

XNET

—

AN X-WINDOWS BASED AX.25 PACKET NETWORK ANALYZER

BY

RICHARD PARRY

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science
North Central College

Approved: _____

Naperville, Illinois
Wednesday, May 24, 1995

ACKNOWLEDGEMENTS

John Ousterhout, thanks for Tcl/Tk

Mark Diekhans, thanks for the addinput patch

Don Libes, thanks for the TERM example

Mark Sproul, thanks for the inspiration

Josh Stein, thanks for the idea

Michael Tortorella, thanks for the start

Kim Tracy, thanks for the supervision

To my family, thanks...

ABSTRACT

—

This Master's project paper describes the development of XNET, a packet network analyzer which collects and displays network information in an *X-Windows* environment. XNET is written in the Tcl/Tk language, a language optimized for graphical user interface development. The hardware for the project consists of a radio transceiver and a Terminal Node Controller (TNC) which connects to the computer system through a serial port.

XNET analyzes AX.25 wireless packet networks which predominately operate in the two meter band (144 to 148 MHz). The AX.25 data link layer protocol is a variant of the international standard X.25 protocol. The protocol has been slightly modified to meet the unique requirements of amateur radio packet communication. Unlike wire and optical fiber based networks where the network configuration is relatively stable, wireless networks continually change as nodes are switched on and off. Wireless networks are also more error prone due to the variability of the medium. It is this variability and randomness that makes a network analyzer, such as XNET, so important to the understanding of the network.

The paper includes an introduction to the Tcl/Tk language and a detailed description of the AX.25 protocol used in wireless packet networks. Also included is an extensive section devoted to the details of the XNET software sufficient for understanding the many design decisions that were made during the development of the program. A description of the hardware used for the system, a complete Operating Manual, and all source code are included as appendices.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	ii
ABSTRACT	iii
TABLE OF CONTENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES.....	viii
LIST OF ABBREVIATIONS	x
1. INTRODUCTION.....	1
1.1. PHYSICAL LAYER.....	2
1.2. THE ROLE OF A NETWORK ANALYZER	3
2. THE DESIGN	4
2.1. REQUIREMENTS	4
2.1.1. HARDWARE AVAILABILITY	5
2.1.2. LANGUAGE SELECTION.....	5
2.1.3. USE GUI	5
2.1.4. DESIGN FOR PORTABILITY	6
2.1.5. DESIGN FOR CONFIGURABILITY	6
2.1.6. DISPLAY NODES AND STATISTICS.....	6
2.1.7. DISPLAY IN TABULAR AND GRAPHICAL FORMAT.....	7
2.1.8. DISPLAY RAW PACKETS.....	7
2.1.9. TIME STAMP FOR AGING	8
3. THE TCL/TK LANGUAGE.....	8
3.1. ADVANTAGES.....	9
3.2. HELLO WORLD	10
3.3. LANGUAGE EXTENSION (addinput).....	11
3.4. DATA TYPES AND ARRAYS.....	12
3.5. WIDGETS.....	12
3.6. COMMENTS	13
4. XNET SOFTWARE.....	15
4.1. xNet — Main Program Module	17
4.1.1. tick — XNET System Clock.....	18
4.1.2. open_input — System Initialization.....	18
4.1.3. file_input — Get Packet Module	19
4.1.3.1. node_buf — Initial Packet Buffer	19
4.1.3.2. NODE FILTERING	20
4.1.3.3. HEARD AND UNHEARD NODES.....	21
4.1.3.4. DATA FLOW	23
4.1.3.4. TIMING	26
4.1.4. node_arr — Permanent Array for Active Nodes.....	27
4.1.5. conn_array — Permanent Array for Connected Nodes.....	29
4.1.6. update_nodes — Update Active Nodes.....	30
4.1.7. update_conn — Update Connection Nodes	32

4.1.8. close_input — STOP XNET	33
4.2. xMap.tcl — Script to Display Map Window	33
4.2.1. clear_map_data — Clear Map.....	35
4.2.2. map_insert — Draw Map.....	35
4.2.2.1. CONNECTION DEFINITION	35
4.3. xNodes.tcl — Script to Display Active Nodes.....	36
4.3.1. make_buttons — Create Buttons.....	37
4.3.2. clear_buttons — Clear Buttons	38
4.3.3. nodes_insert — Display Node Statistics	38
4.4. xTerm.tcl — Create Display and Print Raw Packets	38
4.4.1. term_init — Initialize Terminal	38
4.4.2. term_clear — Clear Terminal.....	39
4.4.3. term_down — Move Cursor Down.....	40
4.4.4. term_insert — Insert the Text in Window	40
4.4.5. term_update_cursor — Move Cursor.....	40
4.5. xGraf.tcl — Script to Display Graphs.....	40
4.5.1. graf1 — Create 1 Hour Graph.....	41
4.5.2. graf5 — Create 5 Hour Graph.....	43
4.5.3. graf25 — Create 25 Hour Graph.....	43
4.5.4. graf_it — Draw the Graph.....	43
4.5.5. clear_graf_data — Clear Window.....	45
4.6. xSimul.tcl — Script to Create Simulation Window.....	46
4.7. xPort.tcl — Script to Create Port Selection Window.....	48
4.8. xPrefs.tcl — Script to Create Preferences Window	49
4.9. simX.sh — UNIX Simulation Script	51
4.10. prefs — Preferences File	52
5. THE HARDWARE.....	52
5.1. TNC OUTPUT	54
6. PROJECT TESTING, RESULTS AND EXTENSIONS.....	55
6.1 VARIANCES	55
6.2 TESTING	56
6.3 XNET USE.....	57
6.4 PROJECT EXTENSIONS	57
6.4.1. TCP/IP VERSION OF XNET.....	57
6.4.2. EXAMINE RAW BIT LEVEL DATA.....	57
7. CONCLUSION	58
8. ANNOTATED BIBLIOGRAPHY AND REFERENCES.....	59
9. TRADEMARKS	62
APPENDIX A — AX.25 PROTOCOL	63
A.1. INTRODUCTION.....	64
A.2. ASYNCHRONOUS TRANSMISSIONS	67
A.3. SYNCHRONOUS TRANSMISSIONS	68

A.3.1. CHARACTER SYNCHRONIZATION.....	68
A.3.2. COUNT SYNCHRONIZATION.....	69
A.3.3. BIT SYNCHRONIZATION.....	69
A.4. BIT STUFFING.....	69
A.5. LINK LAYER.....	70
A.6. INFORMATION FRAME.....	71
A.7. SUPERVISORY FRAME.....	73
A.8. UNNUMBERED FRAME.....	74
A.9. RELIABILITY AND THE FCS FIELD.....	75

APPENDIX B — TERMINAL NODE CONTROLLER	77
B.1. THE PURPOSE OF THE TNC	78
B.1. ASYNCHRONOUS PORT PARAMETERS	78
B.2. SPECIAL CHARACTERS	79
B.3. IDENTIFICATION PARAMETERS	80
B.4. LINK PARAMETERS	81
B.5. MONITOR PARAMETERS	81
B.6. TIMING PARAMETERS	82
APPENDIX C — XNET OPERATING MANUAL	84
APPENDIX D — XNET SOURCE CODE	111
D.1. xNet	114
D.2. xMap.tcl	137
D.3. xNodes.tcl	142
D.4. xTerm.tcl	146
D.5. xGraf.tcl	152
D.6. xSimul.tcl	159
D.7. xPort.tcl	164
D.8. xPrefs.tcl	172
D.9. sim1.sh	179
D.10. prefs	181

LIST OF TABLES

—

Table 2.1-1. High Level XNET Requirements.....	4
Table 4.1.3.2-1. Filtered Node Names	20
Table 4.1.3.2-1. Frame Types	21
Table A.3.1-1. Synchronous Character-oriented Control Characters (ASCII)	68
Table A.6-1. Information Frame Format.....	71
Table A.6-2. Actual Sample Information Frame less Information Field	72
Table A.7-1. Supervisory Frame Format	73
Table A.8-1. Unnumbered Frame Format.....	74
Table B.1-1. Asynchronous Port Parameters	79
Table B.2-1. Special Characters.....	80
Table B.3-1. Identification Parameters.....	80
Table B.4-1. Link Parameters	81
Table B.5-1. Monitor Parameters.....	82
Table B.6-1. Timing Parameters	83

LIST OF FIGURES

—

Figure 1.1-1. Typical Packet Network	3
Figure 3.2-1. Hello World!— Example Simplified Tcl Script.....	10
Figure 3.2-2. Hello World! — Extended Tcl Example Script.....	11
Figure 4-1. XNET System Distribution — 19 Scripts/Files	16
Figure 4.1-1. XNET Main Window	17
Figure 4.1.3-1. node_buf — XNET Node Buffer Data Structure	19
Figure 4.1.3-2. Flow Diagram of Packet from Network, to Array, to Window.....	24
Figure 4.1.3.3-1. Digipeating Example	22
Figure 4.1.3.4-1. High Level XNET Events	27
Figure 4.1.4-1. The Node Array (node_arr).....	28
Figure 4.1.5-1. The Connection Array (conn_arr)	29
Figure 4.2-1. MAP Window.....	34
Figure 4.3-1. NODES Window	37
Figure 4.4-1. TERM Window	39
Figure 4.5.3-1. GRAPH Window — 25 Hour Version.....	42
Figure 4.5.4-1. Circular Buffer.....	44
Figure 4.5.4-2. 1 Hour Buffer	44
Figure 4.5.4-3. 5 Hour Buffer	45
Figure 4.5.4-4. 25 Hour Buffer	45
Figure 4.5.4-5. Circular Buffer.....	45
Figure 4.6-1. SIMUL Window.....	47
Figure 4.6-2. SIMUL Window (Warning Dialog Box).....	48
Figure 4.7-1. PORT Window	49

Figure 4.8-1. PREFS Window	50
Figure 4.8-2. xPrefs.tcl (Information Dialog Box).....	50
Figure 5-1. RF Transceiver and TNC.....	53
Figure 5.-2. XNET Development System	54
Figure 5.1-1. Sample TNC Output.....	54
Figure A.1-1. The Layers of AX.25.....	65
Figure A.1-2. Unbalanced Network Configuration.....	66
Figure A.1-3. A Typical Packet Point to Point Network	67
Figure A.1-3. Balanced Network Configuration.....	67
Figure B.1-1. Typical AX.25 Radio Point to Point Network.....	78

LIST OF ABBREVIATIONS

—

AM	Amplitude Modulation
AMRAD	Amateur Radio Research and Development Corporation
AMTOR	Amateur Teleprinting Over Radio
ASCII	American Standard Code for Information Interchange
CCITT	International Telegraph and Telephone Consultative Committee
DTE	Data Terminal Equipment
EBCDIC	Extended Binary Coded Decimal Interchange Code
FCS.....	Frame Check Sum
FM.....	Frequency Modulation
GUI.....	Graphical User Interface
HDLC.....	High Level Data Link Control
IP	Internet Protocol
ISO	International Standards Organization
LAPB.....	Link Access Procedure Balanced
LAPD	Link Access Procedure D-channel
Modem	Modulator / Demodulator
OSI	Open System Interconnections
PACTOR.....	Packet Amateur Teleprinting Over Radio
PAD.....	Packet Assembler Disassembler
RF.....	Radio Frequency
RTTY	Radio Teletype
SABM	Set Asynchronous Balanced Mode
SDLC.....	Synchronous Data Link Control

SQL Standard Query Language
SSB..... Single Sideband
Tcl..... Tool Command Language
TCP Transmission Control Protocol
Tk Toolkit
TNC..... Terminal Node Controller
TQM..... Total Quality Management
VADCG..... Vancouver (BC) Amateur Digital Communications Group
XNET The name given to the program and project as a whole
xNet..... A specific file of the XNET program

1. INTRODUCTION

For nearly 100 years amateur radio operators around the world have been responsible for numerous developments in wireless communications. Amateur radio operators communicate with each other throughout the world using radio frequency (RF) equipment. Communication can be as short as a few miles or as distant as global communication using satellite, moon bounce, and terrestrial RF. The modulation modes range from simple radiotelegraphy (Morse Code), to voice using amplitude modulation (AM), frequency modulation (FM), and single sideband (SSB). More recently digital packet radio communication has become very popular. It is this last mode of communication that forms the basis for this project.

In the early 1950s, digital communication by way of radioteletype became popular. With the advent of the microcomputer in the early 1970s and the subsequent reduction in both price and size of computers, which allowed more sophisticated methods of digital communication, amateur radio operators developed more sophisticated methods of digital communication. Modulation techniques for communication expanded from simple radioteletype (RTTY), to more complex modulation schemes including: Amateur Teleprinting Over Radio (AMTOR), packet radio (PACKET), and Packet Amateur Teleprinting Over Radio (PACTOR). All of these digital communication techniques and protocols were developed specifically to meet the unique requirements of wireless communication.

In the pursuit of speed and reliability, amateur radio operators have embraced computer technology to develop more advanced forms of communication. We will concentrate on AX.25 packet radio networks and XNET, a network analyzer used to monitor and display the behavior of these networks.

1.1. PHYSICAL LAYER

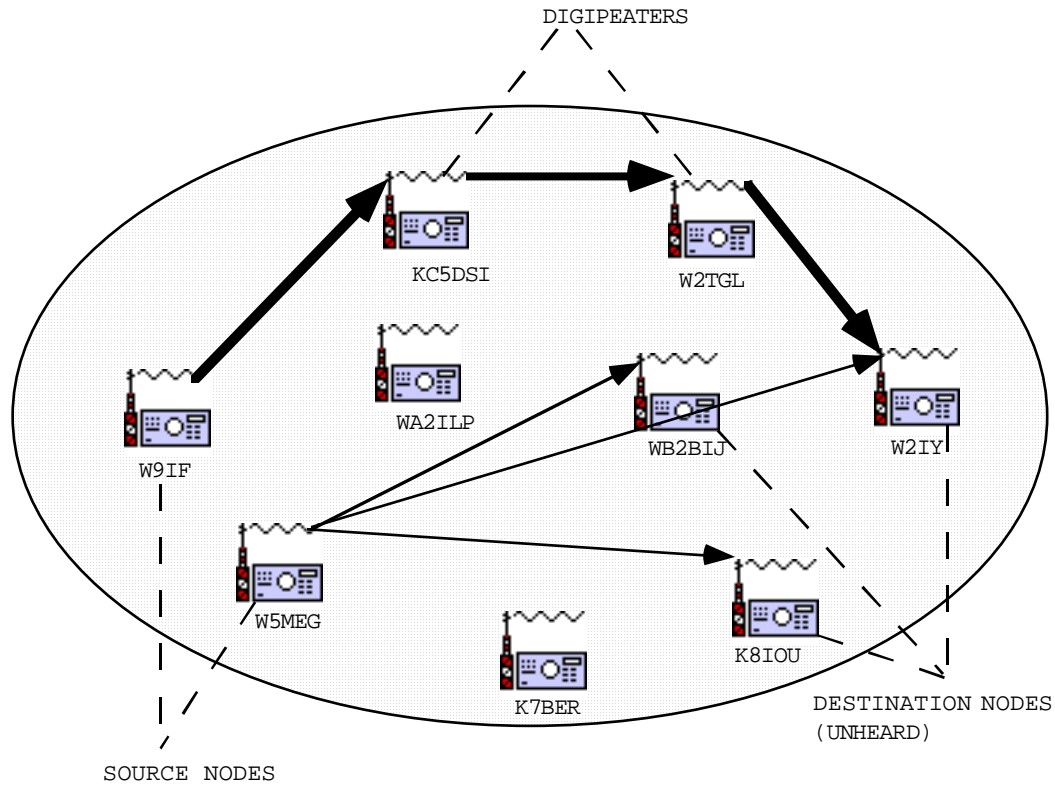
Most networks are composed of electrical copper conductors. However, with the need for greater bandwidth and the reduction in the price of optical fiber, fiber is becoming a superior alternative to copper. These media have one thing in common, they are all *hardwired*. There is a physical cable connecting the nodes. Wireless communication is a vastly different manner of linking the nodes in a network. When copper is used at the physical layer, specific choices include: open two-wire lines, shielded and unshielded twisted pair, and coaxial cables. In a similar manner, when RF is used choices include: satellite, terrestrial microwave, and radio. All of these are used by amateur packet radio operators at the physical layer for communication.

Radio frequency as a communications medium is fairly unreliable. For example, communication at low frequencies (1 MHz to 50 MHz) varies over the course of the day (typically frequencies in this range are unusable at night due to the loss of the ionosphere which is created by the sun during the day). The medium is also affected by an 11 year sun-spot cycle and man-made interference (i.e., motors, toasters and hairdryers). While amateur radio operators make many long distance (around the world) digital connections, most reliable operation and activity occurs above 50 MHz and is restricted to line of sight communication. An explanation of why frequency affects the distance of communication is outside the scope of this paper.

In Figure 1.1-1, several nodes are shown. We use the term *node* to indicate a connection point. While world-wide communication is possible, most communication is done on a local basis (i.e., 30 mile radius)¹, therefore the diagram may be pictured as a metropolitan area. The solid lines represent packet traffic from a source node to a destination node. The thick solid lines shown at the top of the figure show how the

¹ The kind and height of the antenna of the node plays a major role in determining distance. The 30 mile radius will be used in this paper as a typical distance for a wireless AX.25 network.

distance may be extended in a wireless network. Here the nodes, KC5DSI and W2TGL are being used as repeaters. We will refer to this figure later to explain some of the characteristics of an AX.25 packet network including the concepts of *heard* and *unheard* stations, and *digipeating*.



1.2. THE ROLE OF A NETWORK ANALYZER

When multiple nodes are connected to each other we have a network. It is not hard to imagine that as the number of nodes increases and the traffic increases, networks can become overloaded and fail. Even in networks that appear to be functioning, there remains a need to monitor the network to assure that it is working properly. Before a network is constructed, it should be modeled to ascertain that the design requirements will be met. However, even when modeling is performed, the network should be analyzed in the real world to confirm design goals and to understand how the network

may be improved. A network analyzer is a tool that performs this important function. XNET is such a tool, and has been designed to monitor amateur radio packet networks using the AX.25 data link layer protocol.

2. THE DESIGN

In this section emphasis is placed on XNET design requirements and goals. It is important to note, however, that many of the requirements and features that went into the design of XNET are best covered in detail in other sections. For example, the attached Operating Manual forms an integral part of the documentation and the rationale of some design goals is best explained there. The software section of this paper also provides an explanation of the less straightforward algorithms and the justification for their use. Therefore only those aspects of the design that are not covered elsewhere are included in this section.

2.1. REQUIREMENTS

A good design, whether it be a computer software project such as XNET, or building a bridge, begins with requirements. For a large project, the requirements are often so extensive that a separate *Requirements Document* is created. For our present needs, Table 2.1-1 will suffice to show the high level requirements.

- Use hardware that is readily available
- Select language that best meets requirements
- Develop all displays using GUI
- Develop system for portability
- Allow system to be configured by user
- Display all nodes on the network with key statistics
- Display nodes/data in tabular and/or pictorial form
- Display raw packet data

- Time stamp and age packets

2.1.1. HARDWARE AVAILABILITY

In addition to the computer system, only two hardware components are required: a radio frequency receiver and a TNC. These items are a necessary part of a packet radio system and are therefore readily available to the intended user of XNET. Section 5 will discuss the hardware used in the development of XNET in greater detail. The TNC is a critical and complex component of the system. It is discussed in Appendix B.

2.1.2. LANGUAGE SELECTION

Selecting a language was an easy process. Once the requirement for a language designed for GUI development was met, and the limited resources for the XNET project were understood, the number of languages was vastly reduced. When the additional requirement (specified by the author) to select a language used in the engineering, scientific, and UNIX® communities was added, the possibilities were further reduced. The remaining choices were limited to Tcl/Tk and C. When the C programmer avails himself or herself to MOTIF, and the X windowing system to develop a GUI, it is a large effort and adds unnecessary complexity to the development of a software project such as XNET. Therefore Tcl/Tk was chosen. Further support for this decision is provided in section 3 which provides a summary of the strengths of the language for GUI development.

2.1.3. USE GUI

There is little doubt that GUIs are here to stay. In fact, the popularity of the Apple® MacOS® and Windows® operating systems is positive proof. Users demand, as they should, interfaces that address ergonomic requirements as well as technical requirements. However, the casual user of a word processor or spreadsheet (the author included until the

development of XNET) gives little thought to the significant work and time that goes into the development of a well designed GUI. Attention to detail is the hallmark of a well designed GUI. In the development of XNET, much effort and time was given to the size of the windows, color, fonts, re-sizeability, placement, and myriad variables that go into the development of a GUI.

2.1.4. DESIGN FOR PORTABILITY

There are 632,000² amateur radio operators in the United States of which a significant portion use packet radio for communication. The intent was to develop a software tool that could be used by many packet radio users. XNET requires Tcl/Tk. Although not a mainstream programming language like BASIC, PASCAL, or C, it is nonetheless available on many platforms including the ubiquitous IBM® PC. In fact, XNET was developed entirely on a IBM PC clone under the LINUX operating system, a variant of UNIX. It is important to note that both Tcl/Tk and LINUX are in the public domain available free over the Internet.

In addition, since Tcl/Tk is an interpreted language, the user does not have to compile the program for execution, which can be an additional barrier to some users.

2.1.5. DESIGN FOR CONFIGURABILITY

The requirement for XNET to allow the user to configure it is provided by three preferences windows. They allow the user to specify key parameters for the particular system and the needs of the user. The three windows: SIMUL, PORT, and PREFS are discussed in more detail in the software section and the attached Operating Manual.

2.1.6. DISPLAY NODES AND STATISTICS

The requirement to display nodes is met by the NODES window. This window is described in much greater detail in subsequent sections and in the Operating Manual.

² "How Many Hams?," QST magazine, February 1995, page 89.

However, as the reader shall see, the NODES window meets the requirement by displaying in tabular form all active nodes on the network. The age of the last packet is also provided along with other key information. The window may be re-sized which allows the user to develop an uncluttered desktop by shrinking the display or expanding the display to show information of particular interest.

2.1.7. DISPLAY IN TABULAR AND GRAPHICAL FORMAT

There are a total of seven windows available to the user. Each window is independent of the other windows. This independence of the windows played a major role in the design structure of the software. As will be explained in the software section, the seven windows are loaded and executed from seven separate source files. This design decision eased development and will aid in the maintenance of the program.

The independence of the windows also has the advantage of allowing each of the windows to be specifically designed to meet the requirements of that window display and to customize the display accordingly. For example, three of the windows are devoted to setting preferences; they allow the user to customize XNET. *Radiobuttons* are ideally suited for selection from a limited number of items and they are used extensively in these windows. The remaining four windows range from the graphical MAP window to the TERM window used to display raw Terminal Node Controller (TNC) data to the GRAPH and NODES windows. All windows are created to meet the intent of the display and the lack of interaction between windows mitigates compromising technical requirements of one window with that of another.

2.1.8. DISPLAY RAW PACKETS

Raw packet data is displayed in the TERM window. This window displays all data received by XNET from the TNC with the exception that the *trailer* of the packet may be truncated to fit the window. This provides best readability while still meeting the purpose of the window. More specifically, of interest to the person responsible for the network is

the *header* which contains the source, digipeater, destination nodes, and the frame ID. The actual information that is embedded in the packet is of no interest and truncating some of this information is not relevant to the purpose of XNET.

2.1.9. TIME STAMP FOR AGING

XNET time stamps all packets as they are received. This is important since this information is used to *age* the packet. The resolution of the time stamp is one second. This decision was based on the characteristics of an AX.25 network. Unlike other network protocols, where thousands or tens of thousands of packets are transmitted every second, packet radio frames rarely occur more than once per second on average.

3. THE TCL/TK LANGUAGE

XNET is written using Tcl, an abbreviation for Tool Command Language (pronounced "tickle"). A companion to Tcl is Tk, a contraction for *toolkit* (pronounced "tee kay"). Tk is an X11 toolkit based on Tcl. Tcl/Tk is a unique development platform, at least as compared to traditional computer languages such as PASCAL, C, COBOL, and others. A little background is relevant and will aid the reader in understanding the advantages of the language and show how XNET uses the language's features. A second purpose for this section is to explain some of the advantages and disadvantages of the language in general, and specifically how they relate to the development of XNET. However, before addressing the technical aspects, it is relevant to understand the reason for the development of yet another computer language.

Tcl was developed from a need to build a reusable command language. Work began on development of Tcl in 1988 and the first public release occurred in 1989. The language was distributed primarily by way of the Internet along with source and documentation. It was not until 1994 that the first book formally documenting the language was published. However, at this writing at least one other book is planned for

publication in 1995. As the author of the language, John Ousterhout, states, "*Tcl and Tk have succeeded beyond my wildest dreams.*"³

Originally a Tcl application was intended to require some C code for it to be fully implemented. However, the Tcl/Tk application *wish*, made it possible to write an entire application without any C code. XNET is an example. It uses *wish* and therefore did not require any C code.

3.1. ADVANTAGES

There are numerous advantages to using Tcl/Tk for application development. *Tcl* is a scripted, interpreted language much like PERL. It provides an excellent environment for rapid development. An entire graphical user interface (GUI) application can be written in simple scripts using the windowing shell, *wish*. This provides a means for developing the application at a much higher level than traditional C or C++ methods. In addition, since the language was intended specifically for GUI development, powerful single line commands perform many functions automatically as we shall see in the next section. Since it is an interpreted language, it allows the user to create and execute new scripts *on the fly* without the need to recompile or restart the application. This is an important advantage that may be lost to the casual reader. GUI development to a large extent deals with ergonomic and human interface issues. Thus trying a new background color or font, which are important ergonomic factors, can be implemented within seconds. Compiling a large program to ascertain if a new color is more pleasing to the eye may be such a cumbersome process in another language that alternatives may not be pursued.

The downside to an interpreted language, such as Tcl, is that they are often slow. However, workstations upon which Tcl runs have in most cases sufficient horsepower to make speed concerns unimportant. If performance is an issue, the critical portion of the code can be written in C.

³ Ousterhout, John K, "*Tcl and the Tk Toolkit*," Addison-Wesley, Reading, Massachusetts, 1994, page xviii.

The language also allows a huge reduction in the amount of code needed to write an application. Ratios of ten to one are not uncommon when comparing C code with Tcl/Tk code. This fact has the very real advantage of reducing cost by reducing development time.

3.2. HELLO WORLD

A small code segment is provided below showing the famous “Hello, world!” program as it would be written as a Tcl script. As we shall soon see, if the user does not specify all parameters that go into the creation of a window, *Tcl* will open the window using default parameters (i.e., font and font size; window name, size, position, foreground and background color, etc.).

Every Tcl statement can be thought of as a function call, in the form "cmd arg arg arg." The *Tcl* script for our *Hello world!* example consists of two statements. The first statement is the *label* command that declares a label with name *.l* followed by the text for the label. Nothing is displayed until the *packer* is called with the *pack* statement. The results of the code are shown in Figure 3.2-1.

```
label .l -text "Hello world!"  
pack .l
```

There are many default parameters that were assumed by Tcl in creating the window. Below is a more complete description of the simple *label* command. It now includes the foreground and background colors of the label, font type for the text, and a few other parameters that specify the positioning of the label. Although each of the additional parameters are displayed on a separate line for pedagogical reasons, they are actually considered a single line (the backslash character is used to indicate continuation of the

command on a new line). The window created by the script is shown in Figure 3.2-2. The astute reader will notice the word *hello2* displayed in the frame of the widget. This is merely the name of the script file used when the program was written.

```
label .l \  
-text "Hello world!" \  
-relief raised \  
-bd 5 \  
-fg white \  
-bg brown \  
-anchor w \  
-font -Adobe-helvetica-bold-r-normal--*-140*  
  
pack .l \  
-expand yes \  
-fill both \  
-padx 5m \  
-pady 5m \  
-ipadx 10m \  
-ipady 10m
```

3.3. LANGUAGE EXTENSION (*addinput*)

XNET was developed using Tcl version 3.6 and Tk version 7.3. This version of Tk does not support a mechanism to accept input from a background child process. This is an important requirement since XNET must be able to perform normal foreground functions (i.e., clock heartbeat and packet aging) without polling the serial port for data. Fortunately, the *addinput* patch overcomes this deficiency. As of this writing, the next version of Tcl, will be version 4.0. It is currently in beta testing and will incorporate this

feature in the *fileevent* command. XNET will need to be updated in order to be compatible with this new version of Tcl.

3.4. DATA TYPES AND ARRAYS

Despite the power and flexibility of Tcl, there are two characteristics of the language that may be considered limitations. First, Tcl supports only single dimensional arrays. However since it supports *associative* arrays, pseudo multi-dimensional arrays can be implemented. For example, the following two pseudo code statements are not equivalent as they might be in another language.

```
set arr("1,1") 47
set arr("1, 1") 47
```

The space in this case is interpreted as a legitimate character making the index pointer "1,1" different from "1, 1" (note space between digits).

Even when the quotation marks are removed, the following are not equivalent. In fact, the second line gives a syntax error since the blank space is used to delineate an argument to the *set* command.

```
set arr(1,1) 47
set arr(1, 1) 47
```

A second possible weakness of the language is the lack of data types. Tcl supports only *string* data types. Although unusual for most general purpose languages, this is not completely uncommon, since PERL also only supports strings. However, in most cases this characteristic should not have serious consequences outside of perhaps speed and memory usage.

3.5. WIDGETS

The basic building block of a GUI language such as Tcl/Tk, is the *widget*. The term *widget* and *window* are used synonymously in Tk. A widget, like a GUI window, has a

particular appearance and behavior, sometimes called its *look and feel*. Widgets can be divided into classes; examples are: buttons, menus, and scrollbars.

Other widgets include *text* and *canvas* widgets. XNET makes extensive use of the canvas widget to display data. For example, the TERM window which displays raw packet text strings uses a text canvas on which to display the information. The MAP window is an XNET display that uses a canvas widget on which to place data and draw lines.

3.6. COMMENTS

Like all languages, Tcl/Tk allows the user to insert comments for documentation. If the first non-blank line is a pound sign (#), then the remainder of the line is considered a comment as shown below.

```
# this is a comment!
```

The Tcl comment syntax does not allow one to easily remove a portion of code for testing. During development of a program, a software engineer often wishes to *comment out* a portion of a program in a simple manner such as the C language provides.

In addition, since the first character must be a pound sign, it does not create an environment conducive to encouraging the programmer to write multi-line documentation, since adding or removing a group of words is cumbersome.

We have stated that Tcl/Tk is an interpreted language and that it may be slow. This directly affects comments or perhaps more accurately the placement and length of comments. The following three example show how comments affect system performance.

```
proc comment1 {} {  
}  
  
proc comment2 {} {  
    # this is an example comment line  
}
```

```
proc comment3 {} {  
    # this is an example comment line 1  
    # this is an example comment line 2  
    # this is an example comment line 3  
    # this is an example comment line 4  
    # this is an example comment line 5  
    # this is an example comment line 6  
    # this is an example comment line 7  
    # this is an example comment line 8  
    # this is an example comment line 9  
    # this is an example comment line 10  
}
```

The execution times for procedures comment1, 2 and 3 are 22, 27, and 120 microseconds respectively. The first procedure, comment1, is a null procedure that performs no function and represents the baseline for the comparisons. Procedure, comment2, adds 23% overhead to the execution of the procedure. The last procedure adds nearly 550% to the execution of the procedure.

This characteristic of the language has the very real world implication of placing a burden on the software engineer to carefully place comments. Specifically, in time critical procedures, comments should be carefully placed. The example below shows a *for* loop. The comment should be placed outside the repetitive loop as shown in the second code segment, rather than inside.

```
for {set i 0} {$i < 100} {incr i} {  
    # this procedure adds numbers  
    incr sum $i  
}  
  
# this procedure adds numbers  
for {set i 0} {$i < 100} {incr i} {  
    incr sum $i  
}
```

While on the subject of execution times, it is worthwhile to acknowledge that Tcl/Tk supports a *time* command which allows the programmer to easily ascertain execution time. This command was used to compute execution times in the above *comment* examples. Specifically, the code segment below will call procedure *comment2* 10,000 times and return the average iteration time. The procedure is performed many times (i.e.,

10,000) to remove inaccuracies that are inherent in the resolution of the system clock in multi-user, multi-tasking systems.

```
time comment2 10000
27 microseconds per iteration
```

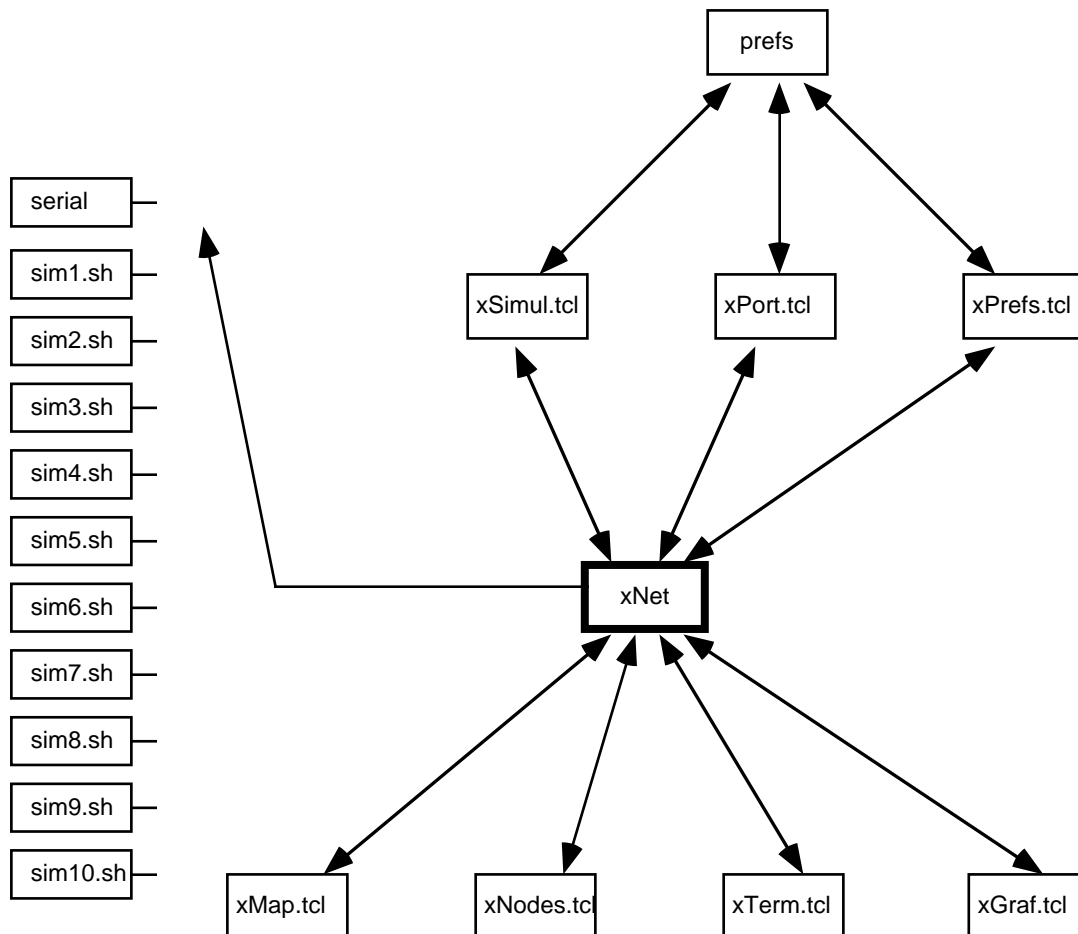
4. XNET SOFTWARE

Approximately 600-700 man hours were expended in XNET development. The raw source code contains over 3,000 lines. No tools were available to ascertain un-commented lines of code such as *ncsl*, however, 2,000 lines is a reasonable estimate. An additional word of caution is worthwhile when interpreting lines of code. Just as a single line of assembler cannot be compared with a single line of a high level language such as PASCAL or COBOL, so too, a single line of Tcl/Tk is not at all like a single line of most other languages. The closest comparison is a database language such as the Standard Query Language (SQL), where a few lines may perform such numerous and complex tasks as: opening a file, sorting the file, searching for a particular occurrence, generating a report, and then closing the file. As we have shown in the *Hello World* example, in two lines of code, Tcl/Tk performs an incredible number of tasks. With that understood, it is probably more reasonable to ask how many *statements* make up the program. Again, no software tools are available to automatically generate this, however, it is approximately 600-800 statements. Comparing the number of hours with the number of statements shows that XNET supports the rule-of-thumb that *each line of tested, documented code requires 1 hour to generate*.

The remainder of this section contains details of the workings of the XNET program. However, a complete understanding of the program is not possible without a firm grasp of the entire system. Therefore in this section, the subject matter is restricted to purely software development and maintenance items. This section was written with the assumption that the reader has sufficient background in GUIs and networks and will refer

to the appendices, especially Appendix C, the *XNET Operating Manual* for additional material. Therefore, information is not repeated here that is more appropriately provided in other sections. The explanation, which follows is limited to the more obscure workings of XNET.

The complete XNET distribution consists of 19 files as shown in Figure 4-1. The main program, xNet is the heart of the network analyzer. XNET begins when the file xNet is executed. It will, in turn, load and execute other script files as the user specifies additional displays (i.e., MAP, TERM, etc.). Those other scripts/files fall into three categories: Tcl script files (.tcl extension, a total of 7), simulation script files (.sh extension, a total of 10) and a preferences file (filename *prefs*). Each of these, including the simulation files which are simple UNIX command script files, will be discussed in more detail.



XNET allows the user to specify three groups of preferences. Each of the groups is selected by way of its own procedure: *xSimul.tcl*, *xPort.tcl*, and *xPrefs.tcl*. Once the preferences have been selected, they are saved in the file *prefs*.

To display network information, four network data display windows are provided. Each display is created by its own procedure: *xMap.tcl*, *xNodes.tcl*, *xTerm.tcl*, and *xGraf.tcl*. The remaining ten files are simulation files that were created by recording actual network traffic.

4.1. xNet — Main Program Module

The file *xNet* is the heart of the network analyzer. This program file contains additional procedures which are described in detail below.

4.1.1. tick — XNET System Clock

The script, xNet, is a major procedure of XNET which performs three functions. First, it is responsible for creating a one second heartbeat which acts as the system clock and represents the time from which all XNET events are time stamped.

Second, at specified intervals (10 seconds), packet ages are updated in the two main arrays: *node_arr* and *conn_arr* by Tcl scripts *update_nodes* and *update_conn*. Both will be discussed in more detail later in this section.

It is worthy to note that a workstation with no other processes running should easily be able to accommodate the 10 second updating task while still being *nice* to other users on the system. However, XNET is intended to work on a variety of systems including Intel® 386 based machines which are relatively slow. In these environments, the processor may become overloaded when rapid updating (every second) is implemented. Therefore, a 10 second update was used as a compromise. However, as previously stated, AX.25 networks are slow and the 10 second updating period is not a significant compromise.

The following code segment located in the *tick* procedure of xNet, shows the ease of changing the update period should one wish to do so.

```
if {$tens == 10} {  
    set tens 0  
    update_nodes  
    update_conn  
}
```

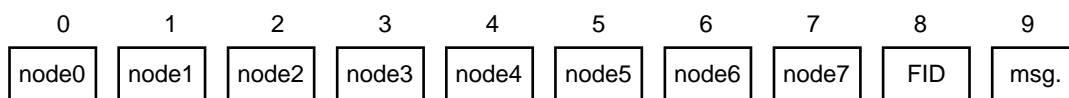
4.1.2. open_input — System Initialization

This procedure initializes and/or re-initializes key variables. It is called each time the program is STARTed. It also clears all open windows to assure that fresh data is not written onto existing window data. For example, the MAP, NODES, TERM, and GRAPH windows are all cleared before new data is accepted.

The procedure also reads the *prefs* file to ascertain key preferences that the user has specified (i.e., baud rate, parity, etc.). If the serial port is specified, the procedure opens and configures the serial port.

4.1.3. file_input — Get Packet Module

This is called each time data from the TNC appears at the serial port. The procedure is primarily responsible for parsing the data string. It extracts the nodes, frame ID, and text message. It then places the data in a temporary array, *node_buf*, for further analysis.



4.1.3.1. node_buf — Initial Packet Buffer

The size of the array *node_buf* (10 elements) was selected to allow a station to use up to 6 digipeaters (repeaters). For example, the following shows station W9IF transmitting a packet to KC5DSI using 6 digipeaters.

```
W9IF>WA2TGL>W2IY>K7YUI>WB2BIJ>WB2ILP>K8UIJ>KC5DSI [UI]: Hi!
```

The rationale behind the selection was based on the minimum number necessary (i.e., two). In other words, the array must support, at a minimum, two elements to accommodate a source and a destination (W9IF and KC5DSI in the example). The use of two digipeaters is not uncommon, therefore a working minimum should be four elements for the nodes. A technical design decision was made to accommodate up to eight node names in an effort to insure XNET would function for present and future cases. It is worthwhile to note, that an inspection of the code will indicate that a null is placed in the first unused element of the array, so the processor does not waste time analyzing all eight elements. For the sake of completeness, if XNET were to be given a packet with greater than eight nodes, the eighth node (element 7) would be incorrectly interpreted as the

destination node and all nodes after this would be ignored. Again for the sake of completeness, this error is not noted since it is virtually impossible and it would have little meaning when weighted with thousands of other packets.

4.1.3.2. NODE FILTERING

There is some filtering that is performed when packets arrive. An examination of the source code will indicate some of the filters. However, further clarification is in order regarding the filtering of broadcast packets. A node may broadcast as follows.

```
W9IF>CQ [UI]: Hello from DeSoto, Texas!
```

This example shows an unnumbered information frame (i.e., [UI]) that is a *broadcast*. In other words, the packet is sent to all nodes on the network. However, there are no nodes on the network with the name CQ. Therefore, the name is removed from the *node_buf* array and will not appear anywhere in XNET other than the TERM window which displays raw network traffic. It should be noted that there is no standard for broadcast names, however, there are a few pseudo-standard names frequently used. Those names that are often used are listed below in Table 4.1.3.2-1 and are not stored by XNET. More to the point, these names are filtered by XNET and do not appear anywhere. However, in the above example, the node W9IF would appear in the NODE window since it is a legitimate node sending a packet.

```
MAIL  
CQ  
ID  
QST  
NOS  
BEACON  
NODES
```

Since there are no standards for names, the user may use any name he wishes. Therefore, an exhaustive *filter list* is impossible which means that XNET can and will display nodes which cannot exist, such as BOB in the example below.

```
W9IF>BOB [UI]: Is anyone out there?
```

The Frame Identification (FID) occurs after the nodes in the packet. In the example below, [I;7,2] is the FID.

```
W9IF>KC5DSI [I;7,2]: Hello from DeSoto, Texas!
```

The AX.25 data link layer protocol specifies a type for all frames. Frames fall into three main categories, Information, Unnumbered, and Supervisory. A TNC such as the AEA-232, categorizes the frames as shown in Table 4.1.3.2-1.

C	Connect Command (also known as a SABM frame)
D.....	Disconnect Command
I.....	Information Command and Response
UA	Unnumbered Acknowledgment Response
UI.....	Unnumbered Information Frame
RR	Receive Ready Command and Response
RJ.....	Reject Command and Response
RNR.....	Receive Not Ready Command and Response
FRMR.....	Frame Reject Response
DM	Disconnected Mode Response

The packet terminates with a msg (message) which is the actual text to be delivered to the destination node, `Hello from DeSoto, Texas!` in the example.

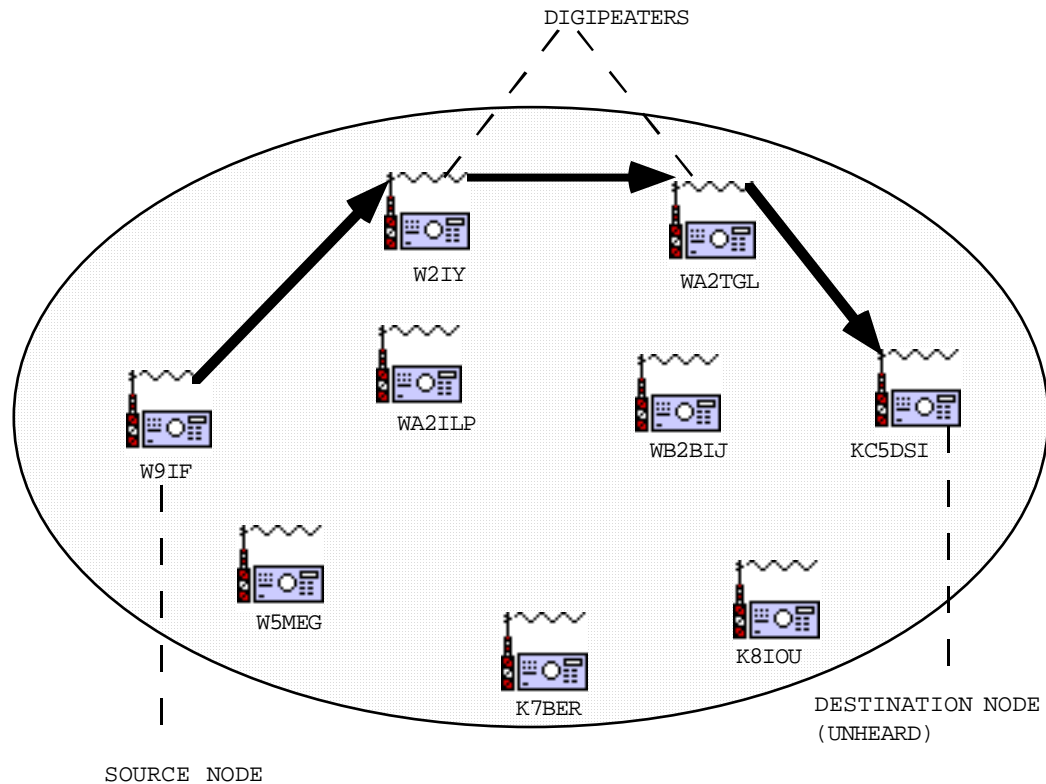
4.1.3.3. HEARD AND UNHEARD NODES

Remember, that this is a wireless network where distances are limited. Therefore, to extend the range of a node, other nodes on the network can be used as *digipeaters*. Amateur radio operators have coined the term, *digipeater* for this process whereby another node is used to extend the range of a node. Note that the terms *repeater* and *digipeater* are used synonymously in this paper.

A closely related topic to digipeating is the concept of *heard* and *unheard* nodes. These terms will be used extensively in this paper. Let's look at a few simple packets to

help explain this important concept. The graphical representation of this traffic is shown in Figure 4.1.3.3-1.

```
W9IF*>W2IY>WA2TGL>KC5DSI [I;7,2]: Hello from DeSoto, Texas!  
W9IF>W2IY*>WA2TGL>KC5DSI [I;7,2]: Hello from DeSoto, Texas!
```



In both the first and second examples, W9IF sends a packet to KC5DSI. What differentiates the cases is which of the nodes is actually *heard*. Another way of looking at it is to ask which station is on the network XNET is monitoring. In the first case, W9IF is heard, most likely a local node that can be heard directly (i.e., within 30 mile radius). The second example, shows W9IF at a distance outside the local network unable to reach KC5DSI directly. Therefore, W2IY and WA2TGL are used as digipeaters to extend the range of W9IF and enable him to send a packet to KC5DSI.

Normally, XNET would have no way to differentiate which node is *heard* and which node is *unheard*. In point of fact, with no other input, XNET assumes that the first node

in the packet string is the heard node and the last node is the unheard node. Fortunately, most (if not all) TNCs which have access to the raw (bit level) information, can determine by an examination of the address field, which of the packets is heard. The user can request that the TNC append an asterisk (*) to the heard node through the use of the MRPT command. If the MRPT function is OFF, only packets from the *originating* station and the *destination* station are output from the TNC. If MRPT is ON, the call signs of all stations in the entire digipeating path are provided.

Two more examples will further explain this concept. The first shows a packet with MRPT ON. The second example shows the same packet with MRPT OFF resulting in the source node name being *hidden* from, and therefore not displayed by XNET. Therefore in this latter case, XNET will display W2IY as being part of the network. The actual source of the packet W9IF is not displayed nor is the unheard digipeater WA2TGL.

```
W9IF>W2IY*>WA2TGL>KC5DSI [I;7,2]: Hello from DeSoto, Texas!  
W2IY*>KC5DSI [I;7,2]: Hello from DeSoto, Texas!
```

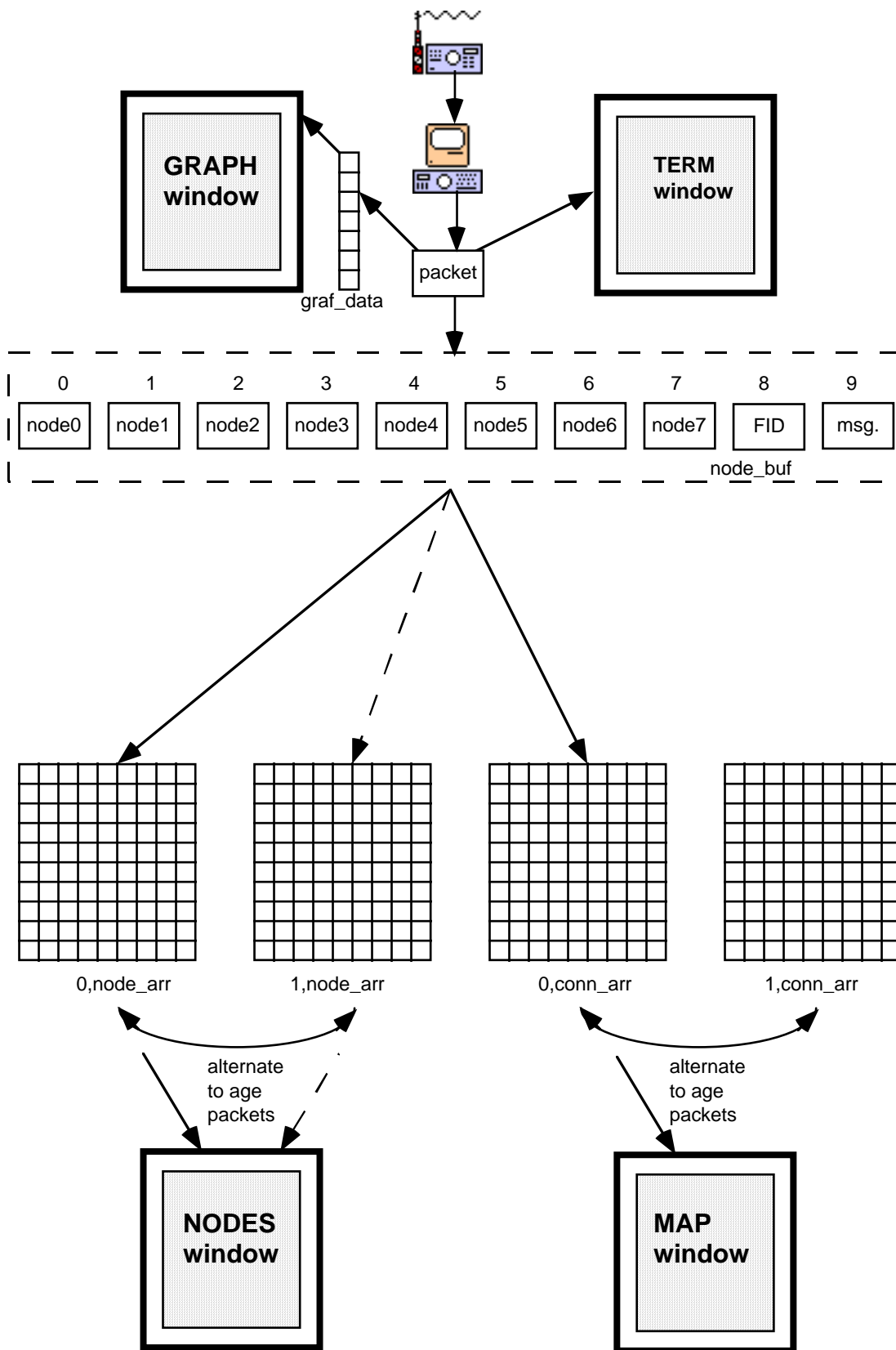
Normally MRPT should be ON since we are interested in knowing the heard nodes. If MRPT is OFF, XNET will display nodes that are not on the network. In fact, since the source will be assumed to be the heard node, XNET may actually be displaying a node that is not only not on the local network, but may have actually transmitted the packet several minutes earlier, since digipeating can significantly delay transmissions.

If an asterisk appears anywhere in the name of a node, XNET will use this as an indication that this is the heard node and that all other nodes are unheard.

4.1.3.4. DATA FLOW

Before explaining the node array (*node_arr*) and connection array (*conn_arr*), an understanding of the flow of data is imperative. Figure 4.1.3.4-2 shows data entering XNET from the TNC. The data is stored in a temporary buffer (*node_buf*). The contents

are then transferred to the node and connections arrays. However, here is where the process becomes a little tricky.



Recall that XNET must *age* packets in order to determine if a node has left the network. This requirement places major constraints and difficulties on the design of the system. There are several ways to implement the aging process. However, the method must be efficient since this process requires computer intensive manipulations. The method chosen to implement aging is to flip-flop both the *node_arr* and the *conn_arr* with alternate copies of themselves.

Let's look at the node array in more detail. Every 10 seconds (update period, see *tick* procedure), each node is examined to see if its age exceeds the time-out specified by the user. If the node is a heard node, it is compared with the heard node time-out value. In a similar manner, if it is an unheard node, the age is compared with the unheard time-out value. Recall that both of these time-out values are specified by the user by way of the PREFS window. If the time-out value is not exceeded, the node name along with all the information that comprises the statistics for the node are copied to the other array. If the time-out value is exceeded, the node information is not copied resulting in the node being removed from the system.

The connect array functions in a similar manner to perform the aging process. However, it is slightly more complex since it needs to re-pack the array both vertically (i.e., remove source nodes) and horizontally (i.e., remove destination nodes). We shall discuss the connection array in greater detail in a subsequent section.

This method of packet aging would be cumbersome and time consuming if the number of nodes were large or the horsepower of the machine XNET were running on were marginal. However, even in congested metropolitan areas (i.e., Chicago, Dallas), there are rarely more than 25 to 40 active nodes on the network. In addition, the method may not be useful if the update period were increased (i.e., < 1 second). However, there is little need to update displays more rapidly than once every 10 seconds.

4.1.3.4. TIMING

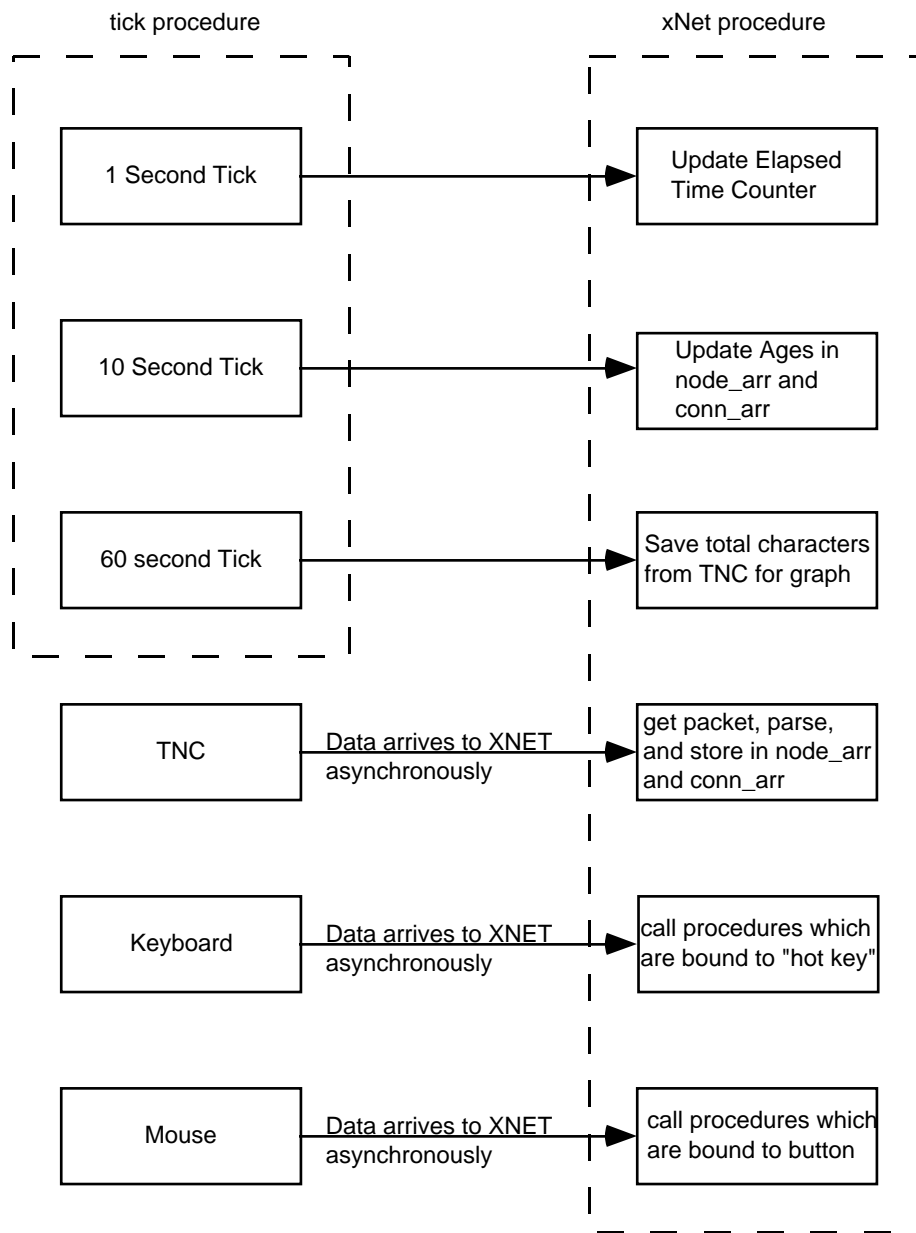
Figure 4.1.3.4-1 shows high level XNET events. Every second an event occurs which provides the means to update the elapsed time counter. At 10 second intervals, the *update_nodes* and *update_conn* procedures are called to update the ages of the packets and to remove packets which have exceeded their time-out. At the end of every minute, the total number of ASCII characters that have been received by XNET from the TNC are stored in an array which will be used to plot network utilization. All of these events are triggered and controlled by the *tick* procedure.

XNET also accepts asynchronous events from the TNC, keyboard, and mouse. The *file_input* procedure is called when characters arrive at the serial port. When a *hot key* (shortcut key) is selected the command specified by that event is called. The following is a code segment that shows how a single keystroke is used to call a procedure. Specifically, when "g" is depressed, the *xGraf.tcl* procedure is executed which plots the graph.

```
bind . <g> {
    source xGraf.tcl
    xGraf
}
```

Mouse events allow the user to select a procedure when a button is pressed. Note that the *body* of the command is identical in these two examples which allows the user to control XNET by way of the mouse or keyboard.

```
button .graf -text "GRAPH" -command {
    source xGraf.tcl
    xGraf
}
```

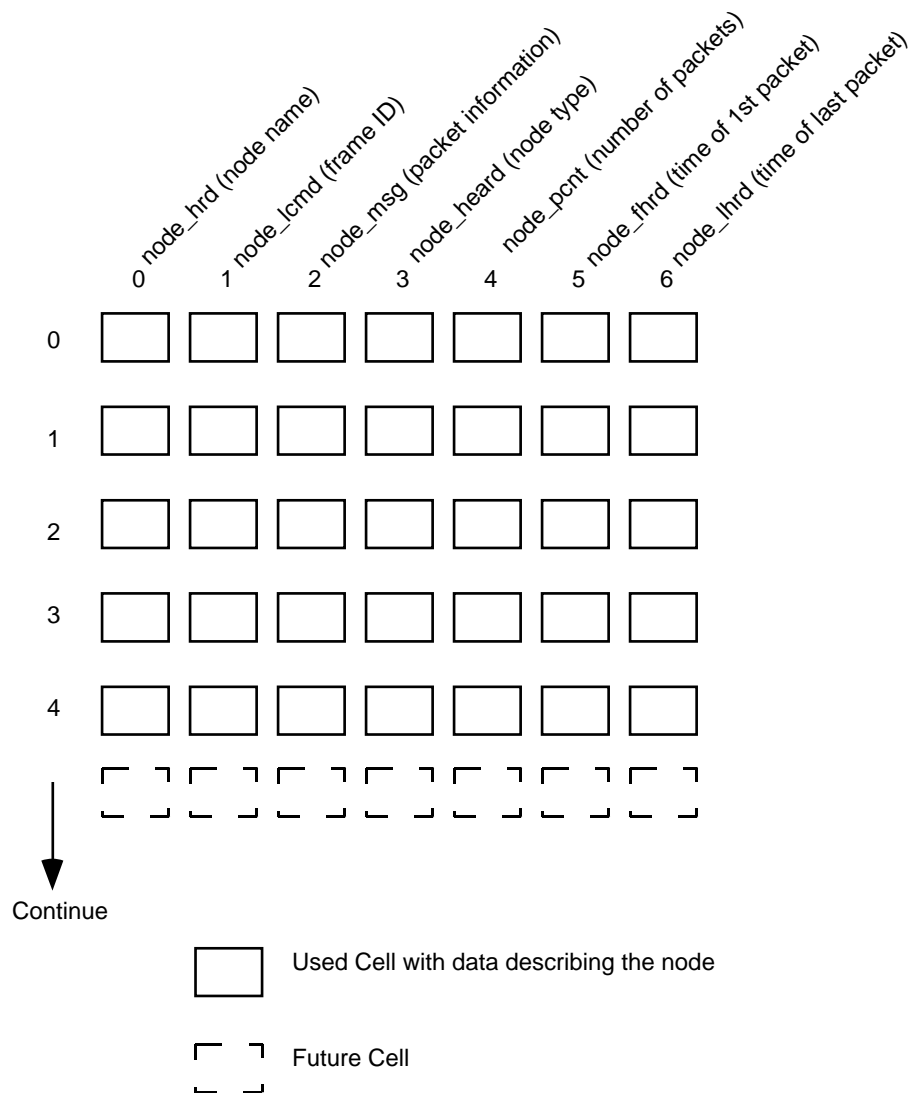


4.1.4. node_arr — Permanent Array for Active Nodes

The array *node_arr* contains an up-to-date copy of all nodes on the network. This includes *heard* and *unheard* nodes. The *conn_arr* contains only nodes that have sent or have received a packet (assuming MRPT is ON).

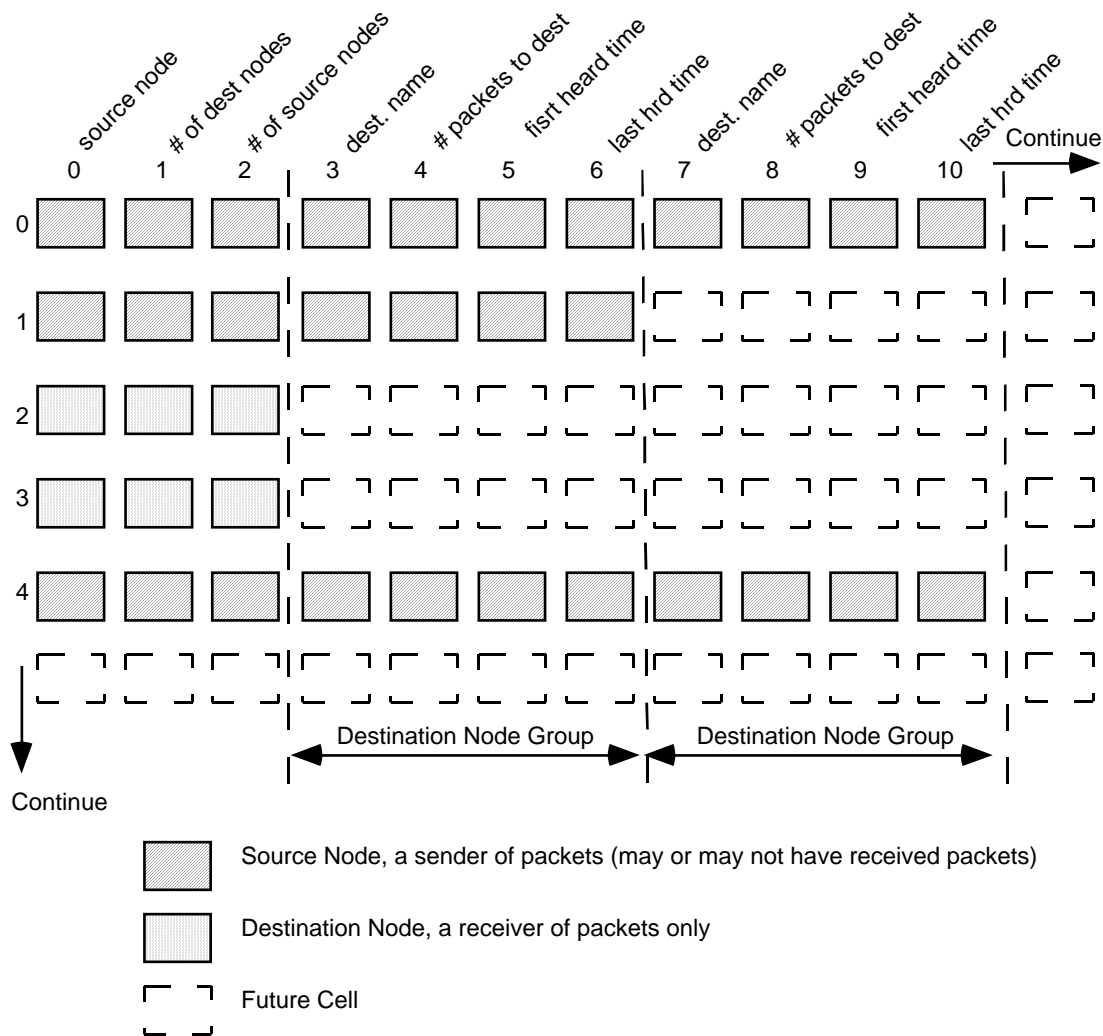
For each node, there are seven data elements. An examination of the array will show that the node name (heard node) is always placed in array index 0. The node's frame type

(i.e., [I;7,6]) is placed in the next array cell, index 1. The message sent by the node is placed in array index 2. The type of node follows next in the array. An *H* is stored for a heard node and *U* for an unheard node. This flag is used for various purposes including specifying the color in the NODES window. The number of packets sent by the node is stored in index 4. The time in seconds when the node was last heard is placed in array index 5, and the current time is placed in index 6. Taking the difference between the two time stamps will later be used to determine the age of the packet. Note that when we refer to time, it is the time in seconds from when XNET was first started.



4.1.5. conn_array — Permanent Array for Connected Nodes

The connection array, *conn_arr*, contains similar, but not identical information to *node_arr*. As mentioned previously, a major difference is that the connection array contains only source nodes that have sent a packet to a legitimate destination node. Digipeaters (repeaters) are not included in this array.



Like *node_arr*, index 0 is the name of the source node. The following index element includes the number of nodes the source is connected to. Array index 2 contains the number of nodes that the source node itself is connected to. Elements 3 through 6 represent a group of statistics for each node that the source is connected to. The

description for these index elements are: name of the destination, number of packets sent to this destination from the source node, the time in seconds when the source last sent a packet to the destination node, and the current time.

4.1.6. update_nodes — Update Active Nodes

Updating is the process whereby packets are aged and those that have exceeded the time-out period are deleted. The exact process is not complex, but tedious.

The procedure begins by first inserting the present time into the proper element of the *node_arr*, specifically the *node_lhrd* element. A second element of the array called *node_fhrd* (node first heard) contains the time a packet was received from this node. Taking the difference between these values yields the age of the packet (in seconds). Let's go through an example to explain this important concept.

Assume that XNET has been running for 700 seconds when a new node is detected on the network. Assume also, for this example only, that the node sends one and only one packet and then leaves the network. For this packet, XNET will timestamp *node_fhrd* and *node_lhrd* with the value of 700. At this point in time, the age of the node is 0 (700-700). Recall that the *tick* procedure updates the nodes every ten seconds. Therefore, when *update_nodes* is called, *node_lhrd* is updated to 710. The age of the packet is now 10 seconds (710-700). The same process occurs 10 seconds later. However, each time the age of the node is updated, it is compared with the time-out period. If the time-out period is exceeded, the node is dropped from the array. Assuming the user has specified a time-out value of 30 seconds using PREFS window, when the age of the packet is 30 seconds (730-700), it is deleted from *node_arr*.

The exact mechanism for dropping nodes is more or less a brute force method. Every 10 seconds, active nodes (those that have not exceeded their time-out) are moved from the existing array to a new copy of the array. Nodes that have exceeded the time-out

period are not moved to the new array. A three dimensional array was developed for this purpose. An example is shown below.

```
node_arr($n_a,$i,$node_lhrd)
```

The variable `n_a` alternates between 0 and 1 to differentiate between the old and new copy of the array. This process is shown at a high level at the bottom of Figure 4.1.3-2.

Let's return to our original example but this time the node will remain on the network. Again, the node sends a packet at elapsed time 700. Now assume that after 15 seconds has passed, the node sends another packet. The value of `node_lhrd` and `node_fhrd` are reset to 715 and again the age is 0. This occurs each time a node transmits a packet. If a node continues to send packets before it has timed out, the age of the packet never exceeds the time-out period and the node is never removed from the array.

Note also that XNET differentiates between heard and unheard nodes. Specifically, there are two time-out periods. The variables are `hrd_to` and `uhrd_to`, both are set in the PREFS menu. Therefore when comparisons are made to determine if a node should be dropped, the type of node is first ascertained. The following is the source code that performs this task.

```
set age [expr $node_arr("$n_a,$j,$node_lhrd") \
- $node_arr("$n_a,$j,$node_fhrd")]

# set flag for this node to "no save" (assume time-out)
set node_save 0

# Check time-out of "heard" station
if {($node_arr("$n_a,$j,$node_heard") == "H") && \
($age < $hrd_to)} {
    set node_save 1
}

# Check time-out of "unheard" station
if {($node_arr("$n_a,$j,$node_heard") == "U") && \
($age < $uhrd_to)} {
    set node_save 1
}
```

There is yet a third time-out period, *con_to* which will be discussed in the *update_conn* section next and is mentioned here only for the sake of completeness and to emphasize that it is not part of the *update_nodes* procedure.

4.1.7. update_conn — Update Connection Nodes

It was mentioned that the *update_nodes* procedure was not complex, merely tedious. However, *update_conn* is both tedious and relatively complex. The complexity arises from the structure of the array and the updating process.

Nodes cannot be so easily removed by merely checking that a single time-out period has been exceeded. A source node may be connected to any number of nodes. Therefore each of the destination nodes must be checked and only when a source has no destinations, and the source itself is not a destination for another node, can the node be removed.

In the *update_nodes* procedure, nodes are moved once to complete the aging (updating) process. However, in *update_conn*, nodes are moved twice. First, for each source node, the age of each of the destination nodes that it is connected to the node must be computed. This is very similar to the aging process outlined in the *update_nodes* procedure. In *update_conn*, if the age exceeds the time-out period, the destination node is not transferred to the new array. This is a horizontal packing of the array since the number of source nodes remains fixed at this time. An examination of Figure 4.1.5-1 helps show the process.

Second, if a source node has no destinations (i.e., all the nodes that it has sent a packet to have timed-out), and if the source node is not a destination for another node, then and only then may the source node be removed from the array. This occurs by moving all nodes that still have connections back to the original array. In essence, the timed-out source nodes are dropped since they never make it to the new array.

Making the process complex is the way the destination node names are indexed. Take the following example.

```
W9IF>KC5DSI [I;4,2]: Hello from DeSoto, Texas!
```

Assume node W9IF is at position 3,0 in the array and that KC5DSI is located at position 8,0. The source node, W9IF, keeps track of connections by saving the value 8 which indicates a connection (index) to KC5DSI. Now if a source node located between positions 3 and 8 is removed, then it is imperative that W9IF not point to index 8 (KC5DSI) but to the new location of KC5DSI which may be 4, 5, 6, or 7 depending on the number of source nodes that will be removed from the array. The following portion of code performs this task.

```
# Before moving and removing nodes with no src. or dest.\
  all destination indexes must be updated.

for {set i 0} {$i < $conn_cnt} {incr i} {
  set dest_name 3

  for {set j 0} {$j < $conn_arr("1,$i,1")} {incr j} {
    set del_cnt 0
    set name_now $conn_arr("1,$i,$dest_name")

    for {set m 0} {$m < $name_now} {incr m} {
      if {($conn_arr("1,$m,1") == 0) && \
        ($conn_arr("1,$m,2") == 0)} {
        incr del_cnt
      }
    }

    # subtract no. of nodes to be deleted
    incr conn_arr("1,$i,$dest_name") -0
    puts "Value to decrement $i $j $m is = $del_cnt"
    incr dest_name 4
  }
}
```

This apparently tedious operation has the advantage of assuring that the array is always *packed*.

4.1.8. close_input — STOP XNET

The *close_input* procedure is responsible for closing the open file and killing the background process which was created when the serial port or the simulation file was selected.

4.2. xMap.tcl — Script to Display Map Window

This procedure is similar to *xNodes.tcl*. Whereas *xNodes.tcl* displays the *node_arr* array, *xMap.tcl* displays the *conn_arr* array. The major difference is that the information is displayed in a graphical format in the MAP window.

It is important to note that only heard nodes with a destination are displayed. The following are two examples with a heard node, KC5DSI, and a destination node W9IF.

```
KC5DSI>W9IF [I;7,7]: how are you?  
KC5DSI>WA2TGL>W2IY>W9IF [I;7,7]: how are you?
```

In the second packet example, the digipeating nodes (WA2TGL and W2IY) are not displayed. This was done primarily to keep the display window uncluttered. The NODES window should be examined should the user wish to ascertain all active nodes on the network.

4.2.1. clear_map_data — Clear Map

This procedure clears all data on the map. This task is accomplished by deleting all objects that have the *tag* name *map_data*.

4.2.2. map_insert — Draw Map

This procedure is responsible for drawing all map data. All nodes are displayed twice, once on the left and once on the right side of the window. Lines are then drawn from left to right to indicate that a packet has been sent to the destination node located on the right side of the window. A non-connected node will have a thin brown line from it to represent the transmission of a packet. For two connected nodes, thick black lines are drawn from each of the nodes. The next section will discuss in more detail the term *connection* as used by XNET.

4.2.2.1. CONNECTION DEFINITION

A connection is said to exist when a source node sends a packet to a destination node and the destination returns a packet. To determine which nodes are connected, the procedure selects each of the source nodes and each of the destination nodes that a source node has sent a packet to. The *conn_arr* array is further searched to ascertain if the destination node has sent a packet to the source, if so a connection is said to exist.

It is important to note that we use the term connection loosely here. The AX.25 protocol is connection oriented. This means that specific packets with well defined frame types must be sent between the two nodes before a true connection is said to exist.

A true AX.25 connection packet exchange is shown below. The first packet shows KC5DSI sending a Connect Command (also known as a SABM frame) with the P bit set. The second packet shows W9IF responding by sending an Unnumbered Acknowledgment frame with the F bit set (the proper response to a P bit being received).

```
KC5DSI>W9IF [C,P]
W9IF>KC5DSI [UA,F]
```

To further explain a possible erroneous connection, assume NODE1 sends a *disconnect* packet to NODE2 and NODE2 transmits a *disconnect* packet to NODE1. Clearly, the types of packets indicate that there is no connection, however, XNET would consider the packet exchange as a connection. The complexity of XNET would rise an order of magnitude if frame IDs were retained in an effort to properly detect a connection. While important, the definition of a connection used by XNET is satisfactory for its purposes.

4.3. xNodes.tcl — Script to Display Active Nodes

The main program, xNet, accepts data from the TNC and stores it in a buffer, *node_buf*. This data is then filtered and stored in another array called *node_arr*. The array contains the information that is displayed by the *xNodes.tcl* procedure.

The NODES window displays all active nodes on the network along with other information relevant to the node. The heard nodes are displayed in blue, while the unheard nodes are displayed in brown.

A heard node is typically the originating node. For example, take the packet as shown below. The originating node, KC5DSI, is the heard node. The receiving node, W9IF, is the unheard node in this example.

```
KC5DSI>W9IF [I;7,6]: Are you there?
```

If W9IF returns a packets such as shown below, then W9IF is also considered a heard node.

```
W9IF>KC5DSI [I;7,7]: Yes I am here!
```

Remember that the distinction between heard and unheard nodes is important since a heard station is considered to be actively sending packets. The receiving (unheard) node may or may not be actively returning packets and may not even be on the network.

4.3.1. make_buttons — Create Buttons

This procedure places buttons on the *canvas* previously created. Initially the buttons contain no text other than an index number used to uniquely specify it. Another

procedure, *nodes_insert*, will change the text of the button which forms the basis for the information that is displayed.

4.3.2. clear_buttons — Clear Buttons

This procedure removes (clears) all text from the buttons and replaces it with a number shown in the left hand column of the button. This procedure is called each time XNET is started.

4.3.3. nodes_insert — Display Node Statistics

In this procedure, the previously created buttons are updated with text which is displayed in the window. This procedure takes the information in the array, *node_arr* and formats it for the display. A simple conversion is also implemented to convert the age of the packet stored in seconds, to a normal hours, minutes, and seconds format.

4.4. xTerm.tcl — Create Display and Print Raw Packets

At first glance, this procedure would appear to be simple to implement, at least the author expected it to be easy. However, a terminal window created by Tcl/Tk is very different than a terminal window of an intelligent (or dumb for that matter) terminal. When a text window is created there are none of the usual means of controlling the position of the cursor such as in a VT100 or VT220 terminal. Therefore, there is more to the procedure than merely opening a window and inserting characters, even though that is all that the procedure performs.

4.4.1. term_init — Initialize Terminal

Here we initialize the terminal. For historic reasons, the first row is 1 and the first column is initialized to zero. The window is initialized with blank lines. This step is necessary since the way that new data will be inserted is to specify the location of existing

characters within the display, rather than the normal cursor control method of inserting characters in traditional VT100 compatible terminals.

4.4.2. term_clear — Clear Terminal

This procedure deletes all characters in the window and then calls *term_init* to initialize the window with blank lines.

4.4.3. term_down — Move Cursor Down

In this procedure the cursor is moved down. If the cursor is already at the end of the window, then the line is moved down by deleting the first line and creating a new line at the end.

4.4.4. term_insert — Insert the Text in Window

This procedure actually inserts the character string into the window. The string is inserted at the cursor position.

4.4.5. term_update_cursor — Move Cursor

This procedure updates the visible cursor.

4.5. xGraf.tcl — Script to Display Graphs

This script file contains the code to display channel network utilization in 1, 5, and 25 hour formats. The main program, xNet, collects the raw data in an array (*graf_data*) and provides it for plotting. The exact data that is collected in the array is the number of characters that has been received from the serial port during the preceding one minute sampling period. In other words, a running total of characters is accumulated. The *tick* procedure which performs the time management function, then stores the value into the array for later display. The following code segment shows this operation.

```
if {$secs == 60} {
    set secs 0
    incr mins

    # Save tot chrs this past minute, \
      reset circulator buffer if end of buffer
    incr graf_indx
    set graf_data($graf_indx) $pac_chr_tot
    set pac_chr_tot 0
}
```

The sixty second sampling period should not be confused with the ten second update packet age routines previously discussed. In the former, the number of characters from the serial port are stored, in the latter, XNET updates the ages of all packets and deletes packets if they have timed-out. See *update_nodes* and *update_conn* procedures for detailed explanation.

The computation of network utilization is shown in the source code below. Note that it uses the radio baud rate which is typically 1200 or 9600 baud. This is the speed of the network, and should not be confused with the bit rate of the communications link between the computer and TNC. In discussing the XNET network utilization graph, it is important to understand that the graph shows *relative* not *absolute* utilization. Therefore emphasis was placed on developing a relationship which described the packet traffic. One reason for this design decision stemmed from the inability to compute an absolute value due to the difficulty of knowing the overhead associated with each packet. Remember, XNET does not have access to the raw level bit stream, it receives highly filtered data. Since there are variables which determine the overhead, a variable was added to the equation (i.e., *pac_overhead*) which can be changed to adjust the relationship. At present, it is set to 1 which experience has shown gives a good graphic distribution.

```
set pac_overhead 1
set graf_ufac [expr 60 * $pac_overhead * $rbaud / 8]
```

4.5.1. graf1 — Create 1 Hour Graph

This procedure, like procedures *graf5*, and *graf25* which follow, is straightforward. It clears the window by removing the axes, titles, and data. It then draws new axes, a title appropriate for the graph (i.e., 1, 5, or 25). Finally new data is plotted by calling procedure *graf_it* which draws the vertical bars.

Before calling *graf_it*, a variable is set to indicate the type of graph to be displayed which represents the number of data points. For a one hour graph, sixty bars are drawn to represent the sixty minutes.

A close examination of the 1 hour plot will show that the 60 vertical data lines are separated with spaces. To be exact, the lines are separated by 4 pixels, the fifth being the actual data line.

The horizontal x-axis has a 300 point resolution. The selection of 300 points was not arbitrary, nor was the selection of 1, 5, and 25 hours plots. The decision to use 300 points was based on the way Tcl/Tk draws lines. Tcl/Tk offers single pixel resolution. Trial and error showed that a 300 pixel display is a good size to view (not too small nor too large). From the 300 pixel parameter, 1, 5, and 25 hour graphs followed as they are evenly divisible by 300. Specifically, plots use 60, 300, and 1,500 point plots respectively.

4.5.2. graf5 — Create 5 Hour Graph

This procedure clears the graph and draws a new graph appropriate to display 5 hours of data. Specifically, 300 points are plotted. The 5 hour graph uses all 300 pixels of the x axis.

4.5.3. graf25 — Create 25 Hour Graph

Like the previous procedures, *graf1* and *graf5*, this procedure clears the graph and draws a new graph consisting of the entire 1,500 points (25 hour) buffer.

This plot actually stuffs 1,500 data points into 300 pixel locations. This means that 5 vertical lines are drawn for every pixel width or 5 vertical lines are drawn on top of each other. This required a design decision. For example, since only one line is visible, only every fifth data point should be drawn. This leads to the logical question, *if you are going to draw every fifth data point, which ones do you pick?* A second possibility is to select a group of 5 data points and then compute the mean. Again this leads to yet other questions, *do you select the minimum, maximum, or mode value rather than the mean?*

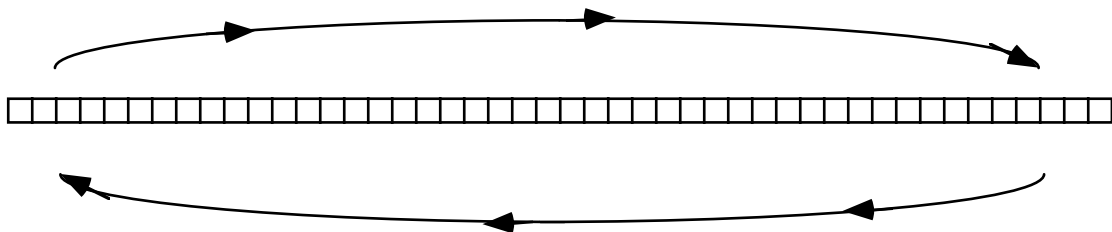
The decision was made to plot 5 data points on top of each other, resulting in the visual appearance that the **maximum** traffic is plotted since drawing a longer line over a shorter line hides the short line. This solution had the advantage of allowing all graphs to be treated in a similar manner and saving processor time. For example, taking 1,500 points and averaging them down in groups of 5 to 300 points would require valuable processor time. Although admittedly, drawing 5 lines rather than 1 also requires time. If processor time were critical, a careful analysis might be in order.

4.5.4. graf_it — Draw the Graph

This procedure is the heart of the graphing display routine. It is responsible for taking the data that has been collected and plotting it. Although it is barely a page of source code in length, the code is relatively tricky. The code implements a circulating buffer

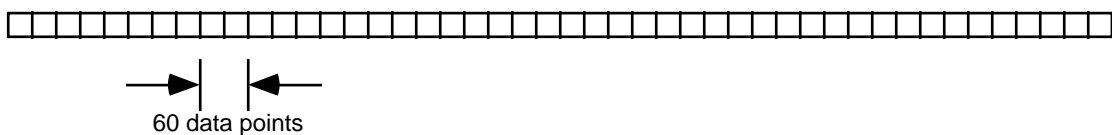
from which elements of the buffer are removed to plot the graph. There are actually two index pointers, one to point to the location where the next value should be stored, and another pointer which addresses the location from which to begin removing values.

XNET collects the data in the xNet procedure in the circulating buffer. At the end of every minute, the number of characters received is saved as an element of an array which contains up to 1,500 data points (1,500 minutes equals 25 hours). When the end of the buffer is reached, the index pointer into the array is returned to the beginning and the process begins again.

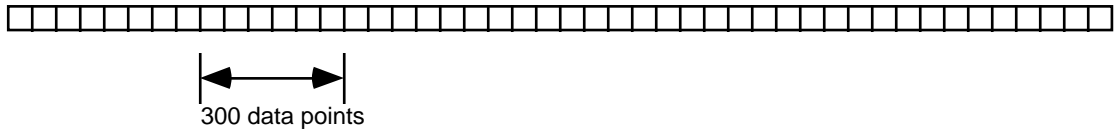


The graphing procedure is responsible for extracting the correct number of data points for plotting. For 1 hour, the number of points to be extracted is 60. For all graphs, the latest data points are selected and plotted which means that old data is overwritten.

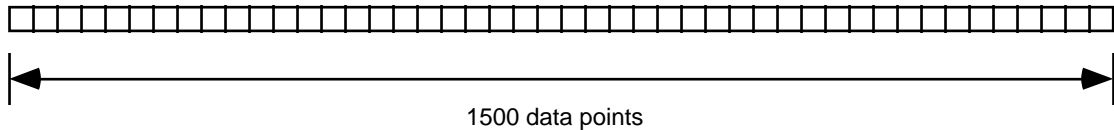
There is an additional case that must be taken into account, specifically when insufficient data has been collected to complete the graph. For example, at the end of the first minute there is only one data point to plot, yet the 1 hour graph (as an example) is expecting to plot 60 points. An examination of the source code shows how this case is handled.



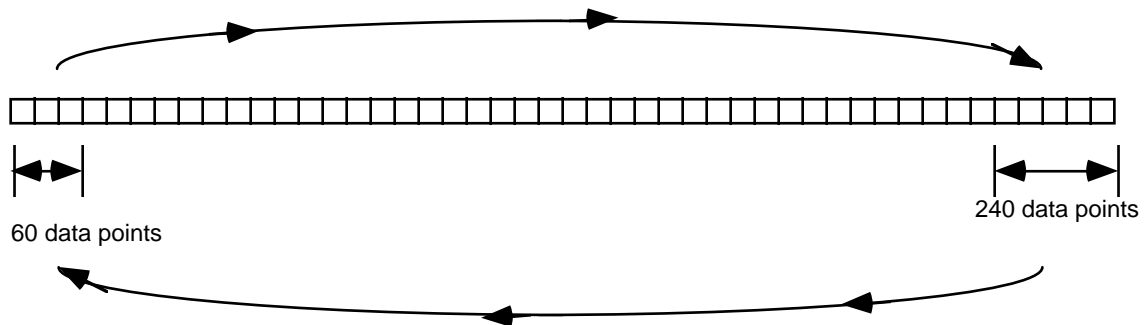
The five hour graph requires that the most recent 300 points (300 minutes) be used to construct the graph.



The 25 hour graph uses the entire buffer of 1,500 points (1,500 minutes).



The tricky part occurs when the end of the buffer is reached. Both the data insertion code and the data extraction code must properly handle this case. This is performed by checking for this condition and resetting the pointers accordingly.



The height of the bar is a function of the network traffic. Specifically, the network traffic represents the number of characters received by XNET for the one minute sampling period, and the radio baud rate. The radio baud is the bit rate of the physical layer of the network, not the bit rate between the TNC and the XNET computer. The radio baud rate is specified by the user in the PREFS window.

4.5.5. clear_graf_data — Clear Window

This is a simple (single line) procedure that is used to clear all data from the graph. The axes and other information that are displayed in the window is removed or updated elsewhere in the code.

Tcl/Tk provides a simple means of removing groups of objects. For example, the code below draws a line.

```
.cGraf.c create line 50 100 75 200 -tag graf_data
```

The key parameter here is the *tag* that is appended to the command. The tag provides a name for the line. In the case of the 25 hour graph, there are over 1,500 data lines, all having the same *tagname* allowing all lines to be removed with the single command shown below.

```
.cGraf.c create delete graf_data
```

4.6. xSimul.tcl — Script to Create Simulation Window

The script file, *xSimul.tcl*, allows the user to specify important serial port parameters⁴ that are used by the program to configure the serial port. When the script is initially called, it reads the *prefs* file to initialize the display.

The window contains a group of 11 *radiobuttons* to allow the user to select the source of input for XNET. A radiobutton is a class of widget that may exist in one and only one state. Specifically, when a user selects one button in the group, a previously selected button belonging to the group is cleared. A radiobutton works in an identical manner as the buttons on a car radio. Selecting a new radio station, clears the previously selected radio station (hence the name *radiobutton*).

This widget allows the user to select from one of the 10 pre-recorded simulation files or the serial port. When selected, the information is saved to the *prefs* file as a permanent means of saving the settings.

If the program is running, a warning window is displayed to indicate that the changes will not be implemented until XNET is first stopped and started. Note that *stop* does not mean that the user must exit the program, merely that the STOP button must be selected.

⁴ See the operating manual for details of the operation and meaning of these variables.

Tcl/Tk provides a means of grabbing the user's attention by not allowing any other functions to be performed until the user acknowledges the dialog box. The following code, show the statements used to prevent the user from making any additional selections until the dialog box shown in Figure 4.6-2 is acknowledged.

```
# Wait for user to acknowledge warning dialog
tkwait visibility .attn
grab set -global .attn
tkwait window .attn
grab release .attn
```

4.7. xPort.tcl — Script to Create Port Selection Window

The script file, *xPort.tcl*, allows the user to specify important serial port parameters⁵ that are to be used by the program to configure the serial port. When the script is initially called, it reads the *prefs* files to initialize the display.

This window contains five groups of radiobuttons to set the port, baud rate, number of stop bits, frame size, and parity. The UNIX serial port provides the user with many parameters, far more than the four that are provided here to be configured by the user. The number of parameters was limited to these since they represent the most common parameters that may need to be modified by the user. In the event the serial port requires additional changes to meet unique interface requirements, the user must make these additional changes using conventional UNIX commands (i.e., *stty*). The Operating Manual provides more details regarding the exact means to accomplish this.

Like the *xSimul.tcl* script, if the program is running, a warning window is displayed to indicate that changes will not be implemented until XNET is first stopped and started. The warning message is virtually identical to that as shown previously in Figure 4.6-2.

⁵ See the operating manual for details of the operation and meaning of these variables.

4.8. xPrefs.tcl — Script to Create Preferences Window

The script file, *xPrefs.tcl*, allows the user to specify key variables⁶ that are used by the program. The window contains four groups of *radiobuttons*. When this script is initially called, it accesses the *prefs* file to initialize the radiobuttons to the values previously selected and saved by the user.

⁶ See the operating manual for details of the operation and meaning of these variables.

Unlike the previous preference selection menus, SIMUL and PORT, changes selected by the user are implemented almost immediately.

4.9. simX.sh — UNIX Simulation Script

There is a group of ten files, *sim1.sh* to *sim10.sh*. These are UNIX shell scripts that were created by recording **actual AX.25 packet traffic**. The script was developed by taking raw ASCII data and adding the necessary *sleep* and *echo* commands to recreate actual packet traffic. The packet data was taken from packet networks in Chicago and Dallas.

Since these files are UNIX scripts, they may be executed independently of XNET. Below is a sample of one of the simulation scripts. Notice that the script is written to loop indefinitely (i.e., while 1 = 1). All scripts are virtually identical, they merely include different packet traffic text in the *echo* command. Below is a short example. Most of the simulations provided in the XNET distribution are significantly longer.

```
:
# Packet simulation script for XNET

cnt=1
space=4

# loop forever!
while [ 1 = 1 ]
do
    echo "ILLYL*>NS4F [I;2,2]:Connected."
    sleep $space

    echo "KA9Q>NOS (k9vxw.ampr.org.)"
    sleep $space

    echo "WA9AEK-1*>NODES [UI]CF:ILLYL"
    sleep $space

    echo "ANL*>N9WMF>W9IF-2>KC5DSI>K8GHJ-2 [I;4,1]:@RTTY DX"
    sleep $space

    echo "ILLYL*>NS9F>W9IF-2>KC5DSI [I;6,7]:"
    sleep 10

    cnt=`expr $cnt + 1`
done
exit 0
```

4.10. prefs — Preferences File

The file *prefs* is a data file used to retain the preferences set by the user. Sample contents of the file are shown below.

```
0
/dev/cua3
9600
2
7
N
1200
300
300
60
```

The file consists of 10 records. The first is the simulation record. It may take on values from *sim1.sh* to *sim10.sh* or 0. If the value is 0, as shown in the example above, it specifies the selection of the serial port rather than one of the simulation files. This record specification is selected by the user using the SIMUL window.

Records two through six are the port, baud rate, stop bits, frame size, and parity records respectively. These preferences are set by the user by way of the PORT window.

The remaining records, seven through ten, are the radio baud rate, heard station time-out (all time-out values are specified in seconds), unheard station time-out, and the connection time-out respectively. This group of preferences are set by the PREFS window.

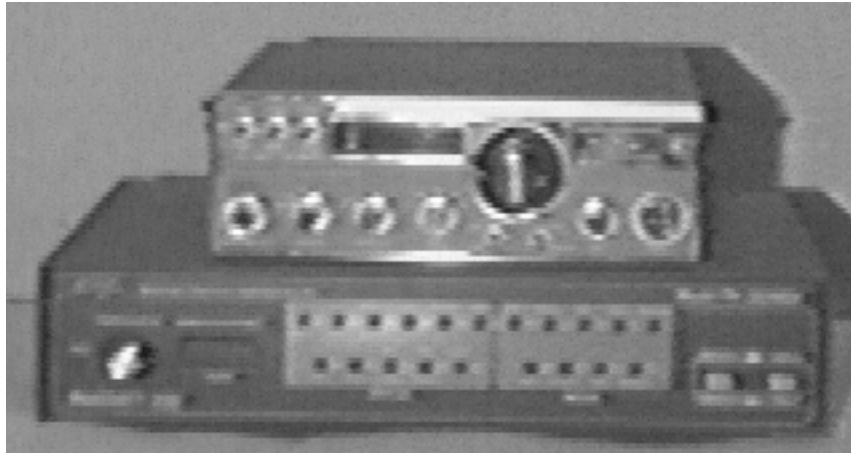
5. THE HARDWARE

The RF transceiver and TNC used during the development of XNET are shown in Figure 5-1. The upper unit is the transceiver which accepts RF signals in the 144 to 148 MHz range. The signal is then demodulated and reduced to audible tones similar to a FAX machine. This analog signal is then used to drive the TNC.

The lower unit, the TNC, is responsible for taking the audible tones, converting the tones to binary data, and then disassembling the packet. The disassembled packet is

provided to the computer system in the form of an ASCII string. It is this ASCII string that is used by XNET.

The TNC performs the reverse task also. It will accept ASCII strings from the computer system, assemble a packet, and output audible tones to be sent to modulate the transceiver connected to the network.



Most of the hardware used to develop XNET is shown in Figure 5-2. A small portion of the Apple Macintosh computer system is shown to the left. It was an invaluable aid to testing XNET. The test procedures will be discussed in more detail in a subsequent section. The TNC and RF transceiver are shown to the right of the Macintosh computer, followed by a 90 MHz Pentium® based LINUX operating system computer at the extreme right.



5.1. TNC OUTPUT

Figure 5.1-1 shows sample output from the TNC. It is the job of the XNET parser to analyze the string and extract the desired fields. It should be noted that the TNC is an intelligent device with many user programmable parameters. We will discuss some of those parameters in more detail in Appendix B.

```
W9QVE*>W9IF (UA)
W9QVE*>W9IF (UA,F)
W9QVE*>W9IF [I;0,0]:[MSYS-1.17-HIS$]
W9QVE*>W9IF [I;0,1]:Hello ?, Welcome to W9QVE's MSYS BBS in Argonne, Il
W9QVE*>W9IF [I;0,2]:Welcome! As a new user you will see this information.
W9QVE*>W9IF [I;0,3]:you will go directly to the BBS command prompt.
W9QVE*>W9IF [I;0,4]:The home bbs you give must NOT be a personal (built into a TNC) BBS
W9QVE*>W9IF [I;0,5]:but rather a well known full service bbs that does mail forwarding. If you
W9QVE*>W9IF [I;0,6]:haven't picked a homebbs yet, you are welcome to use this one. Just enter
W9QVE*>W9IF [I;0,7]:its call with the NH command.
W9QVE*>W9IF [I;0,0]:VE Testing - Next Monthly testing will be on
W9QVE*>W9IF [I;0,1]:July 19, 1994 at 9400 S. Garfield, Burr Ridge, Ill
W9QVE*>W9IF [I;0,2]:for Further Info call Deni W9DS at 708-986-0061 before 3pm on 17th.
W9QVE*>W9IF [I;0,3]:*****
W9QVE*>W9IF [I;0,4]:Please use N command to enter your name
W9QVE*>W9IF [I;0,5]:Please use NQ command to enter your QTH (City and state)
```

6. PROJECT TESTING, RESULTS AND EXTENSIONS

Success or failure of a project can be measured in several ways. Among Total Quality Management (TQM) engineers and managers the rule is:

- Say what you are going to do,
- Do what you say, and
- Have the documentation to prove it.

Using these axioms, **XNET is a success!**

Say what you are going to do. The project proposal was previously submitted and accepted. The proposal outlined in significant detail the purpose of the project, its goals, and initial design specifications including sketches of the screen windows and the information that was to be displayed.

Do what you say. XNET was developed according to the proposal. All major packet requirements described in the original proposal were implemented.

Have the documentation to prove it. The original proposal, this Master's project paper, the Operating Manual, and the source code represent a documentation level sufficient for understanding, using, and maintaining XNET.

6.1 VARIANCES

If one measures success by merely looking at the completion of the project by the due date, then **XNET was a complete success**. However, if one must select an area of departure from the original proposal, it would be the in-between milestones expected along the way to completion. The original intent as outlined in the proposal was to supply completed deliverables on specific dates. The Tcl/Tk learning curve was not steep, however, there were small impediments using the myriad unique characteristics of the language. Developing algorithms to meet the requirements of the design was particularly time consuming.

A large effort was devoted to understanding the *addinput* command, and then implementing it for both the simulation files (i.e., *sim1.sh* through *sim10.sh*) and the hardware serial port. *Aging* packets also presented a major difficulty, not in the language, but developing the algorithm. The most time intensive task was the development of the algorithm to collect the source and destination nodes to allow the MAP window to be drawn.

Therefore, the schedule was not, and could not have been followed in a serial manner as originally expected. When difficulties were encountered however, the logical decision was made to work on portions so as to keep the project moving. The result, **XNET was completed on time!**

6.2 TESTING

In a well managed commercial software project, more time is devoted to testing the software than to writing the code, or at least it should be. Clearly, testing requires a major effort all by itself. At this time, most XNET testing was performed by the programmer and the project's supervisor. Both user, and operational testing was performed. The operational testing was performed with significant testing tools such as *PacketTracker* and a conventional terminal.

PacketTracker, a program written for the Apple Macintosh performs similar functions as XNET. The author was fortunate to have during the entire development effort, both XNET and *PacketTracker* running side by side. During the course of the project, much time was spent comparing the results of the two programs to insure correctness. Disparities between the two programs were carefully monitored and resolved.

The second significant tool the author had to test XNET was the Apple Macintosh itself. The Macintosh was turned into a TNC by using a terminal emulation program. In this way, the tester **forced** packet information to XNET to insure that it behaved properly.

The third and final manner of testing was the pre-recorded simulations. The 10 simulations that are packaged with XNET form an excellent testing tool. An examination of the simulations packets and a comparison with XNET results forms a rich source of testing data. In addition, since these are text files, the files can be, and were modified to **force** known packet strings to XNET.

In summary, despite the lack of a rigorous software test procedure and a test team, XNET has had significant testing. XNET is expected to be released to user groups in the near future for further testing.

6.3 XNET USE

At this writing, XNET has had limited use outside the use and testing of the author and the project's supervisor. As much as possible, XNET has been running and collecting data from local (Chicago and Dallas) packet networks. Eventually the expectation is to release XNET to the Tcl/Tk and amateur radio Internet archives for distribution and use.

6.4 PROJECT EXTENSIONS

Most, if not all projects, especially research projects, can be never ending. XNET is no exception. It is for this very reason that requirements are developed at the beginning of the project. They serve as a way to confirm the end of the project. The project ends when requirements are met. However, a project such as XNET can be expanded. Below are two possible future extensions.

6.4.1. TCP/IP VERSION OF XNET

During the past five years, TCP/IP has become an alternative to AX.25. Therefore, a project of great interest is the development of an XNET for TCP/IP networks which currently co-exist on the same wireless network.

6.4.2. EXAMINE RAW BIT LEVEL DATA

Since data coming to XNET comes from the TNC, some information is lost and/or filtered by the TNC. For example, all packets sent to XNET from the TNC have a valid FCS (frame checksum), since the TNC checks for a valid FCS and will not pass an invalid packet to XNET. Therefore XNET has no way of keeping statistics regarding collisions, communication errors, and other similar parameters that the TNC filters out. Limitations such as these can be cured by placing the TNC in the KISS mode thereby causing the TNC to pass raw bit level data. XNET would then be responsible for parsing the packets, error checking, and numerous other functions. The resulting program would provide additional useful information, and provide an excellent environment for an extension to XNET.

7. CONCLUSION

When one completes a significant project such as XNET, there cannot help but have been a tremendous learning experience. The project covered many areas, of which Tcl/Tk represented a significant portion. Other areas included the opportunity to work closely with UNIX. Researching the AX.25 protocol and AX.25 networks as well as networks in general, were all necessary parts of the project. Using the Internet, the Tcl/Tk newsgroup and communicating with experts from around the world was also a rich resource of knowledge and experience from which to draw.

When the project began, the author had never used Tcl/Tk, so a large effort went into gaining sufficient expertise to allow code writing to begin. When sufficient Tcl/Tk experience was gained, much time was then spent on developing the design and software algorithms. Of particular complexity was the *aging* of the packets.

There is little doubt that GUIs are here to stay. Tcl/Tk and similar GUI related languages are and will continue to be in demand. The ability to work with UNIX was and continues to be a valuable experience.

This paper described XNET, a GUI based network analyzer for wireless AX.25 packet radio networks. Wireless packet radio networks, like other networks are complex systems which may be modeled but should also be analyzed if one wishes to assure proper operation of the network and better understand the network's operation. XNET provides a tool for this vital purpose.

8. ANNOTATED BIBLIOGRAPHY AND REFERENCES

- 8.1. Black, Uyles, "*X.25 and Related Protocols*," IEEE Computer Society Press, Los Alamitos, California, 1991.

The original X.25 specifications are difficult to understand, even to the experienced communications engineer. This is an excellent book that expands upon, and makes sense of the X.25 protocol. X.25 companion standards X.1, X.2, X.10, X.96, X.121, and many others are discussed.

- 8.2. Fox, Terry L, "*AX.25 Amateur Packet-Radio Link-Layer Protocol*," Version 2.0 October 1984, American Radio Relay league, Inc. Newington, Connecticut, October 1984.

This is an extremely important document. While it is less than 40 pages, it provides the specifications for AX.25.

- 8.3. Halsall, Fred, "*Data Communications, Computer Networks and Open Systems*," Addison-Wesley, New York, New York, 1992.

This is a detailed technical book written for engineering students, practicing communication engineers and those wishing a detailed knowledge of computer networking and communications. The book covers virtually all aspects of data communications including all aspects of the seven layer OSI model. The text covers both data and telecommunications concepts.

- 8.4. Horzepa, Stan, "*Your Gateway to Packet Radio*," American Radio Relay league, Inc. Newington, Connecticut, 1989.

This book is well suited to the amateur radio operator who is interested in constructing an operational packet radio system. Included is AX.25 protocol information to allow the reader to understand how packet radio works.

- 8.5. Ingram, Dave, "*How to Get Started In Packet Radio*," The National Amateur Radio Association, Redmond, WA, 1992.

This is a beginner's book for the amateur radio operator who is interested in putting together a complete operational packet radio system. The book provides specific examples, products, and schematics which serve as detailed information to allow a novice to assemble a packet system.

- 8.6. Libes, Done, "*Exploring Expect*," O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

The book is a tutorial for the computer language Expect. Expect is a Tcl-based Toolkit for automating interactive programs. Expect provides easy ways to develop interactive programs. Expect is a new type of tool that provides a means to develop applications that were not considered in the past.

- 8.7. Ousterhout, John K, "*Tcl and the Tk Toolkit*," Addison-Wesley, Reading, Massachusetts, 1994.

This is the authoritative source for Tcl and Tk, written by the creator of the language. The book is written for an experienced programmer. Knowledge of no specific language is required, however, a knowledge of C is beneficial since the book gives several examples of how one can write Tcl applications in C.

- 8.8. Ousterhout, John K, "*Tcl: An Embeddable Command Language*," Proceedings USENIX Winter Conference, January 1990, pp. 133-146.

This is one of the first papers published by the author of Tcl/Tk. It is an early paper describing the features of Tcl.

- 8.9. Ousterhout, John K, "*Tcl An X11 Toolkit Based on the Tcl Language*," Proceedings USENIX Winter Conference, January 1991.

The author of Tcl/Tk discusses Tk, an X11 toolkit in this paper. Tk performs functions similar to Xt.

- 8.10. Ousterhout, John K, "*Tcl / Tk Engineering Manual*," Sun Microsystems, Inc., Unpublished, available via the Internet.

This paper is an engineering manual for persons developing C code for Tcl, Tk and their extensions. It describes an extensive set of conventions for writing code and the associated test scripts.

- 8.11. Rose, Marshall T., "*The Open Book, A Practical Perspective on OSI*," Prentice Hall, Englewood Cliffs, New Jersey, 1990.

This book covers numerous aspects of the OSI model. It explains what OSI is, how OSI is implemented, and how OSI is used in networks. It provides a clear explanation of how computers communicate with each other. Standards are necessary and the OSI model provides the necessary framework.

- 8.12. Sah, Adam, and Jon Blow, "*A Compiler for the Tcl Language*," Unpublished, available via the Internet, May 24, 1993.

This paper is a description of an effort to develop a compiler for Tcl. Emphasis is placed on the difficulties of developing such a product for Tcl and the progress that authors have made in that effort. An early version of the compiler shows approximately ten times the performance of the existing Tcl interpreter.

- 8.13. Wade, Ian, "*NOSIntro, TCP/IP over Packet Radio*," American Radio Relay league, Inc. Newington, Connecticut, 1992.

This publication provides information to allow an amateur radio operator to assemble a TCP/IP packet radio system. The book is dedicated to NOS (Network Operating System) written by Phil Karn, KA9Q, which uses TCP/IP at the middle layer protocols.

- 8.14. The following list summarizes the HDLC specifications published by ISO, the titles have been shortened:

HDLC frame structure and addendum 3309 (two documents)

HDLS elements of procedures 4335 (three documents)

Multilink procedures (MLP) 7448 (one document)

HDLC-LAPB-compatible link-control procedures 7776 (one document)

HDLC consolidation of classes procedures; list of standard HDLC protocols that use HDLC procedures. 7809 (five documents):

HDLC-balanced, link-address information 8885 (one document): HDLC-additional specifications describing use of an XID frame and multilink operations. 8471 (one document):

- 8.15. The following are additional recommendations and standards.

American Telephone and Telegraph Company, "Operating Systems Network Communications Protocol Specification BX.25 — Issue 2."

ANSI X.366 "Advanced Data Communication Control Procedure," (ADCCP).

CCITT Recommendation X.25, "Interface between Data terminal Equipment (DTE) and Data-Circuit Terminating Equipment (DCE) for Terminals Operating in the Packet Mode on Public Data Networks."

ISO 3309, "Data communication — High-Level Data Link Control Procedures — Frame Structure."

ISO 7205, "Reference Model of Open Systems Architecture."

ISO 7776, "Information Processing Systems — Data Communications — 2nd DP 7776 Revised — Description of the 1984 X.25 LAPB — Compatible DTE Data Link Procedures."

- 8.16. The following *USENET* Newsgroup on the Internet is devoted to *Tcl/Tk* programming.

`comp.language.tcl`

- 8.17. The following World Wide Web (*WWW*) server on the *Internet* is devoted to *Tcl/Tk* programming.

`http://playground.sun.com/~ouster`

9. TRADEMARKS

Apple® is the registered trademark of Apple Computer, Inc.

AT&T® is the registered trademark of American Telephone and Telegraph Company.

IBM® is the registered trademark of International Business Machines.

Intel® is the registered trademark of Intel, Corporation.

MacOS® is the registered trademark of Apple Computer, Inc.

Motorola® is the registered trademark of Motorola Corporation.

Pentium® is the registered trademark of Intel, Corporation.

UNIX® is the registered trademark of Novell Inc.

Windows® is the registered trademark of Microsoft Corporation.

APPENDIX A — AX.25 PROTOCOL

A.1. INTRODUCTION

For communication, digital or otherwise, there is a need for a common language, a standard, specifically an agreed upon protocol. XNET has been designed specifically to analyze AX.25 networks. To better understand XNET, a working knowledge of the AX.25 protocol is crucial.

Before proceeding with a description of the AX.25 protocol, a brief explanation of the International Standards Organization (ISO) Open Systems Interconnections (OSI) model is in order. There are seven layers which comprise the OSI model. The *application* layer is at the top followed by the *presentation* and *session* layers. These top three layers represent the higher application-oriented layers. The bottom three are the *network*, *link*, and *physical* layers and these three represent the underlying network-oriented protocol layers. The upper and lower three layers are interfaced by the *transport* layer.

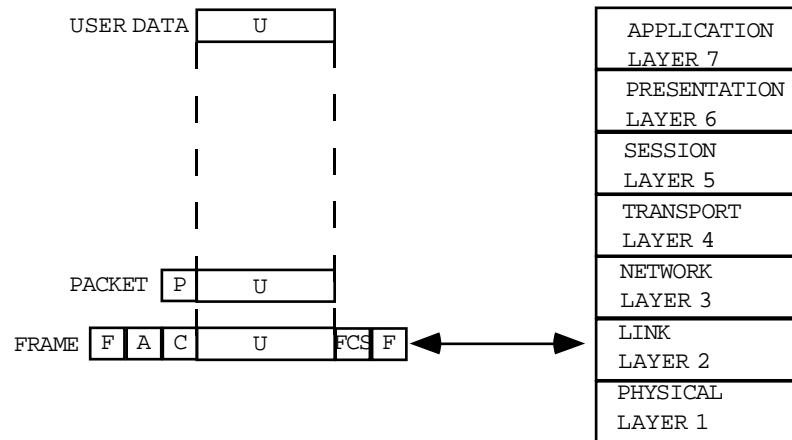
For our purposes, the RF transceiver represents the *physical* layer, (i.e., wireless communication), and the TNC functions at the *data* link layer. The information from the TNC is passed to XNET which monitors information at the *network* layer.

The data link layer is responsible for providing reliable transfer of data over a single link. It provides such functions as framing, error control, synchronization, and flow control. A commercial data link layer protocol and international standard is X.25 and is the basis of the AX.25 which is a modified version of X.25 developed to meet the unique needs of amateur packet radio transmission.

Figure A.1-1 shows how user information is assembled to form a packet and then framed before it is sent over the network.

The International Telegraph and Telephone Consultative Committee (CCITT) has accepted the X-series recommendations as an international standards. X.25 has become a worldwide standard for data communication. The X.25 protocol is a subset of the general

purpose full duplex High Level Data Link Control (HDLC) bit-oriented protocol known as Link Access Procedure, Balanced (LAPB).



The X.25 recommendations were developed in the early 1970s with the first significant draft released in 1976. Following the 1976 release the committee produced three additional recommendations, X.3, X.28, and X.29 to support asynchronous terminals. The first version of X.25 reaching international acceptance was released in 1980. Updates were released in 1984 and 1988 with the latter being the presently accepted release.

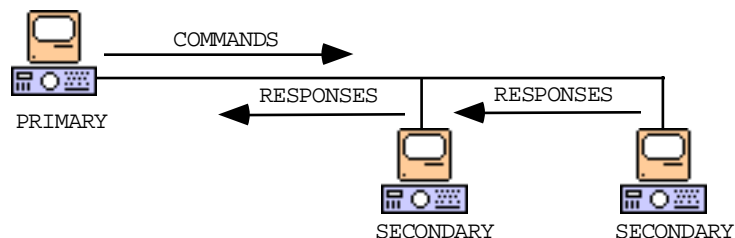
Realizing that most terminals are asynchronous and require additional hardware to allow network communication, a Packet Assembler Disassembler (PAD) was proposed to meet this need. The PAD is an integral part of the hardware required for amateur radio packet communication. However, the term used in amateur radio is a Terminal Node Controller. It performs virtually the same function as a PAD used in commercial X.25 applications.

Over the years there have been several link-layer protocols suggested for amateur packet radio. The first link-layer protocol to achieve widespread use was created by the Vancouver (BC) Amateur Digital Communications Group (VADCG). It was based on the IBM Synchronous Data Link Control (SDLC). The problem with this early version was the addressing scheme which limited nodes to approximately 250. This was

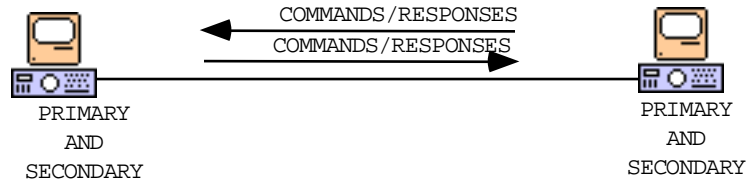
acceptable for a Local Area Network for local club members, but was certainly unacceptable for worldwide communication with millions of nodes.

In 1982 the Amateur Radio Research and Development Corporation (AMRAD) recommended a variant of CCITT X.25 level 2 link access protocol balanced. With the help of amateur radio operators at AT&T Bell Laboratories, who had developed a variant, BX.25 ("B" for Bell), a new protocol AX.25 ("A" for amateur) was developed. The differences between the two recommendations are small. AX.25, in principle follows the CCITT X.25 recommendations, a major departure is the extended address field to accommodate digipeaters (repeaters) that may be used to extended the useful distance of a wireless network. It also follows the principles of the CCITT Q.921 which documents the Link Access Procedure D-channel, (LAPD) in the use of multiple links, distinguished by the address field, on a single shared channel.

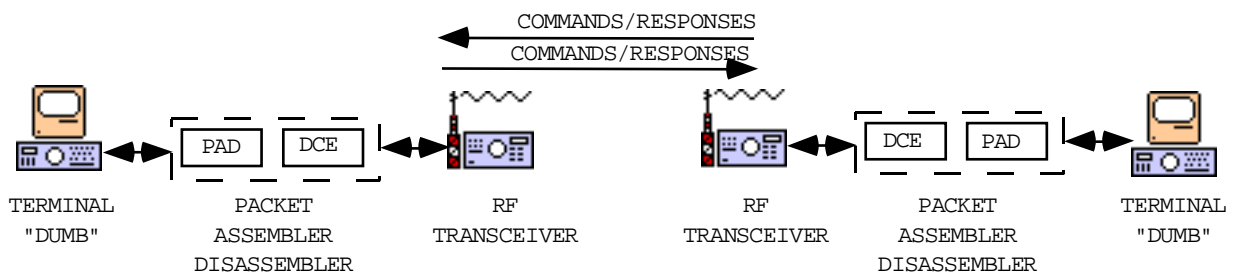
The term *balanced* is used to indicate full duplex point to point communication. Take for example, a simple point to point primary and secondary Data Terminal Equipment (DTE) as shown in Figure A.1-2. The primary, acting as a master, is responsible for all communication and the secondary acts as a slave and cannot initiate commands. If there are multiple slaves as shown in the figure, the master keeps separate sessions for each slave.



In the balanced network in Figure A.2-2, any node on the network is able to send and receive commands and responses.



In amateur radio packet communication however, the simple point to point network is more accurately shown in Figure A.1-3. A terminal (most often a computer emulating a terminal) is connected to a PAD / Modem unit and then to a transceiver. The PAD / Modem combination is more generally referred in amateur radio as the terminal node controller. Each point on the network is called a node.



A.2. ASYNCHRONOUS TRANSMISSIONS

All digital forms of communication require some form of synchronization between the sender and receiver. Transmission types fall into two categories: asynchronous and synchronous. In asynchronous data, a start bit precedes each character. This start bit is used to notify the receiver that a character is beginning and to prepare the receiver for accepting the subsequent data stream. After the bits of the data stream are accepted, one or more stop bits are appended to the end. The stop bit acts as a means of *framing* the data and returning the transmission line level in preparation for the next data stream. This

method is referred to as asynchronous due to the absence of continuous synchronization between the sender and receiver.

The advantage of this method lies in its simplicity. Virtually all computer terminals and low speed devices use asynchronous communication. The price one pays for this simplicity is *throughput*. Every character sent contains a start and stop bit which is wasted overhead and thus not the useful information. Thus to send an 8 bit ASCII character, two additional bits (start and stop) are required which represents 20% (2 overhead bits / 10 total bits) of the total data stream being communication overhead.

A.3. SYNCHRONOUS TRANSMISSIONS

Synchronous transmissions do not surround each character with a stop and start bit, but rather frames the user's data with a specific bit pattern. There are predominately three ways to implement synchronous transmissions: *character-oriented*, *count-oriented*, and *bit-oriented* protocols.

A.3.1. CHARACTER SYNCHRONIZATION

Character synchronization uses the American Standard Code for Interface Interchange (ASCII) standard *synchronization* character (SYN \$16)⁷ to mark the beginning of transmission. The *end of transmission* character (EOT \$04) marks data stream completion. Character-oriented synchronous data-link control relies on specific characters within the ASCII, Extended Binary Coded Decimal Interchange Code (EBCDIC) code set to attain synchronization. Table A3.1-1. shows characters used for this purpose.

SOH	\$01	00000001	Start of header
STX	\$02	00000010	Start of text
ETX	\$03	00000011	End of text
EOT	\$04	00000100	End of transmission
ENQ	\$05	00000101	Enquiry

⁷ This paper uses the dollar sign (\$) to indicate a hexadecimal value. This standard is widely used by Motorola® and is equivalent to the ANSI C two character standard "0x" to indicate a hexadecimal value.

ACK	\$06	00000110	Acknowledge
DLE	\$10	00010000	Data link escape
NAK	\$15	00010101	Negative acknowledge
SYN	\$16	00010110	Synchronous idle
ETB	\$17	00010111	End of transmission block

A.3.2. COUNT SYNCHRONIZATION

The problem with the character-oriented synchronization discussed above lies predominately in its inability to transmit legitimate binary characters that happen to have the same bit patterns as the special synchronization characters. For this reason, character-oriented protocols are being replaced by *count* and *bit* oriented protocols. The count-oriented protocols (also called block protocols) inserts a "count field" at the beginning of the frame. This count is then used by the receiver to specify the length (count) of the data stream that follows and to accept all characters irrespective of the bit pattern. In this way, user data transparency is achieved (i.e., any character may be transmitted).

A.3.3. BIT SYNCHRONIZATION

Bit-oriented data-link control protocols have been used since the 1970s. This technique for data communication is becoming more widely accepted and forms the basis for the X.25 LAPB and AX.25 protocols.

A unique bit pattern, referred to as a *flag* is used. For both X.25 and AX.25, the bit pattern that marks the beginning and end of data is 01111110 (\$7E). These bit patterns are used to inform the receiver that a data stream has begun or is ending. The protocol is in many ways identical to the character-oriented protocol previously discussed. The main difference lies in the use of a flag to mark the beginning and end of transmission rather than the verbose use of a dozen or more control characters to serve as markers.

A.4. BIT STUFFING

Since AX.25 uses bit-oriented data-link control protocol, and all frames begin and end with a flag consisting of a string that could exist virtually anywhere else in the frame, the astute reader will question the wisdom of using a flag to mark the beginning and end of a frame. After all, we have the same problem we had with character-oriented protocols, they suffer from a lack of user data transparency.

To avoid this very real problem, the sending station monitors the data being sent. When a string of five contiguous 1 bits appear, the sending station inserts a 0 bit immediately after the fifth 1 bit. At the receiving end, when five 1 bits are received, the 0 that occurs after the five 1s, is discarded. This additional wasted bit slows the transmission of data, however, it is insignificant and a small price to pay for high reliability data and compares more favorably than other techniques.

The following two examples help illustrate. The first example shows how a character \$7F is modified and returned to its original value. The second example, illustrates that the flag itself must be converted each time it is sent.

```
01111111    (real data)
011111011    (actually sent, the underscore marks the bit stuffing)
01111111    (after receiver has discarded zero)
```

Example A.4-1 \$7F bit stuffed and retrieved

```
01111110    (frame flag)
011111010    (actually sent, the underscore marks the bit stuffing)
01111110    (after receiver has discarded zero)
```

Example A.4-2 \$7E, the CCITT flag, bit stuffed and retrieved

A.5. LINK LAYER

The link layer, also called level 2 due to its position in the ISO OSI model, is responsible for providing the network layer (level 3) with reliable packets. This is accomplished by sending small blocks of data called frames. There are three classes or types of frames: *unnumbered*, *information*, and *supervisory*.

A.6. INFORMATION FRAME

Flag	01111110	all frames begin and end with this flag.
Address	112 to 560 bits	this field is used to identify both the source of the frame and the destination.
Control	N(R), P, N(S), 0	This 8 bit field contains four piece of information. N(R) occupies bits 7, 6, and 5. These 3 bits (8 possibilities) contains the receive sequence number. P occupies bit 4. The P bit (poll) is used in the information frame to request an immediate reply to a frame. The reply to this poll is indicated by setting the response (final) bit in the appropriate frame. N(S) occupies bits 3, 2, and 1. These 3 bits (8 possibilities) contains the sending sequence number. bit 0 has a value of 0 which denotes that the frame is an information frame.
PID	8 bits	Protocol Identifier (PID) field appears only in information fields. It identifies the type of layer 3 (Network / Packet layer of OSI model) protocol is used. xx01xxxx AX.25 layer 3 implemented. xx10xxxx AX.25 layer 3 implemented. 11001100 Internet Protocol datagram layer 3 implemented. 11001101 Address resolution protocol layer 3 implemented. 11110000 No layer 3 implemented. 11111111 Escape character. Next octet contains more Level 3 protocol information.
INFO	N * 8 bits	This is the "real stuff". Virtually everything else that we have discussed is overhead, necessary for the passing of this information in a reliable (error free) manner. This field, which may be up to 256 octets long, contains the information to pass to layer 3.
FCS	16 bits	Frame Sequence Check. It is calculated by the sender and receiver. It is used to assure that corrupt frames are rejected. It is computed in accordance with ISO 3309 (HDLC) recommendations. It is interesting to note, that transmission of bits in a field are transmitted beginning with the least significant bit with the exception of the FCS field. For the FCS field, the most-significant field is transmitted first.
Flag	01111110	all frames begin and end with this flag

The information frame (I-frame) contains the information that we wish to transmit. It begins and ends with the *flag field* as all frames do. The *address field* containing the source and destination is included and, as previously stated, is one of the few departures from the standard X.25 recommendation. The *control field* supplies critical information that assures error control and also provides the means of identifying the frame. In this example, bit 0 (the Least Significant Bit) is 0 and flags the receiver that this is an *information frame*. The *PID field* supplies additional information about the frame. The *info field* is the user data that needs to be transmitted. The *FCS field* consists of two octets, a detailed description of this field is provided later in this section.

Octet	ASCII	Hex Data
Flag		7E
Address	K	96
Address	8	70
Address	M	9A
Address	M	9A
Address	O	9E
Address	space	40
Address	SSID	E0
Address	W	AE
Address	B	84
Address	4	68
Address	J	94
Address	F	8C
Address	I	92
Address	SSID	61
Control	I	3E
PID	none	F0
FCS	part 1	HH
FCS	part 2	HH
Flag		7E

Table A.6-2 shows a typical “real” information frame. Only the FCS field which is computed by the cyclical redundancy check (CRC16) is not shown. Within the address field, the secondary station identifier (SSID) is shown. The SSID sub-field allows an amateur radio operator to have more than one packet radio station operating under the same call sign.

A.7. SUPERVISORY FRAME

The supervisory frames (S frames) provide supervisory link control such as acknowledging or requesting re-transmission of I frames, and link-level window control. The frame begins and ends with the *flag field*. The *address field* containing the source and destination is included. The *control field* provides error control and identifies the frame. In this example, two bits are allocated to identifying the frame, bits 1, and 0. To identify a supervisory frame, the bits are set to 1 and 0 respectively. The *PID field* supplies additional information about the frame; see the field below for details. The *FCS field* consists of two octets.

Flag	01111110	all frames begin and end with this flag.
Address	112 to 560 bits	this field is used to identify both the source of the frame and the destination.
Control	N(R), P, SS, 0, 1	This 8 bit field contains four piece of information. N(R) occupies bits 7, 6, and 5. These 3 bits (8 possibilities) contains the receive sequence number. P occupies bit 4. The P bit (poll) is used in the information frame to request an immediate reply to a frame. The reply to this poll is indicated by setting the response (final) bit in the appropriate frame. SS occupies bits 3 and 2. 0 0 Receive Ready RR 0 1 Receive Not ready RNR 1 0 Reject REJ 1 1 Not Used bits 1 and 0 have values of 0 and 1 respectively and denote that the frame is a supervisory frame.
FCS	16 bits	Frame Sequence Check. It is calculated by the sender and receiver. It is used to assure that corrupt frames are rejected. It is computed in accordance with ISO 3309 (HDLC) recommendations. It is interesting to note, that transmission of bits in a field are transmitted beginning with the least significant bit with the exception of the FCS field. For the FCS field, the most-significant field is transmitted first.
Flag	01111110	all frames begin and end with this flag

A.8. UNNUMBERED FRAME

These frames provide functions such as setting up the initial connection and the final disconnect. Receive and send sequence numbers are not included in this frame, hence the derivation of its name "unnumbered." This is again a slight departure from the X.25 standard. This field is included to meet the unique requirements of amateur radio. The frame begins and ends with the *flag field*. The *address field* containing the source and destination is included. The *control field* provides error control and identifies the frame. In this example, two bits are allocated to identifying the frame, bits 1, and 0. To identify an unnumbered frame, the bits are both set to 1. The *flag field* terminates the frame.

Flag	01111110	all frames begin and end with this flag.
Address	112 to 560 bits	this field is used to identify both the source of the frame and the destination.
Control	MMM, P, MM, 1, 1	This 8 bit field contains four piece of information MMM occupies bits 7, 6, and 5. Modifier bits. P/F occupies bit 4. The P bit (poll) is used in the information frame to request an immediate reply to a frame. The reply to this poll is indicated by setting the response (final) bit in the appropriate frame. MM occupies bits 3 and 2. Modifier bits. bits 1 and 0 both have values of 1 which denotes that the frame is a unnumbered frame.
FCS	16 bits	Frame Sequence Check. It is calculated by the sender and receiver. It is used to assure that corrupt frames are rejected. It is computed in accordance with ISO 3309 (HDLC) recommendations. It is interesting to note, that transmission of bits in a field are transmitted beginning with the least significant bit with the exception of the FCS field. For the FCS field, the most-significant field is transmitted first.
Flag	01111110	all frames begin and end with this flag

A.9. RELIABILITY AND THE FCS FIELD

We have stated that the link layer is responsible for providing the network layer (level 3) with reliable packets. Just how reliable is reliable and how do we assure reliability? It should be noted our definition of reliability means that the data transmitted is accurate irrespective of the number of times a packet must be transmitted. In our definition, we are not concerned with how many times a packet has been sent before it is accepted, we are concerned only with *undetected* errors, errors that the system accepts as being true while actually being incorrect.

The *frame check sequence* (FCS) field is created by a *cyclical redundancy check* (CRC) of the frame. Several polynomials are widely used for the computation of the FCS. Some of the more popular polynomials are shown. Both X.25 and AX.25 use the CRC-CCITT polynomial. Other popular polynomials are shown for the sake of completeness.

CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC16	$x^{16} + x^{15} + x^2 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{16} + x^{12} + x^{11} + x^{16} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^5 + x + 1$

Table A.9-1 Typical CRC Polynomials, AX.25 Uses the CCITT standard⁸

Also provided for the sake of completeness in Table A.9-1 is the undetected bit error rates that can be expected using the CRC-CCITT standard. Note that as the bit error rate increases, the probability of an undetected error also increases. No statistics are available, however, the bit error rate for a wireless network such as the AX.25 discussed, are substantially higher than hardwired networks.

⁸ Halsall, Fred, "Data Communications, Computer Networks and Open Systems," Addison-Wesley, New York, New York, 1992, page 110.

Bits in frame	BIT ERR RATE				
	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}
100	1×10^{-8}	1×10^{-8}	1×10^{-10}	1×10^{-11}	1×10^{-12}
300	9×10^{-8}	7×10^{-9}	7×10^{-10}	7×10^{-11}	7×10^{-12}
1000	7×10^{-9}	5×10^{-10}	4×10^{-11}	4×10^{-12}	4×10^{-13}
3000		2×10^{-11}	1×10^{-12}	1×10^{-13}	1×10^{-13}
10000		1×10^{-10}	2×10^{-11}	2×10^{-12}	2×10^{-13}
30000		$\sim 10^{-5}$	$\sim 10^{-6}$	$\sim 10^{-7}$	$\sim 10^{-8}$

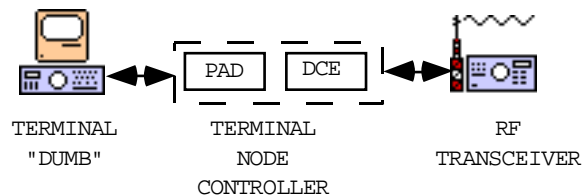
Table A.9-1 Undetected Bit Error Rate from Errors within Frames⁹**APPENDIX B — TERMINAL NODE CONTROLLER**

⁹ Black, Uyles, "X.25 and Related Protocols," IEEE Computer Society Press, Los Alamitos, California, 1991, page 61.

B.1. THE PURPOSE OF THE TNC

We have discussed in detail the AX.25 protocol. In this section, the TNC will be discussed with emphasis on understanding its function and user programmable parameters.

A commercial Packet Assembler Disassembler is a hardware device which is responsible for accepting ASCII characters and assembling X.25 packets. For amateur radio use, the TNC serves this purpose. Actually, a TNC combines the functions of a PAD and a *modulator and demodulator* (Modem). The TNC accepts ASCII characters, assembles AX.25 packets, and produces audible tones that are sent to the transceiver. The output of the TNC is audible signals similar to FAX signals. The reverse task of demodulating the tones and disassembling AX.25 protocol frames and returning them to ASCII text characters is performed at the other end. The dashed box shown in Figure B.1-1 shows a typical amateur radio node controller consisting of both a PAD and a modem.



A complete description of all the TNC parameters is outside the scope of this paper, however a few will be discussed in detail. A complete list of all parameters is shown for completeness.

B.1. ASYNCHRONOUS PORT PARAMETERS

These parameters set the communications between the terminal and the TNC. For example, *transmitting baud (TBAUD)* specifies that the TNC is expecting 9600 baud and

no parity. The user must also set the terminal with identical parameters to assure proper communication. This parameter is set from the PREFS window of XNET. The *host baud* (*HBAUD*) parameter sets the baud rate that is actually output from the TNC. This is what we have referred to as the radio baud rate. Recall that the radio baud rate is specified in the PREFS window, however it is used only to the computation of network utilization which is plotted in the GRAPH window.

```

8Bitconv  ON
ACRDisp   0
AFilter   OFF
ALFDisp   ON
AUTOBaud  OFF
AWlen     8
BBSmsgs   OFF
CASedisp  0 (as is)
DCdconn   OFF
Echo      OFF
EScape    OFF
Flow      OFF
ILfpack   OFF
NUcr      OFF
NULf      OFF
NULLs     0
PARity    0 (none)
TBaud     9600
TRFlow    OFF
TXFlow    OFF
XFlow     OFF

```

B.2. SPECIAL CHARACTERS

These parameters set special characters for the TNC. Most are unique to AX.25, however, *XON* and *XOFF* are universally recognized standards for flow control. Note however, that we are referring to flow control between the computer system and the TNC. Flow control over the wireless network is controlled elsewhere. The remaining characters do not directly relate to XNET and are not discussed further.

```

BKondel   ON
CANline   $18 (CTRL-X)
CANPac    $19 (CTRL-Y)
CHCall    OFF
CHDouble  OFF
CHSwitch  $00
COMmand   $03 (CTRL-C)

```

CWid	\$06 (CTRL-F)
DElete	OFF
ERrchar	\$5F (_)
HEReis	\$02 (CTRL-B)
PASs	\$16 (CTRL-V)
RECeive	\$04 (CTRL-D)
REDispla	\$12 (CTRL-R)
SEndpac	\$0D (CTRL-M)
STArt	\$11 (CTRL-Q)
STOp	\$13 (CTRL-S)
TIme	\$14 (CTRL-T)
XOff	\$13 (CTRL-S)
XON	\$11 (CTRL-Q)

B.3. IDENTIFICATION PARAMETERS

These parameters are predominately amenities to allow easy operation and identification of the station. Take for example, the *CText* parameter. When a connection is made to the TNC, this text is automatically transmitted provided that TNC parameter *CMSG* is *ON*. Note that another parameter *Beacon* is available to allow the TNC to let other operating TNCs on the network know that the TNC is operational. It may be set to automatically identify itself at pre-defined intervals. Although it is used for a much different purpose than the *HELLO* command used by commercial network bridges, it is nonetheless similar in operation.

Unproto	CQ
AAb	
Beacon	EVERY 0 (00 sec.)
BText	
CBell	OFF
CMSg	ON
CText	Rick, W9IF DeSoto, TX.
CUstom	\$0A15
HIId	ON
HOmebbs	none
MId	0 (00 sec.)
MYAlias	RICK
MYcall	W9IF
MYALTcal	
MYIdent	
MYSelcal	
WRu	OFF

B.4. LINK PARAMETERS

As its name implies, these parameters are available to allow the user to specify link level parameters. The *HBaud* parameter is set here which specifies the baud rate of communication between the two communicating TNCs. This is the radio baud rate that the user must specify in the PREFS window of XNET. Again, it is important to remember that *HBaud* is much different than the *TBaud* parameter which is the TNC to computer baud rate. The *PACLen* parameter specifies the number of user typed characters that are accepted before a packet is automatically assembled. The term automatic is important, since the user may force a packet to be assembled merely by typing at least one character and then typing the ENTER character (not to be confused with the RETURN character).

```

Connect      Link state is: DISCONNECTED
ACRPack      ON
ALFPack      OFF
ALTModem     0
Ax2512v2     ON
CFrom        all
CONMode      CONVERSE
CONPerm      OFF
DFrom        all
FULLdup      OFF
HBaud        1200
MAXframe     4
NEwmode      ON
NOmode       OFF
PACLen       80
PASSAll      OFF
RELink       OFF
REtry        10
SQuelch      OFF
TRies        0
USers        10
Vhf          ON
XMITok.....ON

```

B.5. MONITOR PARAMETERS

These parameters are intended to control the display of information. For example, *CONStamp* and *DAYStamp* control the time and date stamping of packets. The *Monitor*

parameter is a multi-level parameter which allows the TNC to filter out unwanted packet header information that is typically not of interest to the casual listener. The MONITOR parameter is initialized by XNET to 6, which means all available information is passed to XNET.

```
CONStamp  ON
DAYStamp  ON
HEAderln  OFF
MBell     OFF
MBx       none
MCon      6 (seq, P/F + all)
MDigi     ON
MFilter   $80
MFrom     all
Monitor   6 (seq, P/F + all)
MProto    OFF
MRpt      ON
MStamp    ON
MTo       none
MXmit     OFF
TRACe     OFF
WHYnot    OFF
```

B.6. TIMING PARAMETERS

As its name implies, the timing parameters are available to specify critical network timing. For example, the *DWait* parameter is used to specify the length of time (in milliseconds) the TNC will wait after hearing traffic on the network before transmitting.

We have stated in this paper that the purpose of XNET is to monitor and display network information to allow one to understand the operation of the network. With the information ascertained with XNET, the parameters shown can be modified to change the network in an effort to improve network efficiency.

```
ACKprior  OFF
AUdelay   2 (20 msec.)
AXDelay   0 (00 msec.)
AXHang    0 (000 msec.)
CHeck     18 (180 sec.)
CMdtime   10 (1000 msec.)
CPactime  OFF
DWait     16 (160 msec.)
FRack     3 (3 sec.)
PACTime   AFTER 10 (1000 msec.)
PErsist   63
PPersist  ON
RESptime  10 (1000 msec.)
SLovertime 30 (300 msec.)
TXdelay   30 (300 msec.)
```

APPENDIX C — XNET OPERATING MANUAL

APPENDIX D — XNET SOURCE CODE