FEBRUARY 1991

NOVELL® RESEARCH

The NetWare DOS Client Environment

Matt Hagen
Consultant
Systems Engineering Division

Abstract:

This AppNote explores several aspects of NetWare IPX and the NetWare DOS shell. It defines the two major roles of the shell. It provides two small programs that aid readers in identifying NetWare IPX and shell interrupts. And it explains how to write C Language and assembly language routines that access shell services.

Contents

Introduction

The term NetWare used to be easy to define. In the early 1980s when a small consulting firm called Superset (working for Novell) first developed the NetWare operating system, the term NetWare referred to a network operating system running in one box and a piece of software called the shell running alongside DOS in several other boxes. The box with the operating system was called the "file server" and the other boxes were called the "clients." Each box contained a network interface card (NIC) along with the software driver necessary to perform I/O to the card. The NICs were connected via cables. The clients sent "requests" to the server and the file server sent "responses" back to the clients. Typically a client would request a file and the server would download the file to the client as a series of packets.

Today, "NetWare" describes more than just a DOS-world software product that enables file sharing among DOS clients. Instead it describes a continuum of products that facilitate the sharing of sundry resources among many client operating systems for a variety of reasons in a host of settings. Even the NetWare DOS client architecture has changed considerably with the addition of the Link Support Layer (LSL) which allows a client workstation to support communication protocol stacks other than the traditional NetWare IPX standard.1

In spite of this tremendous technological evolution, the current NetWare DOS client strategies still bear the mark of the initial implementation, a tribute to the original design. This article describes some of these fundamental strategies.

Throughout this article, "local" refers to the DOS client workstation and "remote" refers to some other box on the network.

NetWare Software for DOS Clients

NetWare requires that a DOS client run a number of terminate-and-stay-resident programs in workstation RAM. In the traditional DOS client model each workstation runs a shell (ANET.COM or NET.COM) and a NetWare IPX module (IPX.COM) which includes both NetWare IPX code and LAN driver code linked together. The DOS ODI model requires a workstation to run a LAN driver, the LSL, NetWare IPX, and the shell.2

The LAN driver sends and receives data via the network interface card by copying the data from workstation RAM to the card and vice versa.

The LSL forms a convenient interface between the LAN drivers and communication protocol stacks like NetWare IPX. It allows one LAN driver to service more than one stack.

NetWare IPX is a communication protocol stack that provides communication services to applications that want to send NetWare IPX-style packets on the network. Applications that use the NetWare IPX interface must build each packet before calling the NetWare IPX interface to send the packet.

The shell performs two major roles: First, the shell interrogates application-generated system calls (to DOS or BIOS) to determine whether each request concerns local resources or remote resources. The most notable examples are the DOS Int 21h file manipulation calls and the BIOS Int 17h printer calls. If the shell detects that a call concerns a remote resource, it builds a request packet, calls the NetWare IPX interface to send the packet to a server, waits for a response packet, receives the response packet, interprets the response, and hands the interpreted response to the application. This is accomplished without the knowledge of the requesting application.

Second, the shell provides a battery of NetWare-specific services that allow NetWare DOS utilities (like SYSCON, FILER, PCONSOLE, and MAP, as well as third-party NetWare applications) to manipulate file server binderies, queues, directories, and files, control network printers, send messages, return file server statistics, and so forth.

More About the Two Roles of the Shell

In some ways both roles of the NetWare DOS shell are similar. Both require the shell to service application-generated requests. Both require the shell to capture certain interrupts (as we will see later). Both require the shell to build NetWare IPX packets and call the NetWare IPX interface to access remote resources (on behalf of the requesting application).

However, the two roles are also quite distinct. The first role involves resources that commonly exist on standalone DOS machines and network servers (e.g. files and printers). In this case, the shell must see certain application-generated requests before DOS or BIOS see them so that the shell can determine whether the request involves a local resource or a remote resource and then either give the request to DOS or BIOS (in local RAM) or send the request to a remote box. This role requires the shell to quietly and unobtrusively watch over the shoulder of DOS and BIOS looking for network requests. The second role involves NetWare-exclusive resources that exist either in the shell itself (e.g. shell statistics) or at remote servers (e.g. file server binderies and network queues). In this case, the shell forgoes any cloak-and-dagger activity and simply processes the request or sends the request to the target remote server.

From another standpoint, the first role involves implicit calls to the shell where the calling application is unaware that it is accessing the shell. The DOS utility DIR (written to work on a standalone machine before the advent of NetWare) makes implicit calls to the shell when a user requests the directory listing of a network drive. The second role involves explicit calls to the shell where the calling application is aware that it is accessing the shell. The NetWare utility SYSCON makes explicit calls to the shell when it creates a bindery object.3

An Implicit Shell Call

To better understand an implicit shell call, let's look at a utility like DIR that searches a directory table for file entries. Let's also suppose that our default drive is F. To find a file the application issues a FindFirst (AH=4Eh) Int 21h DOS call to search a disk for the target file. The shell traps the call, recognizes that the target file can only reside on a file server's (remote) disk, builds a packet, passes the packet to NetWare IPX for transmission to the server, and then waits for a response. Meanwhile the server receives the packet, searches its own disk for the target file, and then issues a response to the client. (The response indicates that the server either found the file or that it didn't find the file). The client's NetWare IPX module receives the response from the server and passes it to the shell which passes it to the requesting application. In this case, as in all cases involving implicit shell calls, the application is unaware that the shell redirected the FindFirst request to a remote resource. As far as the application is concerned, it (the application) accessed a local disk.

Interrupts

We mentioned above that both roles of the shell require the shell to capture certain key interrupts. Let's look more closely at this intriguing concept.

Most Intel-based computers contain an interrupt vector table consisting of 1024 bytes of 256 4-byte interrupt vectors at the beginning of RAM. Each interrupt vector consists of the two-byte segment and the two-byte offset of a routine somewhere else in computer memory. Some vectors point to routines in ROM BIOS. Others point into DOS. If an interrupt-driven LAN driver resides in workstation memory, at least one of the vectors points to the driver's interrupt service routine (for packet reception and transmit completion).4

Applications that alter the interrupt vector table can do so by hooking an interrupt vector or by capturing (sometimes called chaining) an interrupt vector. When an application hooks an interrupt vector it stamps the far address of one of its routines (an interrupt service routine or ISR) into the vector without bothering to save the prior contents of the vector. When an application captures an interrupt vector it saves the existing far pointer (into its own data space) before stamping the vector with a new far pointer. Capturing a vector allows the current ISR to pass an interrupt request along to the prior ISR if the request is not meant for the current ISR. Using this technique several ISRs can share the same interrupt vector. BIOS and DOS usually hook vectors. Typically, LAN drivers hook vectors, too. NetWare IPX and the shell, as we will see in a minute, capture interrupts.

SAVE and COMPARE, two programs provided with this article, allow you to isolate the interrupts that any program hooks or captures. (The listings for both programs appear near the end of this article.) SAVE copies the contents of the first 48 interrupt vectors to a file called VECTOR.DAT. COMPARE compares the current interrupt vector table in RAM with the contents of the file. Neither program accepts command line arguments. You can use these two programs in a variety of ways. You must copy both SAVE.EXE and COMPARE.EXE to a local drive before using them to analyze NetWare IPX or the shell.

To see which interrupts NetWare IPX captures, complete the following steps:

1. Load the LSL and a LAN driver (if you are using DOS ODI software).
2. Run SAVE.
3. Load NetWare IPX.
4. Run COMPARE.

My results look like this:

```
Vector 8h = 14F8:0A7A
          Vector 2Fh = 14F8:08A0
```

Note that the segment values are the same indicating that both handlers exist in the same program (NetWare IPX). By the way, if you are not using DOS ODI software, your results will include the LAN driver's hardware interrupt vector as well.

To do the same for the shell, complete these steps:

1. Load the LSL and a LAN driver (if you are using DOS ODI software), and then load NetWare IPX.
2. Run SAVE.
3. Load NET3.
4. Run COMPARE.

This time my results look like this:

```
Vector 10h = 188B:0684
          Vector 17h = 188B:5E26
          Vector 1Bh = 188B:0695
          Vector 20h = 188B:06A8
          Vector 21h = 188B:06AA
          Vector 24h = 188B:06CE
          Vector 27h = 188B:0801
```

Remember, each SEG:OFF pair is a far pointer to an interrupt service handler inside NetWare IPX and the shell respectively.

You can use these results in a number of ways. For example, to view the shell's Int 21h handler I complete the following steps:5

1. Enter DEBUG by typing the following:

   ```
   debug <Enter>
   ```

2. Enter the following command at the debug `-' prompt (you would use your Int 21h value instead of mine):

```
u 188B:06AA <Enter>
```

Undoubtedly, NetWare IPX and the shell capture each interrupt vector listed above for a good reason. This article, however, deals with only two of the captured interrupts: Int 2Fh for NetWare IPX and Int 21h for the shell. Note that both NetWare IPX and the shell "capture" the interrupts as opposed to "hooking" them.

Interrupt 2Fh

DOS calls Int 2Fh the Multiplex Interrupt. Int 2Fh is available for any application to capture so long as the new ISR agrees to pass any unwanted interrupt requests along to the prior ISR. NetWare IPX captures Int 2Fh as a means of making available to interested applications the whereabouts of its service entry point. To find the NetWare IPX service entry point in workstation RAM an application need only load AH with 7Ah and AL with 00h, and then call Int 2Fh. The NetWare IPX Int 2Fh ISR checks the AX register for these telltale values. If AX does not contain 7A00h (or one of a few other values), the NetWare IPX ISR passes the interrupt request along to the captured ISR. If AX does contain the right value, NetWare IPX returns the SEG:OFF of its service entry point in ES:DI.7

The NetWare IPX service entry point is an assembly language API with rules defined by NetWare IPX. To utilize NetWare IPX services an application must load BX with a value (indicating the target NetWare IPX service), load other registers as necessary, and then make a far call to the NetWare IPX service entry point. Here are some examples of NetWare IPX services:

BX = 00h means NetWare IPX Open Socket
BX = 01h means NetWare IPX Close Socket
BX = 02h means NetWare IPX Get Local Target
BX = 03h means NetWare IPX Send Packet
BX = 04h means NetWare IPX Listen For Packet

To view the NetWare IPX ISR for interrupt 2Fh I would use DEBUG.COM to complete the following steps:

1. Enter DEBUG by typing the following:

   ```
   debug <Enter>
   ```

2. Enter the following command at the debug `-' prompt (again, you would use your Int 2Fh value instead of mine):

   ```
   u 14F8:08A0 <Enter>
   ```

As you look at this assembly code you will notice that the NetWare IPX ISR checks the AX register for the value 7A00h (as we discussed above) in the very first instruction.

DOS Interrupt 21h

At a DOS machine, interrupt 21h is an extremely important interrupt.8 When DOS bootstraps into workstation RAM it hooks Int 21h and points it to a DOS ISR that handles almost all application-generated requests for DOS services. Applications access this interrupt by loading AH with a value (00h..7Fh) indicating the desired DOS service and then executing an Int 21h instruction.

By capturing this interrupt, the shell establishes a means to accomplish its two primary objectives. First, it is able to see each request for a DOS service before DOS does and then either pass the request on to DOS, service the request itself, or send the request to a file server depending on the nature of the request. Second, since the DOS Int 21h routine does not utilize AH values greater than 7Fh, the shell uses the values 80h..FFh to allow applications to make explicit shell requests using Int 21h. To fulfil its first role the shell has no other choice than to capture Int 21h. To fulfil its second role, capturing Int 21h is merely a convenience.

Like NetWare IPX, the shell Int 21h service entry point is an assembly language API with rules dictated to an extent by DOS and further defined by the shell.9 To utilize shell services an application must load AH with a value (indicating the target shell service), load other registers as necessary, and then execute an Int 21h instruction. Here are some examples of shell services:

AH = B6h means get a file's extended file attributes
AH = D7h means logout the workstation from all attached file servers
AH = E7h means get a file server's current date and time

FIND, another program provided with this article, is intended to help you view the DOS Int 21h ISR (when the shell is not loaded) and the shell Int 21h ISR (when the shell is loaded) in execution. (A listing is included near the end of this article.) FIND is a simple assembly language program that makes three calls to the DOS Int 21h handler. The first sets the DTA, the second finds the first file in the current directory, and the third displays the filename on the screen. If you run FIND outside of DEBUG it will display on your computer screen the first file that it finds in the current directory. To view the DOS Int 21h ISR, complete the following steps:

1. Copy FIND.EXE and DEBUG.EXE to a local drive and make sure the shell is not loaded.
2. Enter the following:

   debug find.exe <Enter>
3. Enter the following `proceed' instruction repeatedly until you get to the second INT instruction:

   p <Enter>
4. Enter the following `trace' instruction to trace inside the ISR:

   t <Enter>
5. Enter the following `unassemble' instruction to look at the ISR code:

   u <Enter>
6. Use `p' and `t' to trace through the ISR (in execution) for awhile.
7. Enter the following instruction to `quit' debug:

   q <Enter>

To view the shell Int 21h ISR, load the shell and repeat steps 2 through7. If you run FIND from a local drive the shell ISR will give the request back to DOS. If you run it from a network drive the shell ISR will eventually create a request packet and send it to the server. (Alas, if you try to follow a FIND request on a network drive your workstation will hang before you see the request packet sent to the server. This has something to do with the sticky problem of timeouts.)

NetWare C Libraries

Since the entry points for both NetWare IPX and the shell are assembly language APIs (and somewhat clumsy to access from a C program), Novell provides a C-Interface library to third-party developers that want to access NetWare IPX and/or shell services. The library is available in four compiler versions: Microsoft, Borland, Lattice, and Watcom. Each compiler version includes the following four memory model sizes: small, medium, compact, and large. Novell actually ships the Watcom development tools with the Watcom version of the DOS C-Interface library as a complete package.10

Rather than discussing the API or the internals of the NetWare C for DOS library, or the (slight) differences among the four compiler versions, these last sections provide sample code that might help you write your own C-Interface library to NetWare IPX and the shell or (more realistically) at least write a few routines that you may feel are missing from the current API.11

The routines that you write should (in my estimation) meet the following criteria:

1. Each routine should be independent of any one memory model.
2. Code should be written in C unless assembly language is absolutely vital.
3. Code should try to be compiler-independent.
4. A routine should not duplicate what an existing API already does.

These routines will deal with the following three interfaces:

1. The first is the assembly language interface provided by NetWare IPX and the shell where you load BX and make a far call to NetWare IPX or else load AX and call the shell's Int 21h ISR.
2. The second is the C language interface that you create by writing a few assembly language routines that call the interface described in step 1.
3. The third is the C language interface that you create by writing C language routines that call the interface described in step 2. An application that you might be writing would use this third interface.

In the next section I will describe an API that is missing from the current NetWare C for DOS library and then provide two routines, one in assembly and one in C, that fit the bill.

Building Your Own Routines

When building an NetWare IPX packet on a client workstation destined for an NLM on a file server, it is convenient (and faster) to be able to obtain the server's network and node address and the immediate address relative to the workstation without querying the server's bindery for the information. The client's shell actually has the needed information. The two routines described in this section allow a developer to extract this information from the shell.

The DOS shell maintains several tables that applications can query. Two of these tables are the File Server Name Table and the Connection ID Table. Each of these tables has eight entries. The File Server Name Table consists of an array of eight 48-byte character buffers. Each buffer accommodates one asciiz server name. The Connection ID Table consists of an array of eight 32-byte buffers. Each buffer accommodates information about a server like the server's internetwork address and timeout information. The two tables work in concert. When the shell attaches to a server, the shell places the server name in an available slot in the File Server Name Table. The shell also places information about the server in the corresponding slot in the Connection ID Table. The fact that these two tables are eight entries in size is the reason that a DOS client workstation can attach to a maximum of eight servers.

The two routines provided with this article, GetConnectionIDTableEntry() and GetServerAddress(), allow a client application to obtain a server's network, node, and immediate addresses from the shell without sending a request to the server. I've also included a small program that calls GetServerAddress() and then displays the network, node, and immediate addresses for each of the servers in the shell's Connection ID Table. (The listings for these routines appear near the end of this article in two files: GETCONN.ASM and ADDRESS.C.)12

GetConnectionIDTableEntry() calls the shell Int 21h ISR, returns a pointer to the Connection ID Table, and then copies the appropriate entry into a buffer. It also provides a C language interface to GetServerAddress().

GetServerAddress() calls GetConnectionIDTableEntry() to obtain the specified entry in the Connection ID Table. Then it extracts the network, node, and immediate addresses from the table and copies them into buffers for the calling program.

These two routines describe in miniature how a C-interface library written to the NetWare IPX and shell assembly language interfaces might work.

Listings

This section contains all the listings for all the programs and routines mentioned in this article. SAVE and COMPARE were created with Watcom tools. FIND, GETCONN, and ADDRESS were created with Borland tools. Here is the order of the listings:

1. SAVE.C
2. COMPARE.C
3. FIND.ASM
4. GETCONN.ASM
5. ADDRESS.C

```
/*************************************************************************

* SAVE.C
*************************************************************************/

-include <stdio.h>
-include <stdlib.h>

-define DATAFILE vector.dat
-define HANDLER_COUNT 48

/*************************************************************************
* main
*************************************************************************/


main()
{
            void far *vector=NULL;
            void (far *array[HANDLER_COUNT])();
            FILE *f;

      _fmemmove((void far *)array,vector,(sizeof(void (far*) ())) *HANDLER_COUNT);

      f=fopen(DATAFILE,wb);
            if(f==NULL)
            {

      printf(Cannot open file for write.\n);

      return;
            }

      fwrite(array,sizeof(void (far *)()),HANDLER_COUNT,f);

      fclose(f);
```

```
    }

    /*************************************************************************/
    /*************************************************************************/



/*************************************************************************
* COMPARE.C
*************************************************************************/

-include <stdio.h>
-include <stdlib.h>

-define DATAFILE vector.dat
-define HANDLER_COUNT 48

/*************************************************************************
* main
*************************************************************************/

main(
        int argc,
        char *argv[])
{
        int a;
        void far *vector=NULL;
        void (far *handler)()=NULL;
        void (far *array[HANDLER_COUNT])();
        FILE *f;

    _fmemmove((void far *)array,vector,(sizeof(void (far *)())) *HANDLER_COUNT);

    f=fopen(DATAFILE,rb);
        if(f==NULL)
        {

    printf(Cannot open file for read.\n);

    return;
        }

    for(a=0;a<HANDLER_COUNT;a++)
        {

    fread(&handler,sizeof(void (far *)()),1,f);

    if(handler!=array[a])

    printf("Vector %Xh = %FP\n",a,array[a]);
        }

    fclose(f);
    }

    /*************************************************************************/
    /*************************************************************************/
```

```
;***************************************************************************
; FIND.ASM ;
;***************************************************************************

DTAStructure   struc
Reserved db   21 dup (?)
Attributedb   ?
Time     dw   ?
Date     dw   ?
LowSize  dw   ?
HighSize dw   ?
Name     db   13 dup (?)
DTAStructureends

DOSSEG
.MODELSMALL
.STACK100h
.DATA

DTA  DTAStructure <>
File db   '*.*', 0

.CODE

;***************************************************************************
; FindFirstFile
;***************************************************************************

FindFirstFileprocfar
            push ds
            xor  ax, ax
            push ax
            mov  ax, @data
            mov  ds, ax

      mov  ah, 1ah
            lea  dx, DTA
            int  21h

      mov  ah, 4eh
            xor  cx, cx
            lea  dx, File
            int  21h

      mov  ah, 02h
            lea  bx, DTA.Name

mDisplayName:
            mov  dl, [bx]
            or   dl, dl
            jz   Exit
            int  21h
            inc  bx
```

```
        jmp  DisplayName

Exit:
        ret
FindFirstFileendp
        end

;************************************************************************
;************************************************************************


;************************************************************************
; GETCONN.ASM ;
;************************************************************************

ConnIDTable     struc
InUse           db      ?
Order           db      ?
Net             db      4 dup (?)
Node            db      6 dup (?)
Socket          db      2 dup (?)
Timeout         db      2 dup (2)
ImmedAddr       db      6 dup (?)
Sequence        db      ?
ConnectionNum     db    ?
ConnectionStatus  db    ?
MaxTimeout        db    2 dup (?)
Reserved        db      5 dup (?)
ConnIDTable    ends

DOSSEG
.MODELSMALL
.DATA
.CODE
        public  _GetConnectionIDTableEntry

;************************************************************************
; int GetConnectionIDTableEntry(
;    WORD connID,
;    ConnIDTable *entry) ;

; connID is an index (1..8) into the table. entry is a pointer to a buffer
; big enough to hold a ConnIDTable structure. The routine copies the
; specified Connection ID Table entry into the buffer. The routine returns
; a value of 0 for successful and a non-zero value if connID is an invalid
; number.
;************************************************************************

GetConnectionIDTableEntryproc
        ARG  connID:WORD, entry:DATAPTR
        push bp
        mov  bp, sp
        push si
        push di
        push ds
```

```
    mov  ax, connID
        dec  ax
        cmp  ax, 7
        ja   InvalidIndex

    mov  bl, size ConnIDTable
        mul  bl
        mov  bx, ax

    mov  ax, 0ef03h
        int  21h

    add  si, bx
        mov  bx, es

IF   @DataSize EQ 0
        mov  di, entry
        mov  ax, ds
        mov  es, ax
ELSE
        les  di, entry
ENDIF
        mov  ds, bx
        mov  cx, size ConnIDTable
        shr  cx, 1
        rep  movsw
        xor  ax, ax

InvalidIndex:
        pop  ds
        pop  di
        pop  si
        pop  bp
        ret
GetConnectionIDTableEntry   endp
        end

;***************************************************************************
;***************************************************************************



/***************************************************************************
* ADDRESS.C
***************************************************************************/


-include <stdio.h>
-include <stdlib.h>

-define BYTE unsigned char
-define WORD unsigned short
-define LONG unsigned long

typedef struct
```

```
{
            BYTE inUse;
            BYTE order;
            BYTE net[4];
            BYTE node[6];
            BYTE socket[2];
            BYTE timeout[2];
            BYTE immedAddr[6];
            BYTE sequence;
            BYTE connectionNum;
            BYTE connectionStatus;
            BYTE maxTimeout[2];
            BYTE reserved[5];
}ConnIDTable;

extern int GetConnectionIDTableEntry(
            WORD connID,
            ConnIDTable *entry);

int GetServerAddress(
            WORD connID,
            BYTE *net,
            BYTE *node,
            BYTE *immediate);

/************************************************************************
* main
************************************************************************/

main()
{
            WORD c;
            BYTE net[4];
            BYTE node[6];
            BYTE immediate[6];

      for(c=1;c<9;c++)
            {

      GetServerAddress(c,net,node,immediate);

      printf(-Slot %d %.2X%.2X%.2X%.2X,c,net[0],net[1],net[2], net[3]);

      printf(-%.2X%.2X%.2X%.2X%.2X%.2X,node[0],node[1],node[2],

      node[3], node[4],node[5]);

      printf(%.2X%.2X%.2X%.2X%.2X%.2X\n,immediate[0],immediate[1],

      immediate[2],immediate[3],immediate[4],immediate[5]);
            }

      return(0);
      }

      /************************************************************************
      * GetServerAddress
```

```
    *
    * connID is an index (1..8) into the table. net points to a four-byte
    * buffer for the server's network address. node points to a 6-byte buffer
    * for the server's node address. immediate points to a 6-byte buffer for
    * the node address of the router that routes packets from the client to the
    * server. This routine returns 0 for successful and -1 if connID is an
    * invalid number.
    ************************************************************************/

int GetServerAddress(
        WORD connID,
        BYTE *net,
        BYTE *node,
        BYTE *immediate)
{
        ConnIDTable e;

    if(GetConnectionIDTableEntry(connID,&e)!=0)

    return(-1);

    memmove(net,&e.net,4);
        memmove(node,&e.node,6);
        memmove(immediate,&e.immedAddr,6);

    return(0);
    }

    /************************************************************************/
    /************************************************************************/
```

Endnotes

1. The LSL is an implementation of the Open Data-Link Interface (ODI) jointly developed by Novell and Apple.

2. These TSRs must be loaded in the following order: LSL, LAN driver, NetWare IPX, shell. If, for example, you were using an NE2000 board, you would load the following TSRs in the following order: LSL.COM, NE2000.COM, IPXODI.COM, and NET3.COM.

3. Ron Lee, a senior consultant at Novell, refers to applications that make implicit calls to networking software as networked applications. He refers to applications that make explicit calls to networking software as distributed applications.

4. Peter Norton gives a thorough and down-to-earth description of interrupts and the interrupt vector table in his book, Inside the IBM PC, Prentice Hall Press, New York, 1986.

5. For more information about the DEBUG utility, see Disk Operating System Technical Reference, International Business Machines Corporation, 1985, 1987.

6. For more information about the DOS multiplex interrupt, see Disk Operating System Technical Reference, International Business Machines Corporation, 1985, 1987.

7. For more information about NetWare IPX in workstation memory and calling the NetWare IPX ISR, see NetWare System Calls - DOS, Novell, Inc., 1989.

8. For more information about the DOS Int 21h ISR, see Disk Operating System Technical Reference, International Business Machines Corporation, 1985, 1987.

9. For more information about the shell Int 21h API, see NetWare System Calls - DOS, Novell, Inc. 1989.

10. The Novell products mentioned here include (1) NetWare C Interface for DOS which consists of all four versions of the library in all four memory models and (2) Network C for DOS which consists of Watcom development tools and standard C libraries as well as the Watcom version of the NetWare DOS library.

11. If you were to build an entire library you would want to make the assembly routines in the library as generic as possible, more generic than I present below.

12. I used Borland tools to compile and assemble these routines. To make them with the Watcom tools you need to put cdecl in front of all routines. Watcom uses some interesting "optimized" parameter passing techniques that might be worth examining in a future article.

Editor's Note: The author accepts written feedback at FAX (801) 429-5511.