;login:

THE MAGAZINE OF USENIX & SAGE
June 2003 • volume 28 • number 3

inside:

PROGRAMMING
The Tclsh Spot
by Clif Flynt

the tclsh spot

by Clif Flynt

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of *Tcl/Tk for Real Programmers* and the *TclTutor* instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

clif@cflynt.com

Capturing and Analyzing IP Packets on the AX-4000 Using AxTcl

Previous Tclsh Spot articles described building a program for generating Ethernet packets and doing simple validation tests on them.

This article explains how to use the Spirent/AdTech AX-4000 to capture and analyze packets, report good/bad packet statistics, and generate human-readable descriptions of the packets.

The AX-4000 (*http://www.adtech-inc.com/*) is a configurable piece of hardware that can generate and analyze data packets on four different transmission technologies (IP, ATM, Ethernet, and Frame Relay) simultaneously at speeds up to 10 Gbps.

The AX-4000 can be configured with either a GUI or the AdTech Tcl extension. The GUI is suitable for many purposes, but the Tcl interface is ideal for situations where you want to run the same test multiple times while modifying the test environment.

The basic activity flow for an AdTech Tcl script is:

1. Load the AxTcl extension.
2. Initialize the connection to the AX-4000 controller.
3. Reserve an interface card set.
4. Create a generic interface object attached to the card set.
5. Create an analyzer or generator attached to the interface.
6. Configure the analyzer or generator.
7. Run the test.
8. Analyze the results.

As described in a previous Tclsh Spot article (*;login:*, Vol. 28 #1, February 2003), this script will initialize the AX-4000 by loading the extension, describe the location of the hardware definition (BIOS) files, lock an interface card set, and initialize it:

```
# Define the base directory for the Spirent software.
set base "[file join [file dirname [info script]] ../..]"

# Add directories to search for packages.
lappend auto_path . $base/bios/tcllib/xml $base/tclclib

# Load the extension.
if {[catch {package require ax4kpkg}]} {
      if {[catch {load $base/tclwin/libax4k.dll ax4kpkg}]} {
          catch {load $base/tclclib/libax4k.so ax4kpkg}
      }
}

# Define the hardware bios directory.
ax hwdir $base/bios

# Initialize the AX-4000.
ax init -remote $ipAddress -user clif -conout 1

# Lock a device.
enet lock $deviceID $ipAddress $logicalID

# Create and configure the interface to this card set.
interface create int1 $logicalID -ifmode IPoETHER

int1 set -mode normal -dataRate MBS10
int1 run
```

This code implements steps 1–4 of the basic activity flow and can be copied into each application with little modification.

The AX-4000 can be used as either a packet generator or a packet analyzer, or it can both analyze and generate.

The goal of this software is to validate the previously described packet generator, so we must create an analyzer, but do not need a generator object.

Syntax: analyzer create *Name Device*

analyzer create	Create a new analyzer object.
Name	The name to assign to the new analyzer.
Device	The device to attach this analyzer name to. This is the logical device that was locked in a previous enet lock command.

The analyzer create command follows the paradigm used by Tk to create image objects. It initializes a new analyzer data object, and creates a new command to use to interact with that object.

The analyzer command supports many subcommands, including:

analyzerName set *option value*
: Sets one or more configuration options for this analyzer. Configuration options vary for different analyzer cards.

analyzerName display
Returns a list of the current settings.
analyzerName run
Starts the analyzer running.
analyzerName reset
Stops the analyzer and clears all the statistics the analyzer can gather.
analyzerName stop
Stops the analyzer but does not clear any values.
analyzerName destroy
Destroys the analyzer, freeing it for other use.
analyzerName stats
Returns a set of keyword/value pairs as a list. The exact return depends on the analyzer being used.
analyzerName capture *?subcommand*
Configures the analyzer to capture packets.

The *analyzerName* stats command will return statistics about the packets that have been seen (how many good, how many with certain failures, average size, etc.). By collecting the starting and ending statistics, we can show the statistics of line traffic over a period of time.

The test application must control both the AX-4000 and the script that generates packets. The packet-generating code could be merged into the AX-4000 control script for testing, but it makes more sense to keep the generating code as a separate program, as it will be used in the final application.

There are two ways to execute another program from within a Tcl script. You can open a pipe to the second program with the open command, or, if your script just needs the program to run and is not monitoring the output from the program, you can use the exec command.

The exec command will execute an external program, and whatever output that program would send to stdin or stdout will be returned to the script as the exec return.

Syntax: exec *?-options? arg1 ?arg2...argn?*

exec	Execute arguments in a subprocess.	
-options	The exec command supports two options:	
	-keepnewline	Normally a trailing newline character is deleted from the program output returned by the exec command. If this argument is set, the trailing newline is retained.
	- -	Denotes the last option. All subsequent arguments will be treated as subprocess program names or arguments.
arg	These arguments can be either a program name, a program argument, or a pipeline descriptor.	

This line of code will run the testGen.tcl application at the proper time.

```
exec tclsh /home/clif/IP/PacketMaker/testGen.tcl
```

The packet generator should be started as soon as the AX-4000 has been configured. The AxTcl commands that configure the AX-4000 might take several milliseconds to complete, as the new configuration details are copied from the host machine to the AX-4000.

Since the AxTcl configuration commands return immediately, instead of waiting for all processing to be complete, the Tcl script needs to pause after sending the setup commands before continuing processing.

The Tcl after command provides access to a timer with millisecond resolution. Like other Tcl event-driven programming support, this command allows the script to pause or to schedule a future script.

Syntax: after *milliseconds ?script?*

after	Pause processing of the current script, or schedule a script to be processed in the future.
milliseconds	The number of milliseconds to pause the current processing, or the number of seconds in the future to evaluate another script.
script	If this argument is defined, this script will be evaluated after *milliseconds* have elapsed.

To ensure that the AX-4000 is ready, we must schedule a delay between configuring the AX-4000 and generating packets and between starting and stopping packet collection.

This can be done by simply pausing the application with a command like after 1000, but while the application is paused, it does absolutely nothing: it doesn't check for events, won't respond to keyboard input (except control-c), etc.

For code that might be embedded in an interactive application (I actually put this code inside a GUI test harness), a better paradigm is to schedule events to happen in the future, and use the vwait command to synchronize events.

The vwait command causes the interpreter to stop linear processing of a script until a variable is assigned a new value. While the interpreter is waiting for the variable to change, it continues processing events.

Syntax: vwait *varName*

varName	The variable name to watch. The script following the vwait command will be evaluated after the variable's value is modified.

The next example initializes an analyzer, starts the packet generator running, and collects five seconds' worth of statistics.

```
# Get the starting stats.
set anaStats1 [ana1 stats]

# Start the analyzer.
ana1 run

# Schedule the packet generator to start in 1 second.
# Schedule a break to happen after 5 seconds of operation.

after 1000 {exec tclsh /home/clif/IP/PacketMaker/testGen.tcl}
after 6000 {set stop 1}

set stop 0
vwait stop

# Stop the analyzer, get stats, and destroy it.
set anaStats2 [$analyzer stats]
$analyzer stop
$analyzer destroy

# And unlock the device for the next user.
enet unlock $logicalID

genReport $anaStats1 $anaStats2
```

The *analyzerName* stats command will return the statistics as a list of key/value pairs. The foreach command can be used to step through multiple lists, to collate results. The genReport procedure steps through the list of statistics saved before the application was run, compares it to the results after, and only displays the values that changed.

```
proc genReport {stats1 stats2} {
        foreach {key1 val1} $stats1 {key2 val2} $stats2 {
            if {$val1 != $val2} {
                append report [format "%-30s %12s %12s\n" $key1 $val1 $val2]
            }
        }
    puts $report
}
```

The output from the code above resembles this:

```
ANALYZER STATS
-elapsedTime                    3            5603
-totalPackets                   0             800
-totalPacketBytes               0           67400
-goodPackets                    0             500
-goodPacketBytes                0           38600
-goodDatagramBytes              0           26800
-totalPacketRate                0             212
-goodPacketRate                 0             133
-goodPacketBitRate              0              82
```

```
-goodDatagramBitRate          0.0              56.8
-lineRatePerc                 0.00              1.93
-tcpPackets                   0               300
-tcpRatio                     0.00              0.38
-tcpChecksumErrors            0               200
-udpPackets                   0               300
-udpRatio                     0.00              0.38
-udpChecksumErrors            0               100
-icmpPackets                  0               200
-icmpRatio                    0.00              0.25
-ipPackets                    0.00            800.00
-avgDatagramLength            0                54
-minDatagramLength            0                32
-maxDatagramLength            0                88
-avgPacketLength              0                77
-minPacketLength              0                64
-maxPacketLength              0               106
-substreamCount               0                 4
-substreamErrorCount          0                 2
-filterCount                  1                 5
```

These results show that 800 packets were generated, of which 300 were UDP packets, 200 were ICMP packets, and 300 were TCP packets; 100 of the UDP packets and 200 of the TCP packets had checksum errors.

The generator program was configured to generate the packets described, so this result was expected. While good statistics are a good start, to validate the generator, we should confirm that the checksum errors are all of the expected errors.

To analyze individual packets, we must first capture them. The analyzer is configured to capture packets with the *analyzerName* capture setup command.

Syntax: *analyzerName* capture setup *-option value*

capture setup	Configure the capture rules.
-option value	Option value pairs to configure the capture. Options include:

-segmentsize *Num*
: Specifies the size in blocks of the captured segment. For OC-12c, a block is 2 kilobytes. For OC-48x, a block is 4 kilobytes.

-qualifyEquation *equation*
: Specifies the equation used to qualify the capture. The default is 1 (always true).

-triggerEquation *equation*
: Specifies the equation used to trigger (start) the capture. Default is 1 (start capture immediately).

-triggerPosition *Position*
: Defines when the capture will commence in relation to the trigger event. This will control whether the packet that triggers capturing to start is also captured. May be one of: AFTER_START, BEFORE_CENTER, AFTER_CENTER, or BEFORE_END.

There are many more options that can be used to define the conditions under which you want the AX-4000 to start capturing packets. These are enough for this fairly simple test.

The trigger and qualifying equations support logical operations, with multiple levels of parentheses allowing a script to define arbitrarily complex rules for packet capture. For example, you can define how many bad packets must be seen before capture starts, and which types of packets will be captured after the capturing is triggered.

To test the packet generator, we want to capture all packets for as long as the test is running. Thus, the trigger and qualify equations are trivial, just a TRUE value (1).

After the analyzer is running, the capture can be armed with the *analyzerName* capture arm command. This will cause the capture subsystem to look at (but not necessarily capture) packets. Once the trigger condition is met, packets that match the qualifying rules will be captured.

Syntax: *analyzerName* capture arm *boolean*

capture arm	Turn on (or off) the capture subsystem.
boolean	A boolean value. A True (1) will start the capture subsystem examining packets, and a False (0) will make the subsystem stop examining packets.

This code will initialize the capture subsystem and start capturing packets:

```
# Set up the capture buffer size.
# The qualify and capture equations are both true.
# Stat capturing as soon as the "arm" is set to 1.

ana1 capture setup -segmentSize 1024 \
      -qualifyequation 1 \
      -triggerPosition AFTER_START -triggerequation 1

# Start capturing packets.
ana1 capture arm 1
```

Once a set of packets have been captured, the capture can be turned off by disarming the capture circuit with an *analyzerName* capture arm 0 command.

The next step is to analyze the captured data.

The *analyzerName* capture getdata command retrieves the captured packets from the AX-4000.

Syntax: *analyzerName* capture getdata ?option?

capture getdata	Returns the captured data, or a requested subset of data, either as a list or in an array.
?-array *arrayName*	Stores captured data in an array, instead of returning as a list. Elements are indexed numerically in the order they were received.
start#-end#	The *start* and *end* values are integers representing the first and last packet in the sequence to return.

The data is stored in the array as a keyword/value list, with keywords that include:

-timestamp
  The absolute time in nanoseconds that the packet was received.
-indexNum
  Numeric location of this packet in the packet stream (after trigger).
-eventFlags
  A list of errors in the packet.
-payload
  The packet data (as hex bytes).

With this return, we can check whether or not packets had errors and whether the data conforms to the expected values.

Retrieving the hex values of the packets allows us to do a bit-by-bit comparison between the packets we intended to generate and the data that hit the wire. Doing this by hand is tedious and error-prone. The axdecode command will decode a data packet into human-readable output.

Syntax: axdecode *protocol ?-option value?*

-payload {<Tcl_list>} | -message "<message_string>"} | -dgram <datagram_object> | -hexstring "<hex_string>" | -file <filename>

protocol	The protocol of the message to be decoded. May be one of: PPP, ARP, IP, IS-IS, NHRP, RARP, RTCP, BISUP, B-ICI, or Q.2763.
-option value	Options include:

-payload
: The packet is a numeric list.

-message
: The packet is a string of ASCII characters.

-datagram
: The packet is a datagram object.

-hexstring
: The packet is a string of hexadecimal digits.

-file
: The packet is contained in the specified file.

In order to use the axdecode command, your script must know the type of packet being decoded and convert that value to an appropriate string.

The 12th and 13th bytes of an Ethernet II packet, or 20th and 21st bytes of an 802.3 packet, contain the 16-bit packet type. Converting these bytes to an ASCII string is a simple lookup operation, easily performed with an associative array.

The raw data consists of two hex bytes separated by a space. We normally use a single word for an associative array index. The two hexadecimal bytes could be merged into a single 16-bit value with regsub or the newer string map commands. However, while it is common practice to use a single word for an associative array index, this is not required. Any characters between the parentheses are accepted as an index, so we can easily use two strings as an associative array index.

These two lines of code will define a lookup table and will set the variable type to the appropriate string for an Ethernet II data packet and then display a decoded version of the packet.

```
# Define a lookup table.
array set etherTypes { {08 00} IP {08 06} ARP {80 35} RARP }

# Download captured data to associative array "array1".
ana1 capture getdata -array array1

# Extract first packet from array of captured packets.
array set packet $array1(0)

# Convert bytes 12 and 13 to a string.
set type $etherTypes ([lrange $packet(-payload) 12 13])

# Extract the IP packet from the Ethernet II packet.
set payload [lrange $packet(-payload) 14 end]

# Display the decoded output.
puts [axdecode $type -hexstring $payload]
```

This code would generate output like this for an ICMP Address Mask Request packet with a bad checksum field:

```
-summary {IPv4-<ICMP-<address mask request}
-decode {
<0000, 0032, 01> | ---- packet: IP ----
<0000, 0020, 02>   | ---- packet: IP PDU ----
<0000, 0014, 03>     | ---- packet: IP header ----
<0000, 0001, 03>     | {0100 ....}= 04h [004d] version: IP Internet Protocol
<0000, 0001, 03>     | {.... 0101}= 05h [005d] header length: in 32 bit units, must be 5 or more
<0001, 0001, 04>       | ---- packet: flags ----
<0001, 0001, 04>       | {000. ....}= 00h [000d] precedence field: Routine
<0001, 0001, 04>       | {...0 ....}= 00h [000d] minimize delay: normal delay
<0001, 0001, 04>       | {.... 0...}= 00h [000d] maximize throughput: normal throughput
<0001, 0001, 04>       | {.... .0..}= 00h [000d] maximize reliability: normal reliability
<0001, 0001, 04>       | {.... ..0.}= 00h [000d] minimize monetary cost: normal monetary cost
<0001, 0001, 04>       | {.... ...0}= 00h [000d] unused: must be 0
<0001, 0001, 04>       | ---- end of packet: flags ----
<0002, 0002, 03>     | 00-20 = 20h [032d] total length: in octets include header length, must not
                                         be less than 20
<0004, 0002, 03>     | 00-02 = 02h [002d] identification
<0006, 0001, 03>     | {0... ....}= 00h [000d] reserved: valid
<0006, 0001, 03>     | {.0.. ....}= 00h [000d] Don't Fragment: May Fragment
<0006, 0001, 03>     | {..0. ....}= 00h [000d] More Fragments: Last Fragment
<0006, 0002, 03>     | fragment offset:
<0006, 0001, 03>     | {...0 0000}= 00h [000d] bits 12-8
<0007, 0001, 03>     | {0000 0000}= 00h [000d] bits 7-0
<0006, 0002, 03>     | fragment offset = 00h [000d]: in 64 bits units
<0008, 0001, 03>     | {0010 0000}= 20h [032d] time to live: seconds
<0009, 0001, 03>     | {0000 0001}= 01h [001d] protocol: ICMP (Internet Control Message)
<000A, 0002, 03>     | 07-78 = 778h [1912d] header checksum: checksum is correct
<000C, 0004, 03>     | 192.168.9.2 source ip address
<0010, 0004, 03>     | 192.168.9.17 destination ip address
<0000, 0014, 03>     | ---- end of packet: IP header ----
<0014, 000C, 03>     | ---- packet: IP datagram ----
<0014, 000C, 04>       | ---- packet: ICMP ----
<0014, 0001, 04>       | {0001 0001}= 11h [017d] type: address mask request
<0015, 000B, 05>         | ---- packet: address mask request ----
<0015, 0001, 05>         | {0000 0000}= 00h [000d] code:
<0016, 0002, 05>         | EE-99 = EE99h [61081d] checksum: checksum is incorrect, expected
                                                EEFCh [61180d]
<0018, 0002, 05>         | 00-01 = 01h [001d] identifier
<001A, 0002, 05>         | 00-02 = 02h [002d] sequence number
<001C, 0004, 05>         | 0.0.0.0 mask
<0015, 000B, 05>         | ---- end of packet: address mask request ----
<0014, 000C, 04>       | ---- end of packet: ICMP ----
<0014, 000C, 03>     | ---- end of packet: IP datagram ----
<0000, 0020, 02>   | ---- end of packet: IP PDU ----
<0020, 0012, 01> | packet pad:
<0020, 0010, 01> | 0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 28 79 | ........ ......(y
<0030, 0002, 01> | 0010: E3 C2 | ..
<0000, 0032, 01> | ---- end of packet: IP ----
}
```

Note that the decoded packet shows an ICMP packet with a checksum error in the payload, but the stats command reported no ICMP errors.

The output from tcpdump also shows an error:

```
01:08:11.848609 192.168.9.2 < 192.168.9.17: icmp: address mask request
    (wrong icmp csum) (ttl 32, id 2, len 32)
         4500 0020 0002 0000 2001 0778 c0a8 0902
         c0a8 0911 1100 ee99 0001 0002 0000 0000
```

The code that generated this packet is shown below. It will generate a packet with a checksum error in the payload and a correct checksum in the IP header. The packet-generating code was described in the previous Tclsh Spot article. Initializing the checksum to a non-zero value will generate an invalid checksum.

```
# Generate a bad checksum icmp Address Mask Request
set icmp2 [packet::make ICMP type ICMP_ADDRESS code 0 checksum 99 identifier 1 \
    sequence 2 subnet 00]

set ip2 [packet::make IP version 4 hdrlen 5 tos 0 length 32 id 2 flag 0 \
    offset 0 ttl 32 protocol ICMP checksum 0 source \
    192.168.9.2 dest 192.168.9.17 options {} payload [$icmp2 getField packet]]

set ep2 [packet::make ETHER dest 00:E0:4C:00:14:4D src 00:A0:CC:D1:B6:00 \
    type IP payload [$ip2 getField packet]]
```

The AX-4000 packet scans don't examine payloads. The *analyzerName* stats command returns no IP errors because there are no packets with bad IP checksums. The axdecode and tcpdump -s 15000 -l -x -n -v -i eth1 commands examine the payload and report the internal errors. (A simpler tcpdump -li eth1 command misses the error condition.)

While validating and debugging the packet generator, it's useful to get not only the error report (which tcpdump provides) but also the expected value (as reported by axdecode).

The ability to get a human-readable output becomes more important when we get to confirming more complex packets like the ICMP Destination Unreachable message, which includes the IP header and part of the payload of the packet that failed to reach a destination.

This code will generate a TIMESTAMP request, package it into an IP packet, and then use that IP packet as the body for an ICMP ICMP_DEST_UNREACH message.

```
# Generate a good icmp TIMESTAMP request
set icmp1 [packet::make ICMP type ICMP_TIMESTAMP code 0 \
    checksum 0 identifier 1 sequence 2 \
    originate 0x98765430 receive 0x98765431 transmit 0x98765432]

set ip1 [packet::make IP version 4 hdrlen 5 tos 0 length 32 id 2 flag 0 \
    offset 0 ttl 32 protocol ICMP checksum 0 source \
    192.168.9.2 dest 192.168.9.17 options {} \
    payload [$icmp1 getField packet]]

set pld1 [$ip1 getField packet]

# Make a good DESTINATION UNREACHABLE icmp message, using the
#    previous IP Packet for the body.

set icmp2 [packet::make ICMP type ICMP_DEST_UNREACH \
    code ICMP_DEST_UNREACH.ICMP_NET_UNREACH \
    checksum 0 zero 0 ipHeader [lrange $pld1 0 19] \
    payloadHeader [lrange $pld1 20 27] ]

set ip2 [packet::make IP version 4 hdrlen 5 tos 0 \
    length [expr 20 + [llength $pld]] id 2 flag 0 \
    offset 0 ttl 32 protocol ICMP checksum 0 source \
    192.168.9.2 dest 192.168.9.17 options {} \
    payload [$icmp2 getField packet]]
```

This generates a lengthy description of the packet, which is much more readable than the tcpdump hex display of the packet contents.

```
<0000, 003C, 01> | ---- packet: IP ----
<0000, 0038, 02< | ---- packet: IP PDU ----
<0000, 0014, 03>     | ---- packet: IP header ----
<0000, 0001, 03>     | {0100 ....}= 04h [004d] version: IP Internet Protocol
<0000, 0001, 03>     | {.... 0101}= 05h [005d] header length: in 32-bit units, must be 5 or more
<0001, 0001, 04>       | ---- packet: flags ----
<0001, 0001, 04>       } {000. ....}= 00h [000d] precedence field: Routine
<0001, 0001, 04>       | {...0 ....}= 00h [000d] minimize delay: normal delay
<0001, 0001, 04>       | {.... 0...}= 00h [000d] maximize throughput: normal throughput
<0001, 0001, 04>       | {.... .0..}= 00h [000d] maximize reliability: normal reliability
<0001, 0001, 04>       | {.... ..0.}= 00h [000d] minimize monetary cost: normal monetary cost
<0001, 0001, 04>       | {.... ...0}= 00h [000d] unused: must be 0
<0001, 0001, 04>       | ---- end of packet: flags ----
<0002, 0002, 03<     | 00-38 = 38h [056d] total length: in octets include header length, must not
                                                     be less than 20
<0004, 0002, 03<     | 00-02 = 02h [002d] identification
<0006, 0001, 03<     | {0... ....}= 00h [000d] reserved: valid
<0006, 0001, 03<     | {.0.. ....}= 00h [000d] Don't Fragment: May Fragment
<0006, 0001, 03<     | {..0. ....}= 00h [000d] More Fragments: Last Fragment
<0006, 0002, 03<     | fragment offset:
<0006, 0001, 03<     | {...0 0000}= 00h [000d] bits 12-8
<0007, 0001, 03<     | {0000 0000}= 00h [000d] bits 7-0
<0006, 0002, 03<     | fragment offset = 00h [000d]: in 64-bit units
<0008, 0001, 03<     | {0010 0000}= 20h [032d] time to live: seconds
<0009, 0001, 03<     | {0000 0001}= 01h [001d] protocol: ICMP (Internet Control Message)
<000A, 0002, 03<     | 07-60 = 760h [1888d] header checksum: checksum is correct
<000C, 0004, 03>     | 192.168.9.2 source IP address
<0010, 0004, 03>     | 192.168.9.17 destination IP address
<0000, 0014, 03>     | ---- end of packet: IP header ----
<0014, 0024, 03>     | ---- packet: IP datagram ----
<0014, 0024, 04>       | ---- packet: ICMP ----
<0014, 0001, 04>       | {0000 0011}= 03h [003d] type: destination unreachable
<0015, 0023, 05>         | ---- packet: destination unreachable ----
<0015, 0001, 05>         | {0000 0000}= 00h [000d] code: network unreachable
<0016, 0002, 05>         | C2-F7 = C2F7h [49911d] checksum: checksum is correct
<0018, 0004, 05>         | 00-00-00-00 = 00h [000d] unused
<001C, 001C, 06>           | ---- packet: IP packet caused error ----
<001C, 001C, 07>             | ---- packet: IPv4 ----
<001C, 001C, 08>               | ---- packet: IP PDU ----
<001C, 0014, 09>                 | ---- packet: IP header ----
<001C, 0001, 09>                 | {0100 ....}= 04h [004d] version: IP Internet Protocol
<001C, 0001, 09>                 | {.... 0101}= 05h [005d] header length: in 32-bit units, must
                                                           be 5 or more
<001D, 0001, 0A>                   | ---- packet: flags ----
<001D, 0001, 0A>                   | {000. ....}= 00h [000d] precedence field: Routine
<001D, 0001, 0A>                   | {...0 ....}= 00h [000d] minimize delay: normal delay
<001D, 0001, 0A>                   | {.... 0...}= 00h [000d] maximize throughput: normal
                                                           throughput
<001D, 0001, 0A>                   | {.... .0..}= 00h [000d] |maximize reliability: normal
                                                            reliability
<001D, 0001, 0A>                   | {.... ..0.}= 00h [000d] minimize monetary cost: normal
                                                           monetary cost
```

```
<001D, 0001, 0A>                      | {.... ...0}= 00h [000d] unused: must be 0
<001D, 0001, 0A>                      | ---- end of packet: flags ----
<001E, 0002, 09>                    | 00-20 = 20h[032d] total length: in octets include header
                                                       length, must not be less than 20
<0020, 0002, 09>                    | 00-02 = 02h [002d] identification
<0022, 0001, 09>                    | {0... ....}= 00h [000d] reserved: valid
<0022, 0001, 09>                    | {.0.. ....}= 00h [000d] Don't Fragment: May Fragment
<0022, 0001, 09>                    | {..0. ....}= 00h [000d] More Fragments: Last Fragment
<0022, 0002, 09>                    | fragment offset:
<0022, 0001, 09>                    | {...0 0000}= 00h [000d] bits 12-8
<0023, 0001, 09>                    | {0000 0000}= 00h [000d] bits 7-0
<0022, 0002, 09>                    | fragment offset = 00h [000d]: in 64 bits units
<0024, 0001, 09>                    | {0010 0000}= 20h [032d] time to live: seconds
<0025, 0001, 09>                    | {0000 0001}= 01h [001d] protocol: ICMP (Internet Control
                                                            Message)
<0026, 0002, 09>                    | 07-78 = 778h [1912d] header checksum: checksum is correct
<0028, 0004, 09>                    | 192.168.9.2 source IP address
<002C, 0004, 09>                    | 192.168.9.17 destination IP address
<001C, 0014, 09>                    | ---- end of packet: IP header ----
<0030, 0008, 09>                    | ---- packet: IP datagram ----
<0030, 0008, 0A>                      | ---- packet: ICMP ----
<0030, 0001, 0A>                      | {0000 1101}= 0Dh [013d] type: timestamp request
<0031, 0007, 0B>                        | ---- packet: timestamp request ----
<0031, 0001, 0B>                        | {0000 0000}= 00h [000d] code:
<0032, 0002, 0B>                        | 2D-05 = 2D05h [11525d] checksum: checksum is incorrect,
                                                                expected 5284h [21124d]
<0034, 0004, 0B>                        | 00-01-00-02 = 10002h [65538d] Originate Timestamp
<0031, 0007, 0B>                        | ---- end of packet: timestamp request ----
<0030, 0008, 0A>                      | ---- end of packet: ICMP ----
<0030, 0008, 09>                    | ---- end of packet: IP datagram ----
<001C, 001C, 08>                  | ---- end of packet: IP PDU ----
<001C, 001C, 07>                | ---- end of packet: IPv4 ----
<001C, 001C, 06>              | ---- end of packet: IP packet caused error ----
<0015, 0023, 05>            | ---- end of packet: destination unreachable ----
<0014, 0024, 04>          | ---- end of packet: ICMP ----
<0014, 0024, 03>        | ---- end of packet: IP datagram ----
<0000, 0038, 02>      | ---- end of packet: IP PDU ----
<0038, 0004, 01> | 0007: 00 BE 9F BA | .... = BE9FBAh [12492730d] packet pad
<0000, 003C, 01> | ---- end of packet: IP ----
```

Note that the innermost ICMP checksum is reported as being incorrect. This is because the packet analysis program is recalculating the checksum on the incomplete ICMP message.

In terms of using an AX-4000, this is a trivial application. The equipment is capable of monitoring and collecting data on a much more complex set of rules. However, it's a useful tool for testing and validating this application, which is the point of this exercise.

As usual, the code described in this article is available at *http://www.noucorp.com*.