BY PETER BEST

LAYING A FOUNDATION FOR PING

IN THE PREVIOUS EDITION OF DESIGN CYCLE, I described the steps that were necessary to "fool" your personal computer (PC) into having a conversation with the Ethernet MINI and the minimal driver code we had completed at that time. This month, we will take yet another firmware step forward and complete the coding needed to allow you to PING your Ethernet MINI and communicate with the Visual Basic-based EDTP Internet Test Panel application.

To accomplish the tasks of replying to a PING and sending UDP datagrams to the EDTP Internet Test Panel, we will need to code up an ARP reply routine and a simple UDP echo module. The new ARP routine will replace the manual stuffing of the Ethernet MINI's IP and MAC addresses into the PC's ARP cache I outlined in the last Design Cycle column. The assembly of the UDP echo source code module will do more than just allow us to send UDP datagrams to the EDTP Internet Test Panel Visual Basic application. UDP is the basis of DHCP message communication and having the basic UDP coding techniques under our belts will make our coding lives easier when we are ready to code up the DHCP modules. ARP is currently the most important piece of coding we have to do. So, let's take a swing at getting our Ethernet MINI hardware to respond to an ARP request.

CODING THE ARP FIRMWARE

ARP is network shorthand for Address Resolution Protocol. In our situation, the Ethernet MINI is normally the recipient of an ARP request. An ARP request is generated by any host on a network that is searching for another particular host on the network. In our case, the requesting or searching host is a PC attached to the Ethernet MINI's network. The problem for the searching host is that it only has the desired destination host's IP address. The host that is on the search mission also needs to have the desired destination host's MAC (hardware) address as well, in order to communicate directly to it.

The reasoning behind this is that many differing hosts on different (but connected) networks may all have the same IP address. However, the MAC address of every host on any network must be unique to that particular host. The uniqueness of the MAC addresses is guaranteed if the network host's hardware address was issued by the IEEE. By issuing an ARP request embedded with the desired destination host's IP address, the searching host can obtain the desired destination host's unique MAC address it needs to make the network connection with the desired remote network host that has the IP address specified in the ARP request. Now that you know why we need to code an ARP module, let's do just that.

The ARP components take up the same packet space as the IP header (and then some). Thus, an ARP packet is its own thing, as an ARP message is not encapsulated within an IP packet. Despite this seeming lack of respect for IP, ARP must still use elements of IP to do its job. Consider the following code snippet:

```
;*********************************************************
;*     ARP Layout
;*********************************************************
arp_hwtype          CON $0E
arp_prtype          CON $10
arp_hwlen           CON $12
arp_prlen           CON $13
arp_op              CON $14    ;arp request or response
arp_shaddr          CON $16    ;arp source mac address
arp_sipaddr         CON $1C    ;arp source ip address
arp_thaddr          CON $20    ;arp target mac address
arp_tipaddr         CON $26    ;arp target ip address
```

Take a look at the ported PICBASIC PRO source that I've provided. Notice that the ARP_hwtype and ip_vers_len fields are located at the same location within an Ethernet packet. Note also that the ARP framework is four bytes larger than the area occupied by the IP header. We will use these differences and similarities in our Ethernet MINI firmware to fish out an incoming ARP message and act upon it. The stream we will fish in for ARPs runs through the get_frame subroutine. Here's how we bait the hook:

```
;process an incoming ARP request packet
if packet[enetpacketType0] == $08 &&
packet[enetpacketType1] == $06  then
  if packet[arp_hwtype+1] == $01          &&_
   packet[arp_prtype] == $08              &&_
   packet[arp_prtype+1] == $00            &&_
   packet[arp_hwlen] == $06               &&_
   packet[arp_prlen] == $04               &&_
   packet[arp_tipaddr]    == ipaddrc[0]   &&_
   packet[arp_tipaddr+1] == ipaddrc[1]    &&_
   packet[arp_tipaddr+2] == ipaddrc[2]    &&_
   packet[arp_tipaddr+3] == ipaddrc[3] then

   select case packet[arp_op+1]
    case $01
     gosub arp_reply
    case $02
     for i8 = 0 to 5
      remotemacaddrc[i8] = packet[arp_shaddr+i8]
     next i8
    gosub set_arpflag
   end select
   endif
 endif
```

The ARP Ethernet packet type of $0806 tugs on our fishing line as the bait is being taken. An incoming IP packet would have $0800 in the Ethernet packet type field. The hwtype or hardware address type field's $01 denotes a 10 MB Ethernet. Recall that ARP does indeed use IP components. So, the protocol type field is filled with $0800, which denotes the IP protocol.

A MAC address is made up of six bytes. This is conveyed in the arp_hwlen field with a value of $06. The arp_prlen array entry tells us that the IP address consists of four octets. Octet is Internet document speak for byte. It is possible to only filter on the Ethernet packet type and the IP address to determine that we have hooked an incoming ARP message. However, to be safe, I decided to verify every field I thought that I should to make sure the incoming message was indeed an ARP and that the captured ARP message was directed to the hardware running this ported PICBASIC PRO firmware.

The arp_op array entry is the operation that is being requested. A $01 in the operation field represents an incoming ARP request that we must reply to. If our Ethernet MINI generated an ARP request, we would look for the operation field to contain $02 and the returned ARP message to contain the MAC address of the replying host.

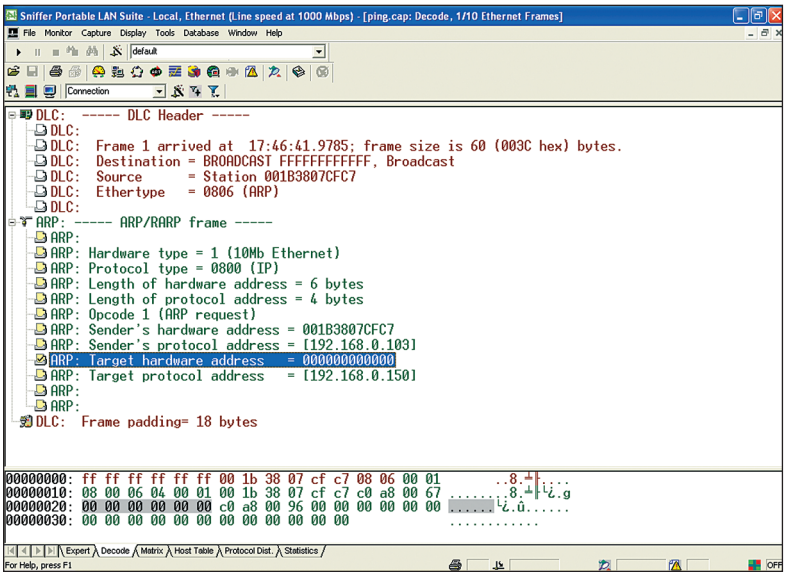
You can immediately see what happens when an incoming ARP message contains the in-question MAC address. We simply stuff the contents of the arp_shaddr (**s**ource**h**ardware**add**ress) array fields into the remotemacaddrc array fields for later use. If you're wondering about the "c" in remotemacaddrc, that's a holdover from the port from C. I use "c" to denote byte variables and "i" to identify integer variables when there are a bunch of them all mixed up under my care.

We really need to respond to an incoming ARP request before we can do anything useful with the Ethernet MINI. So, branch to the arp_reply subroutine in the Ethernet MINI driver PICBASIC PRO source code. The arp_reply subroutine is simply building an ARP reply packet in the PIC18F67J60's internal transmit buffer memory. Recall that we have already hacked out transmit and receive buffer areas within the PIC18F67J60's 8K of packet buffer SRAM.

Once the transmit buffer write pointers (EWRPTH/EWRPTL) are initialized and the mandatory control byte is placed into the PIC18F67J60's transmit buffer, we use the source MAC address we gleaned from the incoming ARP message to use as a return hardware address. The addition of our MAC address as the sender (source) in the initial address fields (DLC Header) of the packet will not fulfill our reply obligation. The MAC address the requesting host is looking for is contained within the body of the ARP message. The best way to illustrate the inner workings of the PICBASIC PRO ARP code is with an illustration.

Figure 1 is a screen capture taken from a Network General Sniffer Portable Ethernet sniff session. The sniff session participants are a laptop and an Ethernet MINI. Note the PC's MAC address (Source = Station 001B3807CFC7) is contained within the DLC Header area and the body of the ARP packet. The reason for this is that networks aren't always simple, router-less two-node dances. The Internet is built around routers and routers need addresses to be able to forward packets around the Internet. Thus, the DLC Header addresses are used by the routers (and host nodes for that matter) for addressing information as a router isn't programmed to dig deep into packets to retrieve address information.

We've already talked about most everything contained



■ FIGURE 1. The Network General Sniffer products are top notch. I rely on the Network General Sniffer when developing embedded Ethernet firmware as I can easily verify my checksum code and see all of the fields of an Ethernet packet I am interested in. This is also a great way to show *Nuts & Volts* readers the innards of Ethernet packets.

in the ARP/RARP frame fields of Figure 1 with the exception of the MAC and IP fields. Figure 1 is a sniff representation of an ARP request that was sent by the laptop on the two-node Ethernet MINI network. Since the PC knows the Ethernet MINI's IP address but does not know the Ethernet MINI's MAC address, the ARP request is transmitted as a broadcast message with the target MAC address left "blank." The Ethernet MINI's IP address is 192.168.0.150 and the Target protocol address field in Figure 1 was populated by the PC before sending the ARP request. The laptop is looking for the Ethernet MINI to respond to this ARP request and fill in the "blank" MAC address, which is, of course, the Ethernet MINI's MAC address.

The Ethernet MINI fulfilled its ARP obligation in Figure 2. Although the Ethernet MINI has inserted its MAC address into the DLC Header reply, the requesting application will actually retrieve the Ethernet MINI's MAC address from the sender's hardware address fields within the ARP frame. Thus, our ported Ethernet MINI ARP PICBASIC PRO source code built an ARP reply frame and turned around the source and destination addresses to realize and send along the ARP reply packet shown in Figure 2. When you study the PICBASIC PRO arp_reply subroutine, you will find that there is nothing in the coding that is complex at all. In fact, I didn't have to do much of anything to port many of the original C source lines to the PICBASIC PRO language.

At this point, the laptop has retrieved the Ethernet MINI's MAC address information and placed it into its ARP cache. You can use the arp /a command on the laptop to see the Ethernet MINI's IP and MAC cache entries. In that, the Network General Sniffer caught the ARP request and reply sequence, and I could see the Ethernet MINI's address information in my laptop's ARP cache. We can therefore be fairly sure that the Ethernet MINI's ARP PICBASIC PRO code worked as designed.

PINGING WITH PICBASIC PRO

The first major step in our PICBASIC PRO code port has been taken. We are now able to identify our PICBASIC PRO-laden Ethernet MINI module to other hosts on an Ethernet network via an ARP reply.

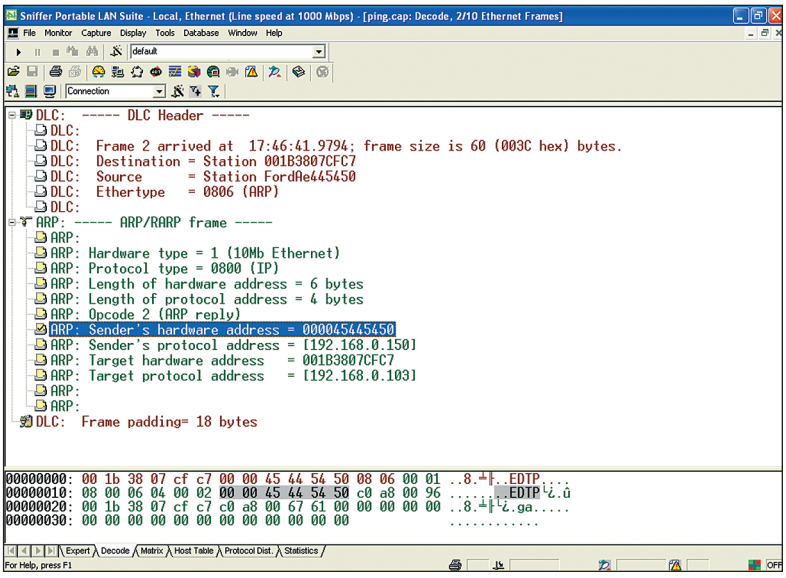
A PING is really an ICMP operation. So, we must port the original C ICMP code to PICBASIC PRO if we want to PING our Ethernet MINI that is running the PICBASIC PRO driver code. Our PING coding takes place in the icmp subroutine. However, the icmp subroutine does not get called unless the correct conditions are met in the code snippet that follows:

```
;process an IP packet
if packet[enetpacketType0] == $08        &&_
  packet[enetpacketType1] == $00         &&_
packet[ip_destaddr]    == ipaddrc[0]     &&_
packet[ip_destaddr+1] == ipaddrc[1]      &&_
packet[ip_destaddr+2] == ipaddrc[2]      &&_
packet[ip_destaddr+3] == ipaddrc[3]   then

select case packet[ip_proto]
 case PROT_ICMP
  gosub icmp
 case PROT_UDP
  if packet[UDP_srcport] == DHCP_SERVER_PORT then
   gosub dhcp_state_engine
  else
   gosub udp
  endif
 case PROT_TCP
  gosub tcp
end select
endif
```

Unlike the ARP frame, this is a true IP datagram as denoted by the $0800 Ethernet type value. Since this is an IP packet, there are a number of things that can be wrapped up inside of it. According to the code I just offered that processes an IP packet, we could have an ICMP, a UDP, or a TCP operation encapsulated within the incoming IP datagram. Another look at the IP packet parsing code tells us that all we need to do is make sure it is indeed an incoming IP datagram and that the incoming IP datagram is actually addressed to our Ethernet MINI. Then — depending on the encapsulated protocol type — we branch off into analyzing and acting upon the payload of the incoming IP datagram we have captured with the Ethernet MINI. Since we're on the subject of PINGing, we'll turn our attention to the PROT_ICMP path, which invokes the ported PICBASIC PRO icmp subroutine.

A PING response is basically nothing more than an echo of the received data payload. However, we must readdress the PING reply packet and calculate some checksums before sending our PING reply. A



■ FIGURE 2. The receiving application retrieves the missing Ethernet MINI MAC address from the highlighted fields in this sniff. Note the human-readable fields are supported by a hex dump of the data at the bottom of this shot. This is a typical Network General Sniffer presentation.

converted PICBASIC PRO subroutine called setipaddrs is responsible for the address reconfiguration.

```
;*******************************************************
;*       SETIPADDRS
;*   This subroutine builds the IP header.
;*******************************************************
    ;move IP source address to destination address
    packet[ip_destaddr]=packet[ip_srcaddr]
    packet[ip_destaddr+1]=packet[ip_srcaddr+1]
    packet[ip_destaddr+2]=packet[ip_srcaddr+2]
    packet[ip_destaddr+3]=packet[ip_srcaddr+3]
    ;make ethernet module IP address source address
    packet[ip_srcaddr]=ipaddrc[0]
    packet[ip_srcaddr+1]=ipaddrc[1]
    packet[ip_srcaddr+2]=ipaddrc[2]
    packet[ip_srcaddr+3]=ipaddrc[3]
    ;move hardware source address to destinatin address
    packet[enetpacketDest0]=packet[enetpacketSrc0]
    packet[enetpacketDest1]=packet[enetpacketSrc1]
    packet[enetpacketDest2]=packet[enetpacketSrc2]
    packet[enetpacketDest3]=packet[enetpacketSrc3]
    packet[enetpacketDest4]=packet[enetpacketSrc4]
    packet[enetpacketDest5]=packet[enetpacketSrc5]
    ;make ethernet module mac address the source address
    packet[enetpacketSrc0]=macaddrc[0]
    packet[enetpacketSrc1]=macaddrc[1]
    packet[enetpacketSrc2]=macaddrc[2]
    packet[enetpacketSrc3]=macaddrc[3]
    packet[enetpacketSrc4]=macaddrc[4]
    packet[enetpacketSrc5]=macaddrc[5]
```

Porting the initial portion of the setipaddrs subroutine was a piece of cake as all I had to do was remove the semicolons from the ends of the ported statements. In actuality, I could have left the semicolons in place as they are comment delimiters in PICBASIC PRO. The downside to this laziness is that sometimes an active statement following the semicolon that needs to be ported gets the commented color scheme, and can be accidentally erased or ignored. During porting, I had the "ignore" scenario occur just before I was about to execute the "erase" scenario. So, I've eliminated the end-of-C-statement semicolons in our C-to-PICBASIC PRO port. As you can see in the setipaddrs code snippet, the setipaddrs subroutine simply swaps the source and destination IP and MAC addresses in preparation for the transmission of the PING reply packet.

Just when you think things are in hand, the bottom of your basket falls through. In the C source we're porting, the 16-bit IP header and ICMP header checksums are calculated using 32-bit variables. Unfortunately, PICBASIC PRO knows what a 32-bit variable is, but it has no orders to do anything about them. The PICBASIC PRO DIV32 mnemonic and some tricky Darrel Taylor PICBASIC PRO/assembler code are all that we have to work with when it comes to using ported unsigned long C variables with PICBASIC PRO. Unfortunately, I was unable to think (or should that be trick) my way through the 16-bit checksum problem using Darrel's readily-available algorithms. That was a loss as Darrel's DIV32 code is compact and efficient. So, I had to resort to Plan B from outer space. Here's the first of the alien invasion code:

```
chksum16        var word
acc3            var word
acc2            var word
acc1            var word
acc0            var word
arg3            var byte
arg2            var byte
arg1            var byte
arg0            var byte
```

Let's translate. Note that there are four acc word variables and four arg byte variables. The acc variables are accumulator variables while the arg variables I've declared are arguments or mathematical operands. When all is said and done, the chksum16 variable will contain our magic checksum, which was calculated with the help of the accumulators and arguments.

The IP checksum is defined as the 16-bit one's complement sum of all 16-bit words in the header. PICBASIC PRO can natively manipulate 16-bit variables. Our problem lies in the fact that when accumulating 16-bit values, an overflow into the 17th bit can possibly occur. PICBASIC PRO will throw the 17th bit in the bit bucket.

Our checksum calculation uses all of the bits from bit 17 up to bit 31. Our job is to arrange all of the IP header bytes into words, add them all together with the carries out into bit 17 and beyond, and invert the sum by performing a one's complement against it. Simple, huh? Yep ... and here's how we'll do it.

In your mind, group the arg variables into a 32-bit variable with arg3 representing the most significant byte of the 32-bit long variable and arg0 acting as the least significant byte of the 32 bit variable. For instance, if arg3=$03, arg2=$02, arg1=$01, and arg0=$00, that would equate logically to a 32-bit value of $3210. Get the idea? The same logic applies to the accumulators with a twist.

Each accumulator is a word variable and is 16 bits in length. PICBASIC PRO doesn't natively allow the programmer to interrogate carry situations. So, if we declared each accumulator variable as a byte, the addition of $FF and $01 in an accumulator variable would render $100, which is out of the bounds of an eight-bit variable. The carry out of the addition of $FF + $01 would be lost to us as the accumulator variable would simply roll over to $00.

The accumulators are all declared as words to trap any byte addition overflows. Consider this. We load acc0 with $FF. We then load arg0 with $01. The values are then added together and stored in acc0 like this: acc0 = acc0 + arg0. Since acc0 is a word variable, the value stored in acc0 following the addition will be $0100. Here's where our logic becomes magic.

We have arranged the accumulators as four logical words with acc0 being the least significant word and acc3 as the most significant word. That's logically 64 bits. The magic comes in as we treat each accumulator as a byte not as a word. The most significant byte of each accumulator

variable is simply a bit bucket for the carry bits. The value in the most significant byte of acc0 is added to the least significant byte of acc1 and the most significant byte of acc0 is cleared to $00. Do you get it?

The most significant byte of each of the accumulator variables is added to the least significant byte of the next accumulator in the 64-bit accumulator chain we logically created. After each carry-over addition, the most significant byte of the accumulator that overflows is cleared to zero. Thus, we logically have only 32 bits in our accumulator chain (the accumulator's least significant bytes), with each accumulator having its own carry bit bucket (the most significant byte of each accumulator) from which we can dip. The bit buckets are serviced by the ripple_crc subroutine, which I present to you here:

```
ripple_crc:
                if acc0 > $FF then
                 acc1 = acc1 + 1
                 acc0 = acc0 & $00FF
                endif
                if acc1 > $FF then
                 acc2 = acc2 + 1
                 acc1 = acc1 & $00FF
                endif
                if acc2 > $FF then
                 acc3 = acc3 + 1
                 acc2 = acc2 & $00FF
                endif
                if acc3 > $FF then
                 hserout[13,10,
                        "CHECKSUM OVERFLOW ERROR",13,10]
                endif
                return
```

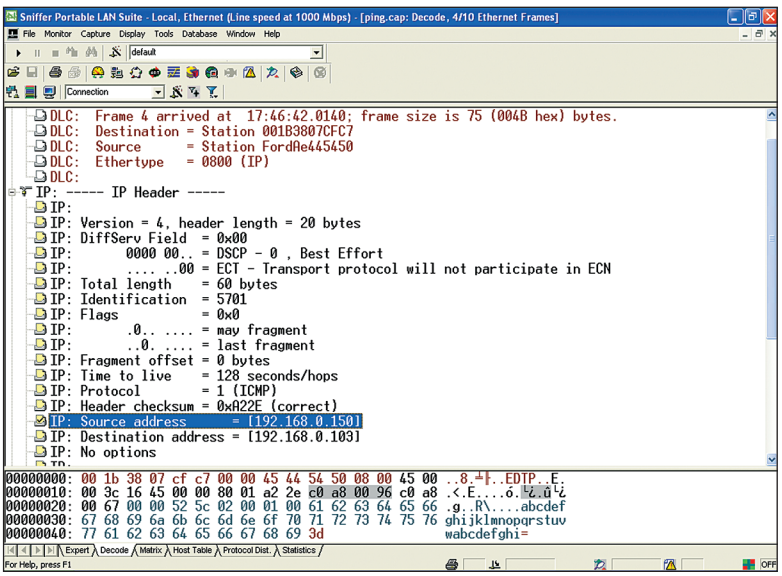
Since the IP checksum specifies that we add 16-bit numbers, we logically group the arg byte variables into pairs, which we logically add to the accumulators as 16-bit values. You can see this happening in the remainder of the setipaddrs subroutine PICBASIC PRO code. Here's the ported PICBASIC PRO code that calculates the IP header checksum:

```
  ;calculate the IP header checksum
  packet[ip_hdr_cksum]=$00
  packet[ip_hdr_cksum+1]=$00

  hdrlen = (packet[ip_vers_len] & $0F) * 4
  ;addr = &packet[ip_vers_len]
      i16 = 0
      gosub clrcrc
      while(hdrlen > 1)
            arg1=packet[ip_vers_len+i16]
            i16 = i16 + 1
            arg0=packet[ip_vers_len+i16]
            i16 = i16 + 1
            acc0 = acc0 + arg0
            gosub ripple_crc
            acc1 = acc1 + arg1
            gosub ripple_crc
            hdrlen = hdrlen - 2
      wend
   if(hdrlen > 0) then
     arg1=packet[ip_vers_len+i16]
     arg0=$00
            cc0 = acc0 + arg0
            gosub ripple_crc
            acc1 = acc1 + arg1
            gosub ripple_crc
      endif
            acc0 = acc0 + acc2
            gosub ripple_crc
            acc1 = acc1 + acc3
            gosub ripple_crc
  chksum16= ~((acc1 << 8) + acc0);
  packet[ip_hdr_cksum] = (chksum16 & $FF00) >> 8
  packet[ip_hdr_cksum+1] = chksum16 & $00FF
return
```

The theory behind our accumulator-based checksum calculator seemed to work well. However, the real proof would come when a PING request jumped from my laptop to the Ethernet MINI.

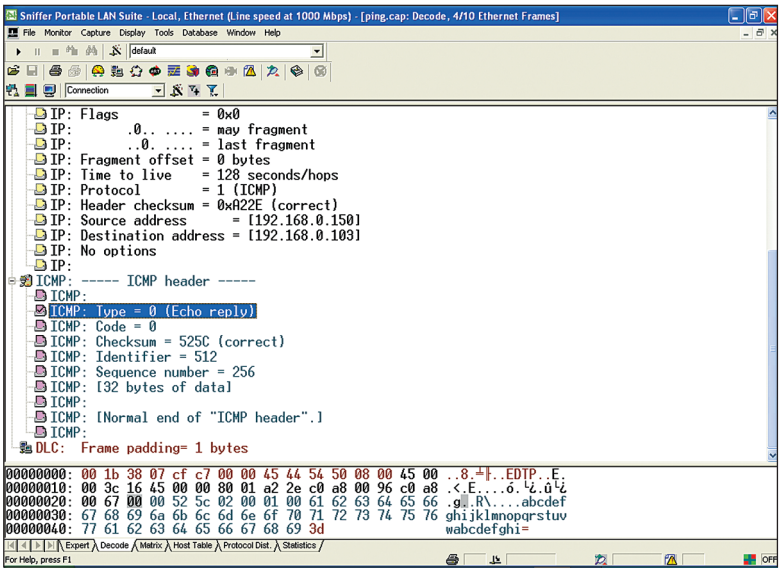
I am vindicated in Figure 3. Our ported PICBASIC PRO checksum code in the setipaddrs subroutine calculated the IP header checksum value in the PING response packet you see sniffed in Figure 3. Our PICBASIC PRO checksum code was again put to the test in Figure 4. We correctly calculated the ICMP checksum, which is defined as the 16-bit one's complement of the one's complement sum of the ICMP message starting with the ICMP Type.



■ FIGURE 3. You can tell that this PING reply emanated from the Ethernet MINI by examining the source and target IP addresses. The Ethernet MINI is addressed as 150 on the network while my laptop has an IP address ending in 103.

A PORTED UDP APPLICATION

The original C source we're using as a base for our PICBASIC PRO port contains a simple UDP application that echoes characters sent from a Visual Basic program running on a PC. I've included the Ethernet MINI UDP application code in this month's Design Cycle download package. I don't think you'll have any problems following the



■ FIGURE 4. Our 16-bit ported PICBASIC PRO checksum routines have passed the test as we have now calculated the IP header and ICMP checksums. Note that the Network General Sniffer points out that this is an Echo Reply and references the byte field in the hex dump at the bottom of the shot. All hail the Network General Ethernet Sniffer!

PICBASIC PRO UDP code flow as it looks just like everything we're mulled over up to now. Now that we have ported the code that drives the PIC18F67J60 Ethernet hardware, most of what we are doing consists of parsing fields of incoming packets, calculating checksums, and correctly placing IP and MAC addresses.

When you go over the UDP code, note that UDP adds yet another concept we must consider in our code. In addition to an IP and MAC address, UDP utilizes source and destination port addresses. The UDP application I've supplied in the download package (**www.nutsvolts.com**) looks for a message sent to well-known port 7, which is the echo port. "Well-known" means that this is a standardized port. You can use port 7 for other things but you may run into a problem if the other guy or gal you want to communicate with has used port 7 for its original intended purpose.

I've also included the time-tested EDTP Internet Test Panel application, which runs on a PC. The EDTP Internet Test Panel application you see in Figure 5 is easy to use. All you have to do is enter the Ethernet MINI's IP address in the Target IP Address window and type characters into the Original Data window. Ignore the Target Port and LCD Data Entry windows as they were part of a past EDTP application. If things work as designed, what you type into the Original Data window will be echoed by the Ethernet MINI back to the Echoed Data window in the EDTP Internet Test Panel application frame.

STACKING UP THE PROTOCOLS

We've worked our way through the coding of the PIC18F67J60 physical layers. With the completion of this month's discussion, we've worked our way through the PICBASIC PRO 32-bit limitation and established a foothold on the coding of the basic Internet protocols. With the successful porting of the ARP, ICMP, and UDP protocols behind us, we are climbing towards the most famous protocol: TCP. Next time, we'll talk more about UDP and port the Ethernet MINI DHCP and TCP C source to PICBASIC PRO. I'm having a blast coding with PICBASIC PRO. **NV**

SOURCES

■ *EDTP Electronics, Inc.* (**www.edtp.com**): Ethernet MINI

■ *microEngineering Labs, Inc.* (**www.melabs.com**): PICBASIC PRO

■ FIGURE 5. This application has been around for a couple of years and has seen widespread use. It's simply a tool that was created to test the functionality of the UDP firmware written at EDTP Electronics. All you have to do is dial in a Target IP Address and start typing.