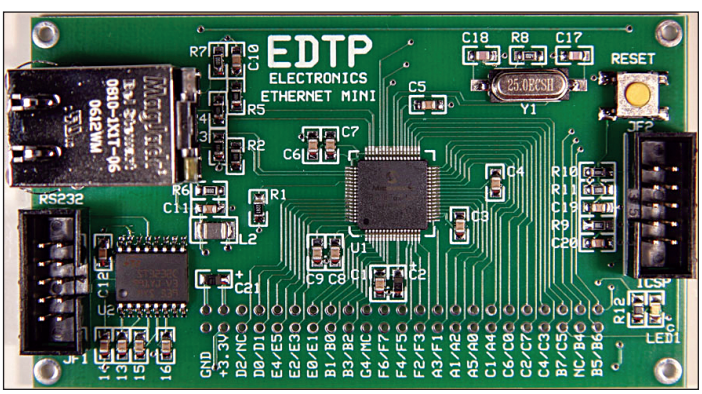
■ BY PETER BEST

REWRITING C IN PICBASIC PRO

THE C PROGRAMMING LANGUAGE HAS GROWN LIKE a creeping weed extending its branches and leaves out from the personal computer (PC) world and into the realm of the microcontroller. To put a C language program into a microcontroller, you will need a C compiler that is tooled for microcontrollers. Good C compilers are based on a set of standards that have been applied to other equal or better C compilers. Thus, C programs written with these standards-based C compilers tend to be portable between hardware platforms. For instance, it is fairly easy to port a Microchip C18 application to the ways of the HI-TECH PICC-18 C compiler. It's also light work to port any Custom Computer Services PIC C compiler application to either Microchip C18 or the HI-TECH PICC-18 C compiler. That's nice. However, the project we're about to tackle will be coded entirely in Basic. When I think of PIC microcontrollers and BASIC, an image of code splashes written with the microEngineering Labs PICBASIC PRO Basic Compiler displays for my mind's eye.

STAND AND BE COUNTED

Many of you *Nuts & Volts* readers are avid Basic programmers. I'm not privy to the *Nuts & Volts* subscriber list, but I'll bet that much of the Basic code aimed at microcontrollers found or referenced within the microEngineering Labs developer's resources forum was created by *Nuts & Volts* readers. Believe it or not, when it comes to portability, source code aimed at the PICBASIC PRO Basic Compiler can be easily redirected to and from lines of C source code. If you don't see the light now, you will by the time we're finished with this project as we will port the entire Ethernet MINI driver from HI-TECH C to PICBASIC PRO Basic.



■ PHOTO 1. We discussed this piece of hardware in the previous installment of Design Cycle. The Ethernet MINI is based on the Microchip PIC18F67J60, a stand-alone Ethernet node/microcontroller combination. You'll need to understand how this puppy works to get the most out of the porting project.

At first glance, the C-to-Basic port we're about to embark on does not look to be a walk in the park. There is a considerable amount of C source we must convert to equivalent Basic statements. As with all huge and seemingly insurmountable tasks, the key to success is to divide and conquer. We will break the porting tasks down into small chunks and logically work our way up the coding hill. However, before we begin this porting hike, it would be wise to have a plan and understand the pitfalls we will encounter along our way.

THE PLAN

We will be writing code for the EDTP Ethernet MINI (shown in Photo 1), which is based on the Microchip PIC18F67J60 microcontroller. A complete set of hardware drivers for the Ethernet MINI — which includes ARP, PING, DHCP, TCP, and UDP — has already been fielded by EDTP's Fred Eady. We will take that existing (and known working) technology and convert line by line Fred's firmware C statements to corresponding Basic statements.

In preparation for this project, I've put in some extensive searching and reading time. I'm very familiar with the Microchip PIC18F67J60. So, 99% of my research was done within the pages of the microEngineering Labs developer resources forum. I'm not totally PICBASIC PRO challenged as I have previously fielded some PICBASIC PRO-based projects. Much of my study was intended to re-familiarize myself with the PICBASIC PRO mnemonics and the ways of the Basic language.

There are many wise men and women contributing to the microEngineering Labs forum and if you're in need of PICBASIC PRO Basic Compiler help or advice, the answers lie in this forum. You can also find some very tricky PICBASIC PRO code there, as well. (See the Sources box for company websites.)

The tools we will use are standard fare. The Microchip MPLAB ICD 2 will be used to interface the Ethernet MINI to Microchip's MPLAB IDE. As I was reading through the microEngineering Labs PICBASIC PRO Basic Compiler forum, I came across a couple of posts that indicated that when "serious" PICBASIC PRO debugging was necessary, the MPLAB ICD 2 was employed. Well, we're going to get pretty serious and that's precisely why I've tapped the MPLAB ICD 2 for this project.

From the editorial point of view, using the MPLAB ICD 2 in conjunction with MPLAB IDE allows me to easily include PIC microcontroller memory dumps and watch values that I see during production into the text that you ultimately read. The EDTP shop is geared up for Microchip products, which means they have and use Microchip development tools in all of their PIC projects. I realize that you may not be equipped with Microchip factory development equipment. There is absolutely no reason why you can't deploy your personal PIC18F67J60 debugging/programming development suite. I once had a discussion with a programmer that I highly respect and he said, "It doesn't matter which language you use if your application works in the end." I feel the same way about development tools. If a suite of development products works for you, get them out and use them.

The PICBASIC PRO Basic Compiler melds nicely into the MPLAB IDE. With the MPLAB ICD 2, source level debugging and many of its options are available with this configuration. For instance, we are allowed to set break-points, examine microcontroller register and variable values, view the resultant assembler code, and look at memory areas. The triad of PICBASIC PRO, MPLAB IDE, and MPLAB ICD 2 development tools also gives us the ability to single-step through the Ethernet MINI driver source code.

Great strides have been made in microcontroller USB interfacing (kudos to Dr. Bob and his HIDMaker FS). One day, I will have to move on to microcontroller-based USB interfacing and leave the trusty RS-232 interface behind. However, as long as I can obtain and solder easy-to-use RS-232 interface ICs into my projects and run Tera Term Pro on my laptop, I'll most likely use an RS-232 interface.

In the case of the Ethernet MINI, we'll plan to activate an interrupt-driven RS-232 interface based upon the PIC18F67J60's hardware EUSART. The availability of an RS-232 interface on the Ethernet MINI provides an extra debugging tool if we choose to deploy it during our porting process. PICBASIC PRO also offers its own firmware-based RS-232-like debugging mnemonics. However, we won't plan on using the PICBASIC PRO serial debugging elements in this project. The base purpose of implementing an RS-232 port on the Ethernet MINI is to display status messages, which we will hope to see in short order.

That's the plan. Use high-visibility debugging tools and techniques to complete the C-to-Basic port. Now, let's consider the rocks that will be in our path.

THE PITFALLS

The Basic programming language has endured the test of time. I can recall sitting in my local RadioShack store (in which I was later employed) writing simple Basic programs on the newly announced TRS-80. If you consider what I was doing with the TRS-80 then versus what you and I are doing right now with this project, you'll find that I've not changed much over the years. Consider this. I was writing Basic code for a 4 MHz Z80 microcontroller in that RadioShack store. I am currently writing Basic code for a 41.6667 MHz PIC18F67J60 microcontroller in *Nuts & Volts Magazine*. Hmmmm ...

When I was banging out Ohm's Law calculations on the TRS-80, college students and engineers were banging out applications on UNIX machines using the publicity-shy C programming language. The commercial Internet as we know it today did not exist in my TRS-80 programming days and you can bet those UNIX guys and gals with their C compilers and big network of computers were hard at work laying down the foundation of the Internet we now know and love.

Obviously, Basic is a viable alternative to C as far as our Ethernet MINI driver project is concerned. Otherwise, we would not be indulging in this conversation, and we would not be wasting cycles by attempting to perform this C-to-Basic port. However, there are some things C that don't line up directly to things Basic. For instance, you can't write Basic macros that correspond to C macros because Basic macros using PICBASIC PRO don't exist. Let me qualify that last statement. Macros written with PICBASIC PRO mnemonics don't exist. Assembler macros can be threaded into PICBASIC PRO source code. I don't care whether it is C or Basic. It's the compiler's job to interpret our source statements and generate the appropriate assembler mnemonics. Thus, we won't code any assembler other than a simple NOP (No OPeration) instruction.

The only reason we will code NOPs in assembler form is that NOP is not a native PICBASIC PRO mnemonic. Since we are unable to code a C-like macro in PICBASIC PRO, we will convert all of the C macros we come across into Basic subroutines. I can see some of you cringing. I realize that a subroutine in any language is not as efficient as a macro or an assembler routine most of the time. However, that's the cards we were dealt and we'll play them as best we can.

The C programming language allows the programmer to enumerate variables automatically. Enumeration is simply the assignment of a number to each element in a set of elements with each successive element number being automatically assigned to the next corresponding element in the sequence. For instance, consider enumerating the constants A, B, and C beginning with the number 1. Constant A would be enumerated as 1, B as 2, and C as 3. Thus, the elements in this enumeration would correspond to their enumerated numeric values. Here is how enumeration was used in the Ethernet MINI C driver source code. Consider the code snippet:

```
typedef enum _DHCP_STATES
{
	DHCP_ENTRY,
	DHCP_INIT,
	DHCP_WAIT,
	DHCP_BROADCAST,
	DHCP_DISCOVER,
	DHCP_REQUEST,
	DHCP_BIND,
	DHCP_BOUND,
	DHCP_DISABLED
}DHCP_STATE_LIST;

DHCP_STATE_LIST DHCPSTATE;
```

In C and Basic, every variable has a type. The typedef keyword is used here to create a new data type name called DHCP_STATE_LIST, which is simply a name that represents the values of the enumerated _DHCP_STATES constants. Each _DHCP_STATE constant between the braces is enumerated (DHCP_ENTRY = 0, DHCP_INIT = 1, etc.). The idea of using a typedef is to make the program easier to read and understand. Thus, in this case each constant has a name associated with its enumerated value. DHCPSTATE is the actual variable that the program will use. DHCPSTATE is of type DHCP_STATE_LIST, which is defined as the enumeration _DHCP_STATES. Now that the enumerated constants have human-understandable names, we can simply check the value of the DHCPSTATE variable using simple C comparison statements like these:

```
switch(DHCPSTATE)
{
	case DHCP_ENTRY:
		printf("\r\nDHCP RESET..");
		for(i=0;i<4;++i)
			tempipaddrc[i] = 0x00;
		DHCPSTATE = DHCP_INIT;
	case DHCP_INIT:
		printf("\r\nDHCP INIT..");
		for(i=0;i<6;++i)
			svrmacaddrc[i] = 0xFF;
		for(i=0;i<4;++i)
			svridc[i] = 0xFF;
		msecs_timer = 0;
		DHCPSTATE = DHCP_WAIT;
	case DHCP_WAIT:
		if(msecs_timer >= 2000)
			DHCPSTATE = DHCP_BROADCAST;
		break;
```

Do you get the idea? We have assigned human names to the enumerated list and the constant values in the enumerated list. DHCPSTATE is an arbitrary variable name that just happens to fit here. We could have just as well designated the variable name as PETER, which would have resulted in the code that follows:

```
DHCP_STATE_LIST PETER;

switch(PETER)
{
	case DHCP_ENTRY:
		printf("\r\nDHCP RESET..");
		for(i=0;i<4;++i)
			tempipaddrc[i] = 0x00;
		PETER = DHCP_INIT;
	case DHCP_INIT:
		printf("\r\nDHCP INIT..");
		for(i=0;i<6;++i)
			svrmacaddrc[i] = 0xFF;
		for(i=0;i<4;++i)
			svridc[i] = 0xFF;
		msecs_timer = 0;
		PETER = DHCP_WAIT;
	case DHCP_WAIT:
		if(msecs_timer >= 2000)
			PETER = DHCP_BROADCAST;
	break;
```

Note that the names within the braces are constant despite what the variable name happens to be. The typedef doesn't reserve memory, which means we can create a number of variables that will use the elements of the enumeration list behind the DHCP_STATE_LIST type by simply declaring a new variable name of the DHCP_STATE _LIST type.

That's pretty fancy stuff and it's a pitfall for us. Here is what we have to code in PICBASIC PRO to build our enumerated DHCP state list:

```
;DHCP STATES
DHCPSTATE    VAR     byte
DHCP_ENTRY           CON    $0
DHCP_INIT            CON    $1
DHCP_WAIT            CON    $2
DHCP_BROADCAST       CON    $3
DHCP_DISCOVER        CON    $4
DHCP_REQUEST         CON    $5
DHCP_BIND            CON    $6
DHCP_BOUND           CON    $7
DHCP_DISABLED        CON    $8
```

The DHCPSTATE variable is a byte that is loaded with one of the constant values of the manually enumerated list of DHCP states. This is a rendition of the C language typedef gone Basic. Despite the fancy trappings of the C typedef keyword, the PICBASIC PRO DHCP STATES definitions are basically all that is taking place under the covers.

The seemingly simple DHCPSTATE case statements we've been looking at are full of PICBASIC PRO potholes. The PICBASIC PRO code that follows is the pothole filler:

```
select case  DHCPSTATE
       case DHCP_ENTRY
             hserout[13,10,"DHCP RESET.."]
             for i8 = 0 to 3
               tempipaddrc[i8] = $00
             next i8
             DHCPSTATE = DHCP_INIT
       case DHCP_INIT
             hserout[13,10,"DHCP INIT.."]
             for i8 = 0 to 5
               svrmacaddrc[i8] = $FF
             next i8

             for i8 = 0 to 3
               svridc[i8] = $FF
             next i8

             msecs_timer = 0
             DHCPSTATE = DHCP_WAIT
       case DHCP_WAIT
             if msecs_timer >= 2000  then
               DHCPSTATE = DHCP_BROADCAST
             endif
end select
```

Let's take in the converted DHCPSTATE comparison code line by line. The C switch statement must be replaced by the PICBASIC PRO select case equivalent. What you will find during the port is that most of the C braces ({}) will either be eliminated or replaced by PICBASIC PRO keywords such as end select, end if, next, and then. Parenthesis normally are not required for PICBASIC PRO keyword use but are used identically as they would be in the C language, in cases where the compiler's mathematical operator priority may override the programmer's desired execution order within a line of PICBASIC PRO source code. All of the C printf statements must be altered to the PICBASIC PRO hserout format with "13,10" replacing the C language "\r\n" carriage return and line feed characters. The C language for statements lack the next keyword required by PICBASIC PRO. Thus, we must also port all of the C language for statements to the PICBASIC PRO for/next format.

The break keyword is a necessary part of the C language switch concept. Normally, the break keyword is always used to end a case statement in C. Note the absence of break keywords in our example. The absence of the break keyword allows the logic to flow into the next case statement without first exiting the C switch function. We will have to deal with that logic as we encounter it. That could mean not using the PICBASIC PRO select case functionality in that part of the ported code.

Definitions and declarations of variables and constants are also a potential problem when porting an application of this size. I easily exceeded the PICBASIC PRO maximum number of DEFINE keywords in this first pass of the port. Consider this line of C code from the Ethernet MINI driver:

```
#define DHCP_DISCOVER_MESSAGE    0x01
```

There are a large number of these types of definitions in the original Ethernet MINI C source code. Instead of using a PICBASIC PRO DEFINE keyword in the port, I called upon the PICBASIC PRO CON keyword construct as shown in this line of ported PICBASIC PRO source code:

```
DHCP_DISCOVER_MESSAGE            CON     $01
```

Function is yet another word you won't find in the PICBASIC PRO programmer's guide when used in the context of PICBASIC PRO programming. Here is a C function that we must port to equivalent PICBASIC PRO code:

```
void wr_phy(char reg, unsigned int data)
{
       // Write the register address
       MIREGADR = reg;
       NOP();
       // Write the data
       // Order is important: write low byte first,
       // high byte last
       MIWRL = LOW_BYTE(data);
       NOP();
       MIWRH = HIGH_BYTE(data);
       NOP();
       // Wait until the PHY register has been written
       while(BUSY);
}
```

Lots of gotchas here. First of all, C functions allow the use of input variables such as the eight-bit character function variable reg and the 16-bit integer function variable data. We'll also have to deal with the macros LOW_BYTE and HIGH_BYTE here, as well. No worries. The PICBASIC PRO port follows:

```
;preload the values needed within the wr_phy subroutine
rgstr = PHCON2
data16 = $0110
gosub wr_phy             ;wr_phy(PHCON2, $0110)

wr_phy:
       ; Write the register address
       MIREGADR = rgstr;
@      nop
       ; Write the data
       ; Order is important: write low byte first, high
byte last
       MIWRL = data16 & $00FF           ;LOW_BYTE(data)
@      nop
       MIWRH = (data16 & $FF00) >> 8    ;HIGH_BYTE(data);
@      nop
       ; Wait until the PHY register has been written
       while BUSY
       wend
       return
```

On the PICBASIC PRO side, we simply turned the wr_phy C function into a PICBASIC PRO wr_phy subroutine. In this case, the original C function did not return a value to the caller. Some of the Ethernet MINI driver C functions do indeed return a value. If a return value is required, an eight-bit or 16-bit variable is declared and inserted into the ported PICBASIC PRO subroutine to

TABLE 1

Address	Symbol Name	Value
ED9	EPKTCNT	0x04
085	_packetheader	0x54
2FB	_packet	0x00
03A	_rxlen	0x4E

■TABLE 1. I punched up the necessary Watch variables to get me to the potential data I hoped to receive from the laptop PINGs. The Watch window also allows the programmer to see directly into the PIC18F67J60's Ethernet registers. I wish I had that capability when I was writing the initial ENC28J60 driver.

LISTING 1

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\FE>ping 192.168.1.150

Pinging 192.168.1.150 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.1.150:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\Documents and Settings\FE>ping 192.168.1.150

Pinging 192.168.1.150 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.1.150:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\Documents and Settings\FE>ping 192.168.1.150

Pinging 192.168.1.150 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.1.150:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\Documents and Settings\FE>arp /all

Displays and modifies the IP-to-Physical address translation tables used by
address resolution protocol (ARP).

ARP -s inet_addr eth_addr [if_addr]
ARP -d inet_addr [if_addr]
ARP -a [inet_addr] [-N if_addr]

  -a            Displays current ARP entries by interrogating the current
                protocol data.  If inet_addr is specified, the IP and Physical
                addresses for only the specified computer are displayed.  If
                more than one network interface uses ARP, entries for each ARP
                table are displayed.
  -g            Same as -a.
  inet_addr     Specifies an internet address.
  -N if_addr    Displays the ARP entries for the network interface specified
                by if_addr.
  -d            Deletes the host specified by inet_addr. inet_addr may be
                wildcarded with * to delete all hosts.
```

continued ...

■ LISTING 1. ARP and IPCONFIG are good friends to have when you're tracing through who's who in IP and MAC land. The commands that you will use most are arp — a, which dumps the PC's ARP cache and IPCONFIG /all, which spills the PC's important network information.

mimic the C function returned value. The original C function input variables are replaced by the variables rgstr (an eight-bit value whose name is short for register) and data16 (a 16-bit value). The PICBASIC PRO subroutine variables are preloaded before the call is made to the PICBASIC PRO subroutine.

The original Ethernet MINI driver C source is full of C comment tags (//) which must all be converted to the PICBASIC PRO comment tag (;). The global replacement capability offered by the MPLAB IDE source code editor comes in handy for this job. However, one must be careful as the global replacement utility can be really dumb and change things you don't want it to touch. The C NOP statements can also be globally mutated with the same comment tag replacement caveat holding true.

Another feature of the C programming language is its inclusion of pointers. The closest we get to a pointer in PICBASIC PRO is an array. So, C pointers — which are preceded by the '*' character — are emulated with values within variables on the PICBASIC PRO side of the port. For example, a C-to-PICBASIC PRO byte pointer conversion will typically look like this:

```
char *bufferptr        //C pointer
==
bufferptr var byte    ;PICBASIC PRO
                      ;pointer
```

There are many other C language concerns we must engage and conquer and, if we have not already discussed them, I will address them as we come to them in the process of the port. Not everything we will have

to deal with will be related to the nuances of C or PICBASIC PRO. I'll leave you with a bug that ate my lunch and licked the pail. Can you see what is wrong with this line of code? It compiles perfectly:

```
MIWRH = (data16 & $FF00) > 8
;HIGH_BYTE(data)
```

This is what happens when you sit at the porting table a few days too many. Here is what that "good" line of code should look like:

```
MIWRH = (data16 & $FF00) >> 8
;HIGH_BYTE(data)
```

I caught this as I single-stepped through the wr_phy subroutine. I was looking to find out why the high byte of the integer stored in the data16 variable wasn't being read back correctly after I supposedly loaded it correctly. The greater than (>) and right shift (>>) operators are identical in C and PICBASIC PRO. The error occurred when I manually entered the greater than operator (>) incorrectly instead of the right shift operator (>>). The PICBASIC PRO code I manually and incorrectly entered was to replace the C macro HIGH_BYTE(data), which is used all over the place. So, to save some keystrokes, I copied the incorrect line of code into every HIGH_BYTE(data) macro line in the PICBASIC PRO ported code. Enough said.

STAGE 1

So far, I've successfully ported the EUSART driver and the PIC18F67J60 initialization driver. I've also ported all of the arrays, pointers, constants, and macros associated with the ported driver components. There is just enough code to bring the Ethernet MINI online.

To test my work up to this point, I also ported and enabled the get_frame function, which loads an incoming frame into the Ethernet MINI driver's packet memory. Since

Listing 1 continued ...

```
  -s             Adds the host and associates the Internet address inet_addr
                 with the Physical address eth_addr.  The Physical address is
                 given as 6 hexadecimal bytes separated by hyphens. The entry
                 is permanent.
  eth_addr       Specifies a physical address.
  if_addr        If present, this specifies the Internet address of the
                 interface whose address translation table should be modified.
                 If not present, the first applicable interface will be used.
Example:
  > arp -s 157.55.85.212   00-aa-00-62-c6-09  .... Adds a static entry.
  > arp -a                                     .... Displays the arp table.

C:\Documents and Settings\FE>arp

Displays and modifies the IP-to-Physical address translation tables used by
address resolution protocol (ARP).

ARP -s inet_addr eth_addr [if_addr]
ARP -d inet_addr [if_addr]
ARP -a [inet_addr] [-N if_addr]

  -a             Displays current ARP entries by interrogating the current
                 protocol data.  If inet_addr is specified, the IP and Physical
                 addresses for only the specified computer are displayed.  If
                 more than one network interface uses ARP, entries for each ARP
                 table are displayed.
  -g             Same as -a.
  inet_addr      Specifies an internet address.
  -N if_addr     Displays the ARP entries for the network interface specified
                 by if_addr.
  -d             Deletes the host specified by inet_addr. inet_addr may be
                 wildcarded with * to delete all hosts.
  -s             Adds the host and associates the Internet address inet_addr
                 with the Physical address eth_addr.  The Physical address is
                 given as 6 hexadecimal bytes separated by hyphens. The entry
                 is permanent.
  eth_addr       Specifies a physical address.
  if_addr        If present, this specifies the Internet address of the
                 interface whose address translation table should be modified.
                 If not present, the first applicable interface will be used.
Example:
  > arp -s 157.55.85.212   00-aa-00-62-c6-09  .... Adds a static entry.
  > arp -a                                     .... Displays the arp table.

C:\ >arp -a

Interface: 192.168.0.100 --- 0x3
  Internet Address      Physical Address      Type
  192.168.0.1           00-14-bf-9a-2e-41     dynamic

C:\ >arp -s 192.168.0.150 00-00-45-44-54-50

C:\ >arp -a

Interface: 192.168.0.100 --- 0x3
  Internet Address      Physical Address      Type
  192.168.0.1           00-14-bf-9a-2e-41     dynamic
  192.168.0.150         00-00-45-44-54-50     static

C:\ >ping 192.168.0.150

Pinging 192.168.0.150 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.150:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),
```

BUFFER DUMP 1

```
Address  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII

  2F0    8C CE 0B 32 59 4D 7B 17 DF 89 CD 00 00 45 44 54 ...2YM{. .....EDT
  300    50 00 11 25 18 0A DB 08 00 45 00 00 3C 4A 9C 00 P..%.... .E..<J..
  310    00 80 01 6D DA C0 A8 00 64 C0 A8 00 96 08 00 3D ...m.... d......=
  320    5C 03 00 0D 00 61 62 63 64 65 66 67 68 69 6A 6B \....abc defghijk
  330    6C 6D 6E 6F 70 71 72 73 74 75 76 77 61 62 63 64 lmnopqrs tuvwabcd
  340    65 66 67 68 69 07 18 DF 2B 41 43 41 43 41 43 41 efghi... +ACACACA
  350    43 41 43 41 00 20 45 46 45 45 46 45 46 41 43 41 CACA. EF EEFEFACA
  360    43 41 43 41 43 41 43 41 43 41 43 41 43 41 43 41 CACACACA CACACACA
  370    43 41 43 41 42 4F 00 FF 53 4D 42 25 00 00 00 00 CACABO.. SMB%....
  380    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........ ........
  390    00 00 00 00 00 00 00 11 00 00 21 00 00 00 00 00 ........ ..!.....
  3A0    00 00 00 00 E8 03 00 00 00 00 00 00 00 00 21 00 ........ ......!.
  3B0    56 00 03 00 01 00 00 00 02 00 32 00 5C 4D 41 49 V....... ..2.\MAI
  3C0    4C 53 4C 4F 54 5C 42 52 4F 57 53 45 00 0F 00 80 LSLOT\BR OWSE....
  3D0    FC 0A 00 53 4E 4F 4F 50 00 00 00 00 00 FC 40 A1 ...SNOOP ......@.
```

■ BUFFER DUMP 1. Don't worry. We won't be reading dumps all of the time. When we start porting the protocol modules (TCP, UDP, etc.), I'll break out the Network General Ethernet Sniffer.

I have not yet ported the ARP and DHCP drivers, I had to cheat to get results. Take a look at Listing 1, which is a printout of the actual commands I entered in a Windows XP command prompt window.

A Linksys router based at 192.168.0.1 and set up as a DHCP server networks my laptop and the Ethernet MINI. I issued the command to display the contents of the laptop's

ARP cache (arp -a). At this point, only the Linksys router is cached by the laptop (192.168.0.1). Since the Ethernet MINI's IP and MAC addresses are not in my laptop's ARP cache, my laptop will issue an ARP request to the Ethernet MINI before issuing the ICMP Echo (PING).

To fake out my laptop and prevent it from issuing an ARP request to the Ethernet MINI, I preloaded my laptop's ARP cache with the Ethernet MINI's IP and MAC addresses (arp -s). After checking the laptop's ARP cache to see if my static entries made it in, I PINGED the Ethernet MINI. As you can see in Listing 1, the four PINGS went unanswered as there is no ICMP code running on the Ethernet MINI yet.

At this point, I really don't care about answering the PINGs as I'm jumping up and down in the EDTP shop with excitement. I'm jumping around looking at the contents of Table 1, which is the text from my MPLAB IDE Watch window. The EPKTCNT register in the MPLAB IDE Watch window registers 0x04 representing all four ICMP Echo messages from the laptop, which are now stored in the PIC18F67J60's receive buffer. I also have a reasonable receive length value returned in the rxlen variable. This is good.

I added the packet and packet_header array watch entries to get the beginning addresses of the arrays. We won't take a close look at the packet_header array as it has already given us some good information via the rxlen variable, which is filled from the packet length slot of the packet_header array. The proof in the pudding is shown in BufferDump 1. The presence of the alphabet in the packet clues me that this may be an ICMP packet without going any further. Just for grins let's pick through the dump and see if I am right.

The first thing we should receive is the destination hardware (MAC) address which, in this case, is the Ethernet MINI's MAC address. Just so happens that beginning at offset 0x02FB in BufferDump 1 you can make out the Ethernet MINI's MAC address (00 00 45 44 54 50). The sender's MAC address should immediately follow and it does as my laptop's MAC address is 00 11 25 18 0A DB. Offset 0x0312 is equal to 0x01, which indicates that this is an ICMP packet. The sender's IP address (my laptop IP address) should be located at 0x0315 (C0 A8 00 64 or 192.168.0.100) and the receiver's IP address (the Ethernet MINI's IP address) should immediately follow (C0 A8 00 96 or 192.168.0.150). Another immediate clue to me that I had a chance of having caught a good packet is the presence of "SNOOP" in the buffer area. SNOOP is my laptop's network name.

SOURCES

■ *HI-TECH* (**www.htsoft.com**) — HI-TECH PICC-18 C compiler

■ *microEngineering Labs* (**www.melabs.com**) — PICBASIC PRO

■ *Microchip* (**www.microchip.com**) — MPLAB ICD 2; MPLAB IDE; PIC18F67J60

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

THE NEXT CYCLE

Now that we know we can receive Ethernet frames, all we have left to do is automate the dump parsing process we just performed on BufferDump 1. I'll post the ported portion of the PICBASIC PRO Ethernet MINI driver source code for you on the *Nuts & Volts* website (**www.nutsvolts.com**). You may also get the code package from the EDTP website (**www.edtp.com**).

When we meet again, we will spin up the Design Cycle to port and bring up the ARP and UDP modules so we can port and enable the Ethernet MINI's DHCP engine. **NV**

CONTACT THE AUTHOR

■ *Peter Best can be reached via email at peter@nerdvilla.com.*