inside higher level ones, or even a frame of one protocol type within another of the same type.

An amateur-radio-to-Internet gateway is an IP host that has (at least) two interfaces, one that connects to the Internet, and another that connects to the amprnet. Each of these interfaces has its own IP address, and that address is local to the network the interface connects to. The radio interface will have an IP address in the 44.0.0.0 network, while the interface that connects to the Internet has an non-ham IP address that is routable over the Internet.

The gateway encapsulates packets from the radio interface within IP frames that are sent out via the Internet interface. In other words, we "wrap up" our amateur radio data inside innocent-looking IP frames that are sent out over the Internet with "real" IP addresses. The left part of Figure 18-1 shows what happens.

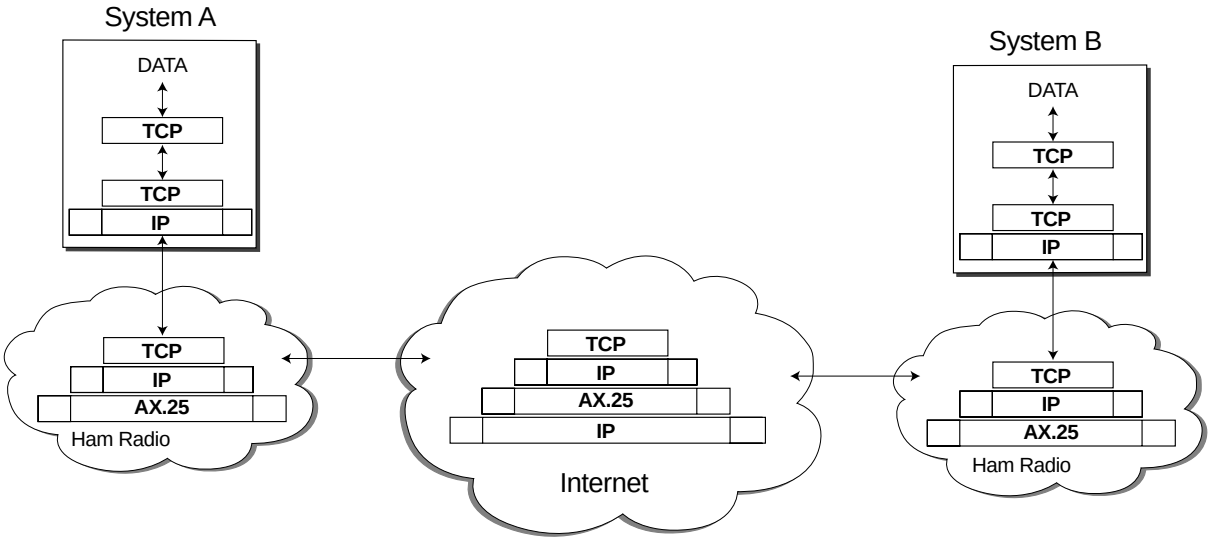


Figure 18-1 - example of encapsulation

Now that we've hidden our data inside an IP frame that can travel across the Internet, where do we send it? To another gateway system, of course. More specifically, we send it to the Internet address of another gateway system. That station unwraps the frame, extracts the original packet, and ships it off to the ham network. The right part of Figure 18-1 shows the recipient's role in the process.

The critical difference between an ordinary IP router and a packet-internet gateway is that amprnet packets are not just passed on to the next host in line, but are hidden inside packets that have nothing to do with the 44.0.0.0 network. The ham packets are treated as data within standard Internet packets. This use of encapsulation and non-ham IP addresses has two important consequences.

First, no network 44 data flows over the internet — you can snoop the backbone for as long as you like, but you'll never see a 44.x.x.x address show up as either the sender or recipient of a packet.[18-1] Second, access to and from the ham network is controlled: the gateway station, as part of its wrapping and unwrapping process, can protect both sides of the system from unauthorized access.

An AX.25 Internet gateway system has a route to each of the other gateways it wants to reach, and each of those other gateways must have a route pointing back to it. These routes use only non-ham IP addresses, and they must be manually configured. As a result, there's a fair amount of coordination involved in running a gateway; you can't just put one together and sit back. Later in this chapter I outline some of the resources that you can use to help you coordinate your gateway with the rest of the amprnet world.

Gateway Platforms

If you'd like to run an Internet-to-packet gateway, there are three different routes you can take: JNOS or TNOS under DOS, JNOS or TNOS under Linux, or Linux native AX.25. Frankly, I'd recommend against running a gateway under DOS, unless you're

[18-1] *Actually, there is a route that connects the 44.0.0.0 amprnet to the Internet at large, but it only talks to gateway stations, and doesn't provide a complete interface between the two.*

running one of the recent NOS versions that use "DOS extender" technology to use memory above 640k. The memory limitations we talked about earlier make it very difficult to configure a "traditional" NOS gateway system that will be stable and reliable.

As between NOS/Linux, and native Linux, the choice is more difficult. If you are running TNOS as a PBBS and want to allow some users from the Internet to have some access to your network, TNOS can handle both the gateway and PBBS functions in one system. It also offers sophisticated security features that let you define exactly what users can and cannot do with the system, depending on whether they are accessing it from the Internet or from the radio world, and depending on what user or domain name they have. JNOS/Linux doesn't have quite as many security knobs to tune, but will also do a good job.

On the other hand, if you only want to provide gateway capabilities and have another system provide user services, the native Linux system offers a "cleaner" approach.

A significant number of people are using each of the NOS/Linux and native Linux approaches. Your decision on which to use should be based on what you want your gateway to do, and on which environment is more comfortable to you.

Gateway Types: AX.25 and IP

It's possible to encapsulate any protocol within any protocol. NOS supports two forms of encapsulation: AX.25 within IP, and IP within IP. AX.25 (or **AX/IP**) encapsulation means that any sort of ham traffic that is based on AX.25, including NET/ROM, can be sent via the gateway. An AX/IP gateway looks to end users like a digipeater, a NET/ROM node, or both. The remote gateway appears to be another digipeater, or a remote NET/ROM node. The existence of the gateway is transparent to users — it acts like any other link between a pair of digis or nodes, though it may be a few thousand miles longer!

IP within IP (or **IP/IP**) encapsulation limits gateway use to TCP/IP users. An IP/IP gateway acts just like a normal IP switch; ham IP users set their default route to use the gateway, and they can reach any other amprnet subnet that has a gateway-to-gateway (or "encap") route available.

Most gateways support AX/IP encapsulation, though some of these don't support NET/ROM. A few support only IP/IP, on the theory that we should be encouraging IP use and not the use of sub-optimal protocols. You'll have to make your own decision. Of course, the gateways at each end of a link must both support the same encapsulation types or things won't work.

There's one important consideration before you begin. Setting up the software to support a gateway is straightforward. But the gateway isn't of any use unless there's another gateway for you to talk with. Ham Internet gateway operation is a community activity, and to make it worthwhile, you need to be tied into that community. There are two important sources of information that you need to consult before you plunge in. First, there is an email mailing list for gateway operators, run by Tony Querubin AH6BW. Drop Tony email at **gateways-l-request@lava.net** asking to subscribe (be patient; it may take a couple of weeks to be added to the list).

The other resource is the web site **http://www.fuller.net/Gateways**, which is run by James Fuller, N7VMR. James' site contains two documents written by Warren Toomey, VK1WXT, that are required reading for any gateway operator: Setting Up a Packet-Internet Gateway, and The Amateur Packet Radio Gateway FAQ. James also runs an automated system for adding information to the "encap.txt" file, which is the official source of routing information for gateway community and is updated daily. You need to register your site so that others know how to route to you, and obtain the encap.txt file yourself so that you know how to route to them. Information on how to do this is available at fuller.net.

Configuration Basics

No matter what system you use, or whether you are using AX/IP or IP/IP encapsulation, the basic gateway configuration steps are the same. We'll talk about NOS gateway configuration first, followed by Linux/AX.25.

NOS IP/IP Configuration

You should first concentrate on getting IP-over-IP gatewaying working. Using TNOS/DOS[18-2] as an example, the AUTOEXEC.NOS file needs, in addition to the other configuration information we've talked about earlier in this book, a set of commands that define the encapsulated routes. For this example, assume that the radio interface is called "vhf", has an IP address of 44.71.36.1, and serves the [44.71.36.0]/24 network. The Internet route is via an Ethernet interface called "eth0", which has an IP address of 192.168.1.1, and is on the 192.168.1.0 network.

The first thing to do is tell TNOS that it will need to encapsulate data. Encapsulation works through a logical interface (similar to the logical NET/ROM interface). The commands

```
ifconfig encap ipaddress 44.71.36.1
ifconfig encap mtu 576
```

tell TNOS that it will be encapsulating data addressed to 44.71.36.1 (the IP address of the gateway), and establishes an MTU of 576 for encapsulated packets (choose an appropriate MTU based on the size of the encapsulated frames you will be sending).

The next step is to establish routes for the encapsulated data. These should be the first routing statements you put in AUTOEXEC.NOS; the default route and other non-encap routes should follow them. An IP/IP gateway route looks like this:

```
# Example route only – don't use this in
# the real world!
route add [44.64.0.0]/16 encap 192.168.2.1
```

[18-2] *I'm using DOS NOS for this example because its direct support of Ethernet cards simplifies things. In a TNOS/Unix system, you'd need to use a pty slip link, as an extra step in reaching the Internet.*

Specifying "encap" as the interface is the magical part of the command. This route says that packets for the class B-sized subnet 44.64.0.0 can be sent to Internet host 192.168.2.1, which has RF connectivity to all the amprnet networks in that address space. Of course, instead of a single encap route in your file, you'll probably have many, because a manual route must be entered at both ends of each gateway-to-gateway link. Maintaining the routing table can quickly become painful. Using the encap.txt file from fuller.net will make the task much easier. To do this, instead of putting encap routes directly in AUTOEXEC.NOS, read them in from a separate file using the **source** command:

```
# Source the routing file externally
source encap.txt
```

If you've provided your gateway information to fuller.net for inclusion in encap.txt, you could create serious routing loops by setting up an encap route to yourself. To avoid this, either edit your own info out of encap.txt before using it, or load it before any of the other routes – that way your local routes will overwrite the ones in the encap file.

You should also adjust the **ip ttl** value in AUTOEXEC.NOS to 32 or even higher, as Internet packets can go through lots of routers on their way from gateway to gateway.

Security Basics

Those few commands are all that's necessary to make a gateway work in IP/IP mode. However, a setup like this will result in a gateway that has serious security holes. There are two forms of access control supported by JNOS and TNOS, and similar control is available within the Linux networking code.

The first security measure is called **IP access control**. You can configure your gateway to forward packets only to and from certain ranges of IP addresses. For example, the commands

```
# format is: ip access [permit|deny|delete] [proto]
#                      address range iface

# allow all amprnet addresses in
ip access permit any 44/8 vhf

# block everything else
ip access deny any all vhf
```

will allow packets to and from amprnet addresses to be forwarded over the "vhf" radio interface, but packets from other address ranges will be blocked. IP access control is the minimum security you need to run a safe gateway, and it may be all you need if you don't offer other services like smtp mail handling, or mailbox login, on your gateway machine.

If your gateway does offer additional services that are based on the TCP protocol, you can use **TCP access control** to provide security for them. TCP access control allows you to specify addresses that can access a specific TCP service port. For example, the convers server usually uses TCP port 3600. The following commands will limit access to the server so that only amprnet hosts and machines on the local ethernet will be able to use it:

```
# allow all amprnet addresses to reach the convers
# server
tcp access permit 44/8 1 3600

# allow the local network
tcp access permit 192.168.1.0/24 1 3600

# block everyone else
tcp access deny all 3600
```

You should consider adding access controls for all the services your gateway system offers. This, by the way, is a good argument for configuring the gateway to provide only minimal services – if the gateway machine does nothing more than pass IP traffic, and another system on the amprnet side of the gateway provides other services, you will close many possible security holes.

By the way, TNOS offers its own set of security options that allow more fine-grained tuning of permissions. If you are running a TNOS gateway, it will be worth your time to study the "security" commands and use them to maximize your system's security.

NOS AX/IP Configuration

Once you have the IP/IP gateway functions working correctly, you can (if you wish) add an AX/IP gateway to encapsulate AX.25 packets within IP. The AX/IP gatewat usually interfaces to the RF AX.25 network as if it were a digipeater; to use the gateway, stations include the local gateway call in their connect string. In NOS, you enable AX/IP by attaching a logical interface defining the local call and remote IP address of the route. Since each AX/IP route requires a unique call/SSID at each end, the practical number of routes you can use is limited.

Once the interface is in place, any packet sent to the local gateway call will be encapsulated and sent to the gateway specified by the remote IP address, which will unencapsulate it and send the packet out over the air to its destination.

As an example, let's say I want to establish an AX/IP route from W8APR to AA9A's gateway. The first step is to get IP/IP encapsulation between our systems working as described above. Next, Tom and I each need to create an AX/IP logical interface and associate it with a unique call/SSID pair:

```
# attach axip command is:
# attach axip <iface> <mtu> <remote IP> <local call>
attach axip ip0 256 192.168.2.1 W8APR-11
ifconfig ip0 desc "AX/IP to Appleton, WI"
ifconfig ip0 ipaddress 44.71.36.1
ifconfig ip0 broadcast 255.255.255.255
```

This command creates an **ip0** interface that's associated with the Tom's Internet IP address 192.168.2.1, and a unique call/SSID combination at my end.[18-3] The **ifconfig desc** statement helps identify the link, and the other **ifconfig** statements set default parameters for safety sake – the actual IP address used isn't critical and can be one of the other ones assigned to the system. Tom's configuration would be similar, but would point to my gateway IP address and his call. The call used at Tom's end isn't part of my config file, but it needs to be known to my local users who want to use the link.

Finally, we need to ensure that digipeating is enabled both for the AX/IP interface, and for whatever RF interfaces will be used to access the gateway:

```
ax digi ip0 on
ax digi vhf on
ax digi uhf on
```

With these commands in place, a user on my network could issue a connect string like this to connect to WA9SXN, who is on Tom's radio LAN:

```
C WA9SXN via W8APR-11 AA9A-6
```

The AX/IP link creates a wormhole, or tunnel, over the Internet that gives users at each end the illusion they have a simple two digi path between their stations.

NOS NET/ROM Gateway Configuration

Configuring NET/ROM over IP is simple once you have the IP/IP and AX/IP links working. You use the **netrom interface** command to associate the NET/ROM system to the AX/IP port:

```
netrom interface ip0 230 100
```

230 is the default route quality, and 100 the minimum quality, to be used on this interface.

[18-3] *It's common to associate an alias-like call (perhaps "DAYGAT" in this case), rather than a call/SSID like N8UR-6, to the AX/IP port – whether you want to do that is a matter of personal taste.*

Although it's easy to configure a NET/ROM over IP gateway, make sure there's a good reason to do so. Using straight AX/IP encapsulation is more efficient – you don't have as many bytes of overhead – and NET/ROM gateways can rapidly pollute the local NET/ROM network by introducing hundreds of additional nodes that may confuse more than help your users. You'll recall that I stressed in the NET/ROM discussion above how important it was for all the nodes on a NET/ROM network to have compatible parameter settings – well, that's just about an impossible task to accomplish when gateway nodes are added to the mix. As they say, "You Have Been Warned."

Linux IP/IP Configuration

Linux supports IP/IP encapsulation using a "tunnel" driver that's available as a kernel option (and that is unrelated to the AX.25 code). It also supports AX/IP encapsulation using the ax25ipd program. One difference from NOS is that the **ax25ipd** is a standalone program – you can enable AX/IP encapsulation without setting up a separate IP/IP configuration.

To configure IP/IP encapsulation, first make sure that you've compiled the IP tunnel driver into your kernel. In the **make config** menus, you need to select **TCP/IP Networking**, then **IP: Forwarding/Gatewaying**, and finally **IP: tunneling**. The kernel is then ready to use tunnel devices with names like **tunl0** and **tunl1**.

To use a tunnel device, configure it with the ifconfig command like any other interface, and then add one or more routes for the tunnel routes you want:

```
/sbin/ifconfig tunl0 192.168.1.1 mtu 512 up
/sbin/route add -net 44.128.1.0 \
    netmask 255.255.255.0 tunl0 gw 192.168.10.1
/sbin/route add -net 44.128.2.0 \
    netmask 255.255.255.0 tunl0 gw 192.168.11.1
/sbin/route add -net 44.128.3.0 \
    netmask 255.255.255.0 tunl0 gw 192.168.12.1
```

As in NOS, the IP address assigned to the tunnel driver should be the same one used for the primary AX.25 interface on the machine.

Linux offers a set of security options similar to the **ip allow/deny** and **tcp allow/deny** commands in NOS. Check **host.allow** and **host.deny** man pages for configuration details.

As with NOS, to make your gateway useful you will need to provide encapsulated routes to other gateways. The encap.txt file from **http://www.fuller.net** can't be used as-is, because it is formatted for the NOS configuration file structure, but Ron Atkinson, N8FOW wrote a script that will convert it to the proper format for Linux. It's available from the TAPR site as **munge_encap.sh**.

It takes encap.txt as its standard input, and generates Linux route commands on its standard output:

```
./munge-encap.sh < encap.txt > encap.linux
```

Before using it, you will need to edit a couple of lines in munge_encap.sh to identify your local network. The output file (encap.linux in the example) can be run as a shell script to establish the encap routes.

Linux AX/IP Configuration

AX/IP encapsulation in Linux is independent of the tunnel driver, so if you want you can have AX/IP links without creating underlying IP/IP links. Linux AX/IP instead uses a standalone program, **ax25ipd**, to handle the encapsulation chores. It's configured by the **/etc/ax25/ax25ipd.conf** file. I have to admit that getting AX/IP encapsulation working on my system was one of the tougher challenges I've had, although the resulting configuration file is actually very simple. Making it work isn't too hard, but understanding why it works from the limited documentation and the configuration file proved more difficult for me.

Here's what it cost me lots of hair to figure out: **ax25ipd** does nothing more than create a logical interface that looks like a KISS TNC. It's attached to one end of a pseudo-tty pipe (you'll remember we discussed that earlier as part of configuring TNOS/Linux). The other end of the pty is the subject of a **kissattach** command, and the result is an ax25 interface that looks just like any other – it is configured with the **axports** file and can be used anywhere a real TNC interface could be.

The **ax25ipd.conf** file establishes the remote IP addresses and callsigns for each route. Unlike NOS, ax25ipd only requires a single call/SSID combination for all the AX/IP routes, rather than a separate one for each. It also requires you to know the call being used at the other end of the gateway. The gateway looks like a digipeater, and users connect through it just as with NOS:

```
Connect WA9SXN v W8APR-11, AA9A-6
```

Here's the kissattach command, which is normally issued before the ax25ipd daemon is started, as with the other non-IP interfaces:

```
# set up the axip port – use the master end of
# the pty pair
/usr/sbin/kissattach -i 44.71.36.1 /dev/ptyqf axip
```

Since the axip port is running before the daemon is started, it needs to use the master side of the pty pair. The ax25ipd daemon is started later, and in its simplest form, the **/etc/ax25ipd.conf** file looks like this:

```
# ax25ipd.conf

socket ip
mode tnc
speed 9600

device /dev/ttyqf   # slave end of the pair

loglevel 2

# Just one of many possible routes,
# each specified by the remote call
# and remote IP address
route AA9A-6 192.168.2.1
```

You shouldn't have to mess with the "socket" or "mode" parameters. Although the option to set the **mode** to "digi" looks interesting, it's not as useful as you might think, and the normal configuration is **mode tnc**. The **speed** setting is a placeholder since this is a logical interface that doesn't run at a defined baud rate. This example shows only a single route command, but you could have many, each one binding a remote IP address to a remote call. All the AX/IP routes use the same local call as the first digipeater in the connect string. That call is configured in **/etc/axports** like any other port:

```
# /etc/ax25/axports
# The format of this file is:
# name callsign speed paclen window description
axip    W8APR-11  -  255  4   axip interface
```

If you want users to access the AX/IP gateway via digipeater syntax as in NOS, you need to take one more step. Linux AX.25 doesn't support digipeating by default. To enable it, you must run the **ax25digi** program:

```
# start the ax digi
/usr/sbin/axdigi &
```

The "**&**" sign following the command tells Linux to run this program in the background; if you forget to include that character, the startup script will hang at that point forever.[18-4]

axdigi requires no configuration – just start it and it runs, allowing cross-port digipeating through your system. Unfortunately, there's no way to limit the ports that are allowed to digipeat – it will allow users to digi from any port to any other port – so keep that in mind if you decide to use this tool. As of this writing axdigi is not included in the standard ax25-utilities program set, but you can find it at the TAPR site.

[18-4] *If you accidentally get hung up by forgetting to include the "/" character after axdigi, you may be able to recover by switching to another Linux virtual console, logging in as root, and them killing the offending program, like so: killall axdigi.*

Linux NET/ROM Gateway Configuration

To use NET/ROM via AXIP encapsulation in Linux, first you need to make sure AXIP is configured and working properly. Then, add the axip port to the existing /etc/nrbroadcast file:

```
# /etc/ax25/nrbroadcast
# The format of this file is:
# ax25_name min_obs def_qual worst_qual verbose
axip    5       250     100     1
```

Again, you need to use care in setting these parameters, particularly the "def_qual" default quality value. The "verbose" flag is set to 1 because we want to propagate nodes via this port. The final step is to enable broadcasts via ax25ipd by configuring broadcast callsigns, and enabling broadcasts, on the routes that will carry NET/ROM traffic. Here's the ax25ipd file we looked at before, with some new lines added:

```
# ax25ipd.conf

socket ip
mode tnc
speed 9600

device /dev/ttyqf   # slave end of the pair

loglevel 2

# Broadcast Address definition. Any of the
# addresses listed will be forwarded to any of
# the routes flagged as broadcast capable routes.
broadcast QST-0 NODES-0

# Just two of many possible routes,
# each specified by the remote call
# and remote IP address
#
# The "b" flag indicates that broadcasts may
# be transmitted on the route; this is necessary
# to enable NET/ROM on that route.

route AA9A-6 192.168.2.1
route WA9EQP 192.168.3.1 b  # broadcast enabled
```

Chapter

19

The Converse Bridge

You may have seen "QSO bridges" or "Chat Boxes" that allow more than two packet users to carry on a round-table conversation. These are necessary because packet radio is a point-to-point protocol — error checking only works between a pair of stations. The QSO bridge solves this problem by retransmitting a packet sent by one member of the QSO to every other member. That way everyone is guaranteed to get an error-free packet (at the cost of the same data being transmitted multiple times).

Both JNOS and TNOS offer an advanced version of this capability called a convers[19-1] server. There's also a convers daemon for Linux. The convers server allows users to split into separate groups ("channels") to carry on their own conversations, so you don't have to try to pick your messages out of multiple QSOs (conversations). It also has other features that make it more similar to the Internet IRC (Internet Relay Chat) service than to the simple QSO bridge.

The biggest advantage of the convers server over the QSO bridge is that multiple servers can be linked together. In other words, the mvfma.ampr.org server in Dayton can link with the n8xja.ampr.org server in Springfield via a UHF link, and the round-table can include users at each end accessing the system on their local 2m channel. That's pretty cool, but now think about the Internet gateways we've been talking about. Picture a world-wide round-table, with hundreds of hams joining in. It exists today through Internet-linked convers servers.

[19-1] *No, I don't know why they left the "e" off "Converse."*

Configuring Convers: NOS

Configuring convers in NOS is straightforward. You give the server a name, tell it the remote hosts with which it will link and what radio interfaces it should use, and start it up:

```
# Provide the name of this server, usually your city or
# region name.
convers host Dayton

# decide which of the system interfaces will allow
# convers connects
convers interface vhf
convers interface uhf

# You can assign a callsign to the server; users who
# connect to it will be dropped directly into convers.
convers W8APR-5

# link to other convers servers.  The last parameter
# is optional and is the name of the remote server.  It
# is displayed with the /Links command only when that
# link is not up.
convers link 44.71.36.3 Springfield
convers link 44.71.14.5 Cincinnati

#start it up
start convers
```

Unlike gateway routes, each convers link does not have a reverse link at the other end – the link is established from one end only, and it's very important to make sure that convers links do not point back at each other. If they do, the result will be nasty loops. If you link to server A, who links to server B, neither one of them should link back to you. The convers server has loop detection code built in, but it's best to avoid the problem by designing link commands to ensure that they are one-way only.

Using Convers

There are three ways users can access the NOS convers server: First, if the **convers <mycall>** is set, AX.25 connections to that call will go directly to the server. Second, a user connecting to the PBBS by any means – AX.25, NET/ROM, or TCP/IP – can issue the "C" command to go to the server. Finally, TCP/IP users can telnet to the convers port (usually 3600) with a command like "telnet w8apr.ampr.org 3600".

Once you've connected to the server, everything you type will be retransmitted to every other user on the same convers **channel**. Channels allow convers bridges to support many users who may be carrying on separate conversations. Users on a channel see everything that other users on that channel type, but they don't see transmissions on other channels. When you first connect, you will be placed in channel 0, which serves as the calling channel.

Commands to the convers server begin with a "/" character – lines beginning with any other character are sent as data. To disconnect from the server, you can use either the **/bye** or **/exit** command. If you're logging in from another machine, you may need to set your login to your callsign with the **/name** command. You can change channels with the **/channel n** command, where n is the channel number. The **/users** and **/who** commands list all the users on each channel of the server; in a system with several links, there could be dozens of users. The **/links** command lists all connections to other hosts. The **/nick** command lets you establish a nickname (like your first name) and the **/note** or **/pers** command lets you set a personal ID string with information about yourself – other users can see that if they use the **/who** or **/whois** commands. There are several other commands, which vary somewhat depending on the convers software version. The **/msg** and **/write** commands let you send a private message to another user, by specifying the user call and then the message. The /? or **/help** commands will send you a help screen.

Convers for Linux

There are a couple of programs that will let a Linux/AX.25 system act as a convers server. One is **convers**, which derived from the German WAMPES software, and the other is **Tampa PingPong**, also known as tpp-convers, which Brian Lantz, KO4KS, adapted from convers. I'll describe how to configure **tpp-convers** version 1.14 with patches installed (the patched version is available on the book disk, or from www.tapr.org)[19-2].

First, make sure you're logged in as root, and then make an appropriate directory in which to unpack the sources; unlike the other packages described earlier, tpp-convers-1.14.tgz doesn't create its own directory. Next, unpack the archive file in that directory:

```
mkdir /usr/src/tpp-convers
cd /usr/src/tpp-convers
tar zxvf tpp-convers-1.14.tgz
```

You'll end up with a bunch of README files, and I suggest you read them. The files are a combination of README's from the original convers package, as well as Brian's new ones. Once you've looked at the documentation files, and in particular the "**INSTALL**" and "**README.TPP**" files, run the configuration program:

```
./Configure
```

Configure does a lot of checking to see what your system looks like, and it may take a while – be patient! It will set things up to match your installation. If you're using a version other than the one from TAPR, the default file locations won't be the same as those used by the other AX.25 tools. In that case, I suggest you edit the **Makefile** after running Configure to change the default file locations. Here's how I would change the tpp-convers Makefile:

[19-2] *As of this writing, tpp-convers has been at version 1.14 for a couple of years, although KO4KS has issued three patch files that fix some bugs. It's not clear if there will be another release of this, or a rumored successor program called TACS. In the meantime, the TAPR version of TPP, in a file called tpp-convers-1.14-n8ur.tgz, includes the three patch files, the change to file locations noted below, and several minor tweaks to make it compile properly under a current Linux (Debian 1.2, to be precise) system.*

```
             Default              Suggested
DATA_DIR=    /tcp                 /var/ax25/convers
CONF_DIR=    /usr/local/etc       /etc/ax25
BIN_DIR=     /usr/local/sbin      /usr/sbin
UBIN_DIR=    /usr/local/bin       /usr/bin
MAN_DIR=     /usr/local/man       /usr/man
```

You should manually create the **/var/ax25/convers** directory and a **personals/** subdirectory in it. Then, build and install the files with this command:

```
make install
```

You will end up with the server daemon, **conversd**, a user program used to access the server, **convers**, and a couple of useful utilities that I'll describe later. The programs will be installed in the directories defined in the Makefile.

Before running the conversd daemon, you need to edit its configuration file, **/etc/ax25/convers.conf**. Here's a sample, based on the one in the documentation:

```
# tpp-convers config file
Server        Dayton
Email         jra@febo.com
Secretnum     12345
Mode          internet
Sysinfo       Dayton TPP 1.14 Server
Security      restricted
Link          Hydra 1 1 hydra.carleton.ca:3600
```

The "Server" line defines the name of your server; the "Email" line is the address of the primary sysop; the "Secretnum" line is a password for administrator privileges (check the readme files to learn more about this), and the "Mode" line defines either "amprnet" or "internet" operating mode – if you're connecting to other NOS or conversd based servers, you'll want to set this to "internet" to provide full functionality. The setting of "Security" determines whether users need to be on an authorized list to connect to the server. Finally, the Sysinfo field is an additional description of your system for the benefit of other users.

Each “Link” line establishes a connection to one server. The first field is the short name of the server. The second and third fields are group value and quality respectively, and for most uses should both be set to “1”. The last field is the hostname and port number to use for the connection; 3600 is the standard convers port. As with NOS convers, you should make sure that your links are established in only one direction. Unlike NOS, tpp-convers allows you to specify more than one link. You can create a "chain" of links, and if the first link fails, the server will try the next in line, and so on. This can greatly improve the reliability of convers service, as the heavily used servers have a tendency to fall asleep once in a while.

Here is an example of how to configure a server chain:

```
Link    server1 1 1 server1.somewhere.net:3600
Link    server2 1 1 server1.elsewhere.net:3600 server1
Link    server3 1 1 server1.nowhere.net:3600 server2
```

When conversd starts, it will first attempt to connect to server1, and if that fails, or if server1 later stops responding, it will try to connect to server2. If server2 fails, conversd will try to connect to server3.

Once you’ve configured convers.conf, you can start the program with the simple command **conversd** – it will automatically go into the background. You should add it to the configuration file (like /etc/rc.d/rc.local) that contains your other AX.25 startup commands.

If you’d like to make the convers server available from the LinuxNode prompt, include a line like this in **node.conf** to create a CONV command:

```
Alias   CONVers   “telnet %{2:localhost} \
3600 \”/n %u\””
```

Now, node users will see a CONVers option from the menu, and entering it will connect them to the server.

You may also want to convers access via a specific call or alias (like the "convers mycall" command in NOS). Unfortunately, you can't just run the tpp-convers program from the ax25d.conf file; differences in end-of-line handling[19-3] will cause problems. Tomi Manninen, OH2BNS, wrote a convers client program that addresses this problem, and makes a couple of additional modifications to improve performance over the air. His **axconv** is available from tapr.org. It's easy to compile. Put the axconv.c.gz file in a convenient directory and issue the following commands as root:

```
gunzip axconv.c.gz
gcc -Wall -O6 -o axconv axconv.c
cp axconv /usr/sbin/axconv
```

This will compile the program and copy it to the /usr/sbin directory along with the other ax25 utility programs. The command to start axconv looks like this:

```
axconv [-c <channel>] [-n <name>] \
    [-p <paclen>] [-t <timeout>]<host>
```

The –c switch lets you define a starting channel; the –n switch sets the login name; the –p switch sets the maximum packet length used; and the –t switch sets an inactivity timeout (a setting of zero disables the timeout). The <host> is the convers server. You may also follow the host with the port number; 3600 is the default so you won't normally need to do that.

Adding appropriate lines in ax25d.conf will let you start axconv automatically:

```
#
[MVCONV VIA *]
NOCALL  * * * * * * L
default * * * * * * - root \
/usr/sbin/axconv axconv \
-p 256 -c 3694 -n %u localhost
```

[19-3] The fact that Unix uses just the linefeed character at the end of a line while MS-DOS systems use both the carriage return and linefeed characters causes lots of annoyance.

These lines establish a call – DAYCON – that will start the axconv program when anyone connects to that callsign on any AX.25 port. ax25d will launch axconv and have it connect to the convers server on the local machine, using a default convers channel of 3694. The "-n%u" parameter means the user will be logged in with their call (the "%u" is a variable that ax25d fills with the connecting station's callsign).

The tpp-convers package also includes a couple of other useful programs. **cstat** can be typed from the command line to show the status of the server's links to other sites, while **online** shows all connected users. Finally, the **convers** program is a client that allows you to connect locally to the server.

When you connect locally using the tpp **convers** program, your Linux login name will be used as your identifier to the convers network. If your login name isn't your call, you may be viewed as a non-ham intruder, so I suggest that you set up a user account named after your call and use that when you want to join the convers roundtable this way. If you connect to the server via telnet, you can avoid this problem by setting your login with the **/name** command (the commands you type are in bold):

```
mvfma $ telnet mvfma 3600
Trying 192.168.1.1...
Connected to w8apr.febo.com.
Escape character is '^]'.
/NAME N8UR
conversd @ Dayton Tampa PingPong-Release 1.14 - Type /HELP for help.
*** There is 1 user online
*** You created a new channel 0.
*** Welcome back, n8ur!
*** (10:56) conversd made you a channel operator for channel 0
```

Chapter

20

Conclusion

This has been a whirlwind tour of TCP/IP, but I hope you've found it helpful, and even more, I hope it has encouraged you to explore the new capabilities that either NOS or Linux/AX.25 offer. To learn the subtleties of NOS, you should do two things: read the reference manual for the version you're using, and *play with the programs*. Learning Linux is similar: get a good book on Linux basics, read the documentation for the AX.25 features and utilities, and experiment. Once you know the ins and outs, please share your knowledge with others. The ham radio TCP/IP community is still small, and we need all the Elmers we can get!

Appendix A - Resources for NOS and TCP/IP

Appendix B - AMPRNet Coordinators

AMPRNet IP address coordinators as of 10 October 1998
Corrections and updates to brian@ucsd.edu.

Note: the people listed here have volunteered to issue IP addresses for their areas. They are not paid to do this service; please understand that they are perfectly at ease to deal with coordination responses at a lower priority than the things that matter more, such as job and family. Please be patient when requesting an address.

Note subnet widths.

44.002 / 18	USA:Calif: Sacramento	K6RTV	Bob Meyer
44.004 / 18	USA:Calif: Si Valley - SFO	N6OYU	Douglas Thom
44.006 / 18	USA:Calif: Sta Barb / Ventura	WB5EKU	Don Jacob
44.008 / 18	USA:Calif: San Diego	WB6CYT	Brian Kantor
44.010 / 18	USA:Calif: Orange County	AA6TN	Terry Neal
44.012 / 18	USA:Eastern Washington,Idaho	KD7RO	Steven King
44.014 / 18	USA:Hawaii & Pacific Islands	WH6BH	Derek Young
44.016 / 18	USA:Calif: Los Angeles / Valley	WA6FWI	Jeff Angus
44.017 / 18	USA:Calif: Antelope / Kern County	KK6JQ	Dana Myers
44.018 / 18	USA:Calif: San Brdo & Riverside	KE6QH	Geoffrey Joy
44.020 / 18	USA:Colorado: Northeast	K0YUM	Fred Schneider
44.022 / 18	USA:Alaska	KL7JL	John Stannard
44.024 / 18	USA:Washington:Western / Puget	KD7NM	Bob Donnell
44.026 / 18	USA:Oregon	WA7TAS	Ron Henderson
44.028 / 18	USA:Texas: North	KC5ENU	Eric Martin
44.030 / 18	USA:New Mexico	AA5DF	Tim Baggett
44.032 / 18	USA:Colorado: Southeast	N3EUA	Bdale Garbee
44.034 / 18	USA:Tennesee	K9JA	Jeff Austen
44.036 / 18	USA:Georgia	N3AIA	Doug Reed
44.038 / 18	USA:South Carolina	KD4HTU	Dave Gosselin
44.040 / 18	USA:Utah	KA7OEI	Clint Turner
44.042 / 18	USA:Mississippi	KB5GGO	John Martin
44.044 / 18	USA:Massachusetts:western	KA1XN	Bob Wilson
44.046 / 18	USA:Missouri	N0EIR	Dave Salaman
44.048 / 18	USA:Indiana	K9DC	Dave Gingrich
44.050 / 18	USA:Iowa	KC0OX	Ron Breitwisch
44.052 / 18	USA:New Hampshire	K8LT	Gary Grebus
44.054 / 18	USA:Vermont	KD1R	Ralph Stetson
44.056 / 18	USA:Eastern&Central Mass	N1MGO	Gordon LaPoint
44.056.12 / 24	USA:Massachusetts: Worcester	N1MPY	Wayne McMahon
44.058 / 18	USA:West Virginia	KB8EHT	Tim Connolly
44.060 / 18	USA:Maryland	WB3FFV	Howard Leadmon
44.062.0 / 24	USA:Virginia-Central	K4ZIV	Russ Garber
44.062.32 / 24	USA:Virginia-Charlottsville	AC4ZQ	Mike Duvall

44.062.64 / 24	USA:Virginia-Eastern	WA4YSE	Lyman Byrd
44.062.128 / 24	USA:Virginia-Western	K4ZIV	Russ Garber
44.062.192 / 24	USA:Virginia-Northern	K4ZIV	Russ Garber
44.064 / 18	USA:New Jersey: northern	K2BJG	Robert Anderson
44.065 / 18	USA:New Jersey: southern	K2UT	Bob Applegate
44.066 / 18	USA:Delaware	NF3F	Butch Rollins
44.068.1 / 20	USA:New York: NY & LI	N2MDQ	Steve Dworkin
44.068.48 / 25	USA:New York: 30 Rock only	WA2NDV	Frank Garofalo
44.068.52 / 20	USA:New York: NY & LI	N2MDQ	Steve Dworkin
44.068.64 / 20	USA:New York: ENY	N2IGU	Bob Bellini
44.069 / 18	USA:New York: WNY	N2RJT	Dave Brown
44.070 / 18	USA:Ohio - oldnet	N8UR	John Ackermann
44.071 / 18	USA:Ohio - newnet	N8UR	John Ackermann
44.072 / 16	USA:Illinois	WA9AEK	Ken Stritzel
44.073 / 16	USA:Illinois	WA9AEK	Ken Stritzel
44.074 / 18	USA:North Carolina (east)	WA3JPY	Mark Bitterlich
44.075 / 18	USA:North Carolina (west)	WB4WOR	Charles Layno
44.076 / 18	USA:Texas: south	none	
44.077 / 18	USA:Texas: west	KA5EJX	Rod Huckabay
44.078 / 18	USA:Oklahoma	KB0QJ	J. Frank Fields
44.080 / 18	USA:Pennsylvania: eastern	WA3DSP	Doug Crompton
44.082 / 18	USA:Montana	WB7ETT	Don Heide
44.084 / 18	USA:Colorado: Western	K9MWM	Bob Ludtke
44.086 / 18	USA:Wyoming	WB7CJO	Reid Fletcher
44.088 / 18	USA:Connecticut	N1URO	Brian Rogers
44.090 / 18	USA:Nebraska	NF0N	Mike Nickolaus
44.092 / 18	USA:Wisconsin, up pen Michigan	N9UDL	Thomas Landmann
44.094 / 18	USA:Minnesota	N0QBJ	Bob Brose
44.096 / 18	USA:District of Columbia	N4YDP	Richard Cramer
44.098 / 18	USA:Florida	KO4KS	Brian A. Lantz
44.100 / 18	USA:Alabama	KE4NJB	Bruce Tenison
44.102 / 18	USA:Michigan (west lower pen)	N8WKM	Dan Thompson
44.102 / 18	USA:Michigan (east lower pen)	WB8TKL	Jay Nugent
44.104 / 18	USA:Rhode Island	W1CG	Charles Greene
44.106 / 18	USA:Kentucky	WA8FJK	Gregory A Cross
44.108 / 18	USA:Louisiana	N5KNX	James Dugal
44.110 / 18	USA:Arkansas	WD5B	Richard Duncan
44.112 / 18	USA:Pennsylvania: western	N3CVL	Bob Hoffman
44.114 / 18	USA:N&S Dakota	N7GXP	Steven Elwood
44.116 / 18	USA:Oregon:NW&PDX,Vancouver,WA	WS7S	Tom Kloos
44.118 / 18	USA:Maine	N1DXM	Carl Ingerson
44.120 / 18	USA:special use in Nevada	KI3V	Richard Hallman
44.122 / 18	USA:Kansas	K0HYD	Dale Puckett
44.123 / 18	USA:Virgin Islands	NP2W	Bernie McDonnell
44.124 / 18	USA:Arizona	KF7TP	Keith Justice
44.125.0 / 18	USA:Southern Nevada	KF7TI	Earl Petersen

44.125.128/18	USA:Northern Nevada	N8KHN	Bill Healy
44.126/18	USA:Puerto Rico	KP4TR	Ramon Gonzalez

#
44.128 is reserved for testing. Do not use for operational networks.
You may safely assume that any packets with 44.128 addresses are bogons
unless you are using them for some sort of testing
#

44.128/16	TEST		

#
International subnet coordinators by country
#

44.129/16	Japan	JF3LGC	Toshiyuki Mabuchi
44.129.192/24	Japan	JM1WBB	Isao SEKI
44.130/16	Germany	DL4TA	Ralf D Kloth
44.131/16	United Kingdom	G1PLT	Paul Taylor
44.132/16	Indonesia	YC1DAV	Onno W. Purbo
44.133/16	Spain	EA4DQX	Jose Antonio Garcia
44.134/16	Italy	I2KFX	
44.135/16	Canada	VE3JF	Barry McLarnon
44.136/16	Australia	VK2ZXQ	John Tanner
44.137/16	Netherlands	PE1CHL	Rob Janssen
44.138/16	Israel	4X1GP	Peleg Lapid
44.139/16	Finland	OH1MQK	Matti Aarnio
44.140/16	Sweden	SM0ORB	Anders Tornqvist
44.141/16	Norway	LA6KJ	Sven Astrup
44.142/16	Switzerland	HB9CAT	Marco Zollinger
44.143/16	Austria	OE1KDA	Krzysztof Dabrowski
44.144/16	Belgium	ON7LE	Eric Langhendries
44.145/16	Denmark	OZ1BVN	Soren B. Jensen
44.146/16	Phillipines	DU1UJ	Eddie Manolo
44.147/16	New Zealand	ZL1UFO	David Arnett
44.148/16	Ecuador	HC5K	Ted Jaramillo
44.149/16	Hong Kong	VS6YHJ	Thomason FAN
44.150/16	Slovenija	S53AD	Tomo Stegovec
44.151/16	France	F5BQP	Pierre-Francois Monet
44.152/16	Venezuela	YV5CIV	Pedro Jose Colina P.
44.153/16	Argentina (+Paraguay,Bolivia)	LU7ABF	Pedro Converso
44.154/16	Greece	SV1UY	Demetre Valaris
44.155/16	Ireland	EI9GL	Paul Healy
44.156/16	Hungary	HA8FN	Laszlo Fidrich
44.157/16	Chile	CE6EZB	Raul Burgos
44.158/16	Portugal	CT1CUM	Carlos Sousa
44.159/16	Thailand	HS1JC	Kunchit Charmaraman
44.160/16	South Africa	ZS6BLY	Wessel du Preez
44.161/16	Luxembourg	(none)	
44.162/16	Cyprus	5B4TX	C. Costis

44.163	Central America Coordinator General	YN1TV Theo Vlaar
	Secretariate	YN7DS Humberto A. Diaz S.
44.163.16 / 20	Panama	HP2CWB Jose Ng Lee
44.163.32 / 20	Costa Rica	TI2YO Minor Barrantas Fallas
44.163.48 / 20	Nicaragua	YN5JAR Jose Antonio Roman
44.163.64 / 20	Honduras	HR2JAE Jorge A. Escoto (San Pedro Sula)
		HR1BY Wolf Baron (Tegucigalpa)
44.163.80 / 20	El Salvador	YS1TG Mario Giolitti
44.163.96 / 20	Guatamala	TG9CL Carlos Eduardo Estrada
44.163.112 / 20	Belize	V31LO Tony Rath
44.163.128 / 20	Netherland Antilles	PJ2JW Joop Willems
44.164.0 / 22	Surinam	PZ2AC Otto Morroy
44.164.4 / 22	French Guiana	
44.164.8 / 22	Guyana	
44.164.12 / 22	Mozambique	C91BT / PA3CBH Theo Vlaar
44.164.128 / 22	Trinidad&Tobago	9Y4UWI Dr. Patrick Hosein
44.164.132 / 22	Falkland Islands	VP8CSA Mark
44.164.136 / 22	Aruba	P43T Anthony Thiel
44.165 / 16	Poland	SP5WCA Andrzej K. Brandt
44.166 / 16	Korea	HL1DKK LEE Jong-Jin
44.167 / 16	India	VU2LBW Lakshman ("Lucky") Bijanki
44.168 / 16	Taiwan	BV5AF Bolon
44.169 / 16	Nigeria	5N0OBA Kunle
44.170 / 16	Croatia	?? Sinisa Novosel
44.171.16 / 20	Colombia	HK3EGI Daniel E Visbal
44.171.32 / 20	Peru	OA4CZU Rafael H. Mantilla
44.171.48 / 20	Uruguay	CX7AP Dennis Cahill
44.172 / 16	Sri Lanka	4S7EF Ekendra
44.173 / 16	Mexico	XE2 / WP2B Regnerus Dantuma
44.174 / 16	Brazil	PY5JO Joao Fabio de Oliveira
44.175.0 / 20	Cuba	CO2JA Jose Amador
44.175.16 / 20	Dominican Republic	HI8GN Jose Ramon
44.175.32 / 20	Haiti	HH2B Bernard Russo
44.176 / 16	Turkey	TA2T A. Tahir DENGIZ
44.177 / 16	Czech Republic	OK2XDP Michal Dobes
44.178 / 16	Russia	RA3APW Karen Tadewosyan
44.179.0 / 20	Gibraltar	ZB0D Jim Watt
44.179.32 / 20	Malta / Gozo	G0DEO / 9H1IA William Batey
44.180 / 16	Yugoslavia(former)	YT7MPB Miroslav Skoric

44.181 / 16	Slowak Republic	OM3WKW	Branislav Chvila
44.182 / 16	Romania	YO2LGU	Norbert Hanigovszki
44.183 / 16	Iceland	TF3BNT	Benedikt Sveinsson
44.184 / 16	Lebanon	OD5NZ	Ramzi Abdallah
44.185 / 16	Bulgaria	LZ1NY	Victor ?
44.186 / 16	Singapore	9V1ZY	Edwin Teh
44.187.0 / 20	Lithuania	LY2IC	Vytas Matonis
44.187.16 / 20	San Marino	T77IG	Don Pino
44.188.0 / 20	Armenia	?	Edgar Der-Danieliantz
44.188.16 / 20	Azerbaijan	none	
44.188.32 / 20	Belarus	none	
44.188.48 / 20	Estonia	ES1LAU	Anto Veldre
44.188.64 / 20	Georgia	none	
44.188.80 / 20	Kazakhstan	none	
44.188.96 / 20	Kyrgyzstan	none	
44.188.112 / 20	Latvia	YL2PG	Gunnars E. Postnieks
44.188.128 / 20	none	none	
44.188.144 / 20	Moldova	none	
44.188.160 / 20	Tajikstan	none	
44.188.176 / 20	Turkmenistan	none	
44.188.192 / 20	Ukraine-Kiev	UT2UZ	Nick Fedoseev
44.188.208 / 20	Ukraine-Donetsk	UR7IEK	Jim Smelyansky
44.188.224 / 20	Ukraine-Lviv	UT1WPR	Vic Golutvin
44.188.240 / 20	Uzbekistan	none	
44.189.0 / 20	Bosnia & Herzegovinia	T97S	Sead Sogoljevic
44.193	Outer Space-AMSAT	W3IWI	Tom Clark
44.194	Oceana		(none yet)
44.195	Antarctica (all treaty zones)	KC4AAA	Brent Jones
44.196	Arctic	(none yet)	
44.197	African Continent		
44.197.16 / 20	Ivory Coast	?	J.V. Mayega
44.197.64 / 20	Central	(none yet)	
44.198		Pacific Islands	
44.198.0 / 22	Guam	KH2EI	Phil Weber

Appendix C - Sample AUTOEXEC.NOS

Appendix D - NET/ROM.NOS

Appendix E - BBS.NOS

Appendix F - DOMAIN.TXT

Appendix G - FTPUSERS

Appendix H - REWRITE

Appendix I - FORWARD.BBS

Appendix J - BM.RC

Appendix K - Configuring Linux to TNOS