

Using the NetWare 3.x Internal Debugger

Morgan B. Adair
Technical Consultant
Systems Engineering Division

NetWare v3.x includes an assembly language-oriented debugger. This AppNote uses a programming example and step-by-step instructions to illustrate how software developers can use the internal debugger when developing NetWare Loadable Modules (NLMS).

Copyright © 1991 by Novell, Inc., Provo, Utah. All rights reserved.

As a means of promoting NetWare AppNotes, Novell grants you without charge the right to reproduce, distribute, and use copies of the AppNotes, provided you do not receive any payment, commercial benefit, or other consideration for the reproduction or distribution, or change any copyright notices appearing on or in the document.

Disclaimer

Novell, Inc. makes no representations or warranties with respect to the contents or use of these Application Notes (AppNotes) or of any of the third-party products discussed in the AppNotes. Novell reserves the right to revise these AppNotes and to make changes in their content at any time, without obligation to notify any person or entity of such revisions or changes. These AppNotes do not constitute an endorsement of the third-party product or products that were tested. Configuration(s) tested or described may or may not be the only available solution. Any test is not a determination of product quality or correctness, nor does it ensure compliance with any federal, state, or local requirements. Novell does not warranty products except as stated in applicable Novell product warranties or license agreements.

Contents

The NetWare v3.x Internal Debugger	59
Debugger Basics	59
Example Program: LISTNLMS.NLM	60
Preparing for the Debugging Session	62
Starting to Debug	65
Conclusion	69
Appendix A: Internal Debugger	
Quick Reference	70
Breakpoints	71
Debugger Expressions	72

The NetWare v3.x Internal Debugger

NetWare v3.x has an internal assembly language-oriented debugger, which was used as a tool in developing the NetWare kernel and many of the NLMs and drivers that ship with NetWare 3.x. The internal debugger was left in the released code so that third-party NLM developers can also use the internal debugger as a development tool.

This AppNote presents a programming example and step-by-step instructions to illustrate some techniques for using the internal debugger when developing NLMs. Only a few of the debugger's 42 commands are used in the example debugging session included in this AppNote, but if you work along with the commands presented here, you will see how to

- break into the debugger from a running C program
- identify the point in your program's execution where it entered the debugger
- locate your program's code data in memory
- disassemble program code
- examine the contents of CPU registers and the stack
- look at the file server's screen from the debugger
- locate memory locations corresponding to program variables
- traverse a linked list that a program built dynamically
- exit the debugger and return to normal file server operation

The example program given in this AppNote was developed using the NetWare C for NLMs Software Development Kit, version 2.0. The example debugging session was executed on a NetWare v3.11 server. The example program can be developed using any version of Network C, but some of the debugging commands described in this AppNote require NetWare v3.1 or greater.

Debugger Basics

NetWare enters the internal debugger when the CPU executes an interrupt 3. There are four ways to activate the debugger:

- Press <Shift><Shift><Alt><Esc>. This method is not available if the SECURE CONSOLE command has been executed on the file server.
- Execute an INT 3 instruction in an assembly language routine.
- Call the Breakpoint function (part of the Network C library) from a C program.
- Type "386debug" after the server has abended.

In debugger commands, you must enter all numbers (memory addresses, offsets, byte, word, or instruction counts, and so on) are entered in hexadecimal. The debugger's help screens say that the "c" command to change the contents of memory can take a string parameter, but this version of the command has not yet been implemented.

Network C library functions use stack-based parameter passing. All parameters are passed as four bytes on stack, except doubles and structures or unions larger than four bytes.

NetWare v3.x uses protected mode and a flat memory model, so all

addresses are 32 bits. The debugger, therefore, does not even report the value of segment registers (applause, cheers). The mapping of virtual to physical memory addresses changed between NetWare versions 3.1 and 3.11. A simplified memory map for the three 3.x versions of NetWare released to date is given below.

Figure 1: NetWare v3.x memory map.

Example Program: LISTNLMS.NLM

Type in the following program, or download it from NetWire, CompuServe's Novell Forum A, Forum Library 16, Novell Uploads (GO NOVA).

Compile and link the program using the NetWork C for NLMs Software Developer's Kit (a make file is given after the program listing).

```

/*****
*   LISTNLMS.C
*   Illustrate linked list traversal with NetWare internal debugger
*   Builds list of NLMs in SYS:SYSTEM
*
*   Morgan Adair
*   7/11/91
*****/

#include <stdio.h>
#include <nwdir.h>
#include <string.h>
#include <malloc.h>
#include <conio.h>
#include <errno.h>

typedef struct filename {
    char    fname[NAME_MAX+1]; /* 12 + 1 bytes */
    struct filename *next;
} FILE_NAME;

void Cleanup(void);

DIR    *sysSystem;
FILE_NAME *fileList = NULL;

void main(void)
{
    DIR    *dirEntry;
    FILE_NAME *newNode;
    int    numFiles = 0;

    atexit(Cleanup);

    sysSystem = opendir("SYS:SYSTEM\\*.NLM");

    if (!sysSystem) {
        printf("Unable to open SYS:SYSTEM");
        exit();
    }

    Breakpoint(1); /* just for fun, break into the debugger here */

    do { /* while getting directory entries */
        dirEntry = readdir(sysSystem);
        if (dirEntry) {
            newNode = (FILE_NAME *)malloc(sizeof(FILE_NAME));
            if (!newNode) {
                printf("Out of memory");
                exit();
            }
            numFiles++;
            strcpy(newNode->fname, dirEntry->d_name);
            newNode->next = fileList;
            fileList = newNode;
            /* display file name, just to keep up the appearance that
               we're doing something useful (maybe if we sorted the
               file names alphabetically . . .) */
            printf("%-20s", newNode->fname);
        }
    } while (dirEntry);

    Breakpoint(2);
}

void Cleanup(void)
{
    FILE_NAME *newNode;

    closedir(sysSystem);

    while (fileList) {
        newNode = fileList;
        fileList = fileList->next;
        free(newNode);
    }
}

```

```

    }
}

```

Here is the make file I used to build LISTNLMS.

```

.BEFORE
    @set inc386=mba/sys:\nwnlms\h
    @set wcg386=mba/sys:\nwnlms\bin\386wcgl.exe

CLIBIMP = mba/sys:\nwnlms\imp\clib.imp
OBJFILE = listnlms.obj
PRELUDE = mba/sys:\nwnlms\imp\prelude.obj
NLMNAME = listnlms

.c.obj:
    @echo Compiling $[*].c
    @wcc386p /zq /d1 /3s $[*].c

$(NLMNAME).nlm : $(OBJFILE) $(NLMNAME).def
    @wlink @$ (NLMNAME).def
    @del $(NLMNAME).def

$(NLMNAME).def : makefile
    @echo FORMAT    NOVELL NLM 'List NLM files in SYS:SYSTEM' >$(NLMNAME).def
    @echo FILE      $(OBJFILE) >>$(NLMNAME).def
    @echo FILE      $(PRELUDE) >>$(NLMNAME).def
    @echo NAME      $(NLMNAME) >>$(NLMNAME).def
    @echo MODULE    clib >>$(NLMNAME).def
    @echo OPTION    VERSION=0.10 >>$(NLMNAME).def
    @echo OPTION    SCREENNAME 'listnlms' >>$(NLMNAME).def
    @echo IMPORT    @$ (CLIBIMP) >>$(NLMNAME).def
    @echo MODULE    clib.nlm >>$(NLMNAME).def

$(OBJFILE) : $(NLMNAME).c

```

Preparing for the Debugging Session

Before beginning the debugging session, disassemble LISTNLMS's object file using the Watcom disassembler:

```
wdisasm /s /l listnlms
```

The "/s" option tells the disassembler to include the C source code lines in the disassembly listing. The "/l" option causes the disassembly listing to be saved in a file.

The disassembly listing for LISTNLMS follows. When debugging your own NLMS, you will probably want to have copies of both the source code and the disassembly listing for your NLM, either printed or on a workstation screen.

```

Module: listnlms.c
Group: 'DGROUP' CONST,_DATA,_BSS

Segment: '_TEXT' BYTE USE32 000000f2 bytes

/*****
*   LISTNLMS.C
*   Illustrate linked list traversal with NetWare internal debugger
*   Builds list of NLMS in SYS:SYSTEM
*
*   Morgan Adair
*   7/11/91
*****/

#include <stdio.h>

```



```

#include <nwdir.h>
#include <string.h>
#include <malloc.h>
#include <conio.h>
#include <errno.h>

typedef struct filename {
    char    fname[NAME_MAX+1]; /* 12 + 1 bytes */
    struct filename *next;
} FILE_NAME;

void CleanUp(void);

DIR    *sysSystem;
FILE_NAME *fileList = NULL;

void main(void)
{
    DIR    *dirEntry;
    FILE_NAME *newNode;
    int    numFiles = 0;

0000  b8 14 00 00 00    main        mov     eax,00000014H
0005  e8 00 00 00 00    call    __STK
000a  53                push    ebx
000b  56                push    esi

    atexit(CleanUp);

000c  68 00 00 00 00    push    offset CleanUp
0011  e8 00 00 00 00    call    atexit
0016  83 c4 04          add     esp,0004H

    sysSystem = opendir("SYS:SYSTEM\\*.NLM");

0019  68 04 00 00 00    push    offset L7
001e  e8 00 00 00 00    call    opendir
0023  83 c4 04          add     esp,0004H
0026  a3 00 00 00 00    mov     sysSystem,eax

    if (!sysSystem) {
002b  85 c0            test    eax,eax
002d  75 12            jne     L1

        printf("Unable to open SYS:SYSTEM");
002f  68 15 00 00 00    push    offset L8
0034  e8 00 00 00 00    call    printf
0039  83 c4 04          add     esp,0004H

        exit();
    }

    /* just for fun, break into the debugger here */
003c  e8 00 00 00 00    call    exit

    Breakpoint(1);

    do { /* while getting directory entries */
0041  6a 01            L1     push    01H
0043  e8 00 00 00 00    call    Breakpoint
0048  83 c4 04          add     esp,0004H

        dirEntry = readdir(sysSystem);
004b  ff 35 00 00 00 00 L2     push    sysSystem
0051  e8 00 00 00 00    call    readdir
0056  83 c4 04          add     esp,0004H
0059  89 c6            mov     esi,eax

        if (dirEntry) {
005b  85 c0            test    eax,eax
005d  74 4b            je      L4

            newNode = (FILE_NAME *)malloc(sizeof(FILE_NAME));
005f  6a 11            push    11H
0061  e8 00 00 00 00    call    malloc

```

0066	83 c4 04	add	esp,0004H
0069	89 c3	mov	ebx,eax
	if (!newNode) {		
006b	85 c0	test	eax,eax
006d	75 12	jne	L3

```

                                printf("Out of memory");
006f 68 2f 00 00 00          push    offset L9
0074 e8 00 00 00 00          call   printf
0079 83 c4 04                add     esp,0004H

                                exit();
                                }
                                numFiles++;
007c e8 00 00 00 00          call   exit

                                strcpy(newNode->fname, dirEntry->d_name);
0081 8d 46 2c                L3      lea     eax,+2cH[esi]
0084 50                      push    eax
0085 53                      push    ebx
0086 e8 00 00 00 00          call   strcpy
008b 83 c4 08                add     esp,0008H

                                newNode->next = fileList;
008e a1 00 00 00 00          mov     eax,fileList
0093 89 43 0d                mov     +0dH[ebx],eax

                                fileList = newNode;
                                /* display file name, just to keep up the appearance that
                                   we're doing something useful (maybe if we sorted the
                                   file names alphabetically . . .) */
0096 89 1d 00 00 00 00      mov     fileList,ebx

                                printf("%-20s", newNode->fname);
                                }
009c 53                      push    ebx
009d 68 3d 00 00 00          push    offset L10
00a2 e8 00 00 00 00          call   printf
00a7 83 c4 08                add     esp,0008H

                                } while (dirEntry);

00aa 85 f6                L4      test    esi,esi
00ac 75 9d                jne     L2

                                Breakpoint(2);
00ae 6a 02                push    02H
00b0 e8 00 00 00 00          call   Breakpoint
00b5 83 c4 04                add     esp,0004H

                                }

00b8 5e                      pop     esi
00b9 5b                      pop     ebx
00ba c3                L5      ret

void Cleanup(void)
{
    FILE_NAME  *newNode;

00bb b8 08 00 00 00      Cleanup  mov     eax,00000008H
00c0 e8 00 00 00 00      call   __STK

                                closedir(sysSystem);

00c5 ff 35 00 00 00 00          push    sysSystem
00cb e8 00 00 00 00          call   closedir
00d0 83 c4 04                L6      add     esp,0004H

                                while (fileList) {
00d3 83 3d 00 00 00 00      00      cmp     dword ptr fileList,0000H
00da 74 de                je      L5

                                newNode = fileList;
00dc 8b 15 00 00 00 00          mov     edx,fileList

                                fileList = fileList->next;
00e2 8b 42 0d                mov     eax,+0dH[edx]
00e5 a3 00 00 00 00          mov     fileList,eax

```

```
        free (newNode);  
    }
```

```

00eb  e8 00 00 00 00          call    free
00f0  eb de                   jmp     L6
push    edx

```

No disassembly errors

```

-----
Segment: 'CONST' DWORD USE32 00000043 bytes
0000  00 00 00 00          _main_entry_ DD      __Null_Argv
0004  53 59 53 3a 53 59 53 54 L7      - SYS:SYST
000c  45 4d 5c 2a 2e 4e 4c 4d      - EM\*.NLM
0014  00                      - .
0015  55 6e 61 62 6c 65 20 74 L8      - Unable t
001d  6f 20 6f 70 65 6e 20 53      - o open S
0025  59 53 3a 53 59 53 54 45      - YS:SYSTE
002d  4d 00                  - M.
002f  4f 75 74 20 6f 66 20 6d L9      - Out of m
0037  65 6d 6f 72 79 00          - emory.
003d  25 2d 32 30 73 00          L10      - %-20s.

```

No disassembly errors

```

-----
Segment: '_DATA' DWORD USE32 00000004 bytes
0000  00 00 00 00          fileList - ....

```

No disassembly errors

```

-----
Segment: '_BSS' DWORD USE32 00000004 bytes

```

No disassembly errors

Starting to Debug

Load LISTNLMS on your test file server. As soon as LISTNLMS begins executing, it breaks into the internal debugger, and displays messages similar to those shown below.

```

Break at 003876C8 because of CLib Breakpoint call
EAX = 00000001 EBX = 00000001 ECX = 00000000 EDX = 0038305C
ESI = 00383228 EDI = 003833A4 EBP = 0037F1F6 ESP = 0037F134
EIP = 003876C8 FLAGS = 00007202 (IF)
00876C8 83C404      ADD     ESP,00000004

```

The sample program contained two calls to Breakpoint, each one passing a different value to the function. If we did not know which call interrupted execution of the NLM, the ".a" command displays the value passed to Breakpoint.

```

# .a
Debug entry: 1110
Break caused by: CLib Breakpoint call

Error code: 00000001

```

Since the NetWork C for NLMs library functions use stack-based parameter passing, the value that was passed to Breakpoint is also the top value on the stack. To look at the stack, use the debugger's "d" command to dump memory at the address contained in the stack pointer, ESP (0037F134 in the example).

Because the address of the memory location we want to examine is contained in a register, we can either specify the address explicitly:

```
# d 0037F134
0037F134  01 00 00 00 70 28 30 00-01 00 00 00 51 44 3A 00  ....p(8.....QD:..
```

or by the name of the register containing the address:

```
# d esp
0037F134  01 00 00 00 70 28 30 00-01 00 00 00 51 44 3A 00  ....p(8.....QD:..
```

Restart LISTNLMS by issuing a "g" command. The program is interrupted again by another breakpoint call.

```
# g
Break at 00387735 because of CLib Breakpoint call
EAX = 00000002 EBX = 00383654 ECX = 0005F158 EDX = 00000000
ESI = 00000000 EDI = 003833A4 EBP = 0037F1F6 ESP = 0037F134
EIP = 00387735 FLAGS = 00007202 (IF)
00387735 83C404          ADD     ESP,00000004
```

Notice that the value of the stack pointer has not changed since the last breakpoint. We can re-execute the command to dump the stack by pressing the up arrow key until the previous "d" command is displayed again, then pressing <Enter>.

```
# d esp
0037F134  02 00 00 00 00 C4 8B 03 00-01 00 00 00 51 44 3A 00  ....D.....QD:..
```

At this point, you may want to view the list of file names displayed on LISTNLMS's screen. You can do this by executing the debugger's "v" command. Then press any key to cycle through each of the server's screens.

At this point in LISTNLM's execution, the program has built a linked list of all NLM files in SYS:SYSTEM. To examine the linked list, we have to find where the first node in the list is stored in memory. To do so, start by finding where NetWare has loaded LISTNLMS.NLM into memory by executing the ".m" command. The debugger displays a list of all modules (server, NLM, LAN drivers, disk drivers) that have been loaded.

```
# .m
SERVER.NLM      NetWare Server Operating System
  Code Address: 00100000h  Length: 0007C620h
  Data Address: 0017C620h  Length: 00039350h
LISTNLMS.NLM    List NLM files in SYS:SYSTEM
  Version 0.10    July 11, 1991
  Code Address: 00387680h  Length: 000001ABh
  Data Address: 00387830h  Length: 00000050h
.
.
.
```

Use the debugger's "u" command to disassemble LISTNLMS, beginning at the program's code address. Pressing <Enter> continues program disassembly, 16 lines at a time. Disassemble LISTNLMS until the line shown in bold is displayed.

```

# u 00387680
00387680 B814000000 MOV EAX,00000014
00387685 E8E0CE0100 CALL CLIB.NLM|__STK
0038768A 53 PUSH EBX
0038768B 56 PUSH ESI
0038768C 683B773800 PUSH 0038773B
00387691 E836D20100 CALL CLIB.NLM|atexit
00387696 83C404 ADD ESP,00000004
00387699 6834783800 PUSH 00387834
0038769E E81A5E0100 CALL CLIB.NLM|opendir
003876A3 83C404 ADD ESP,00000004
003876A6 A378783800 MOV [00387878],EAX
003876AB 85C0 TEST EAX,EAX
003876AD 7512 JNZ 003876C1
003876AF 6845783800 PUSH 00387845
003876B4 E850A00300 CALL CLIB.NLM|printf
003876B9 83C404 ADD ESP,00000004
#
003876BC E8ADDB0100 CALL CLIB.NLM|exit
003876C1 6A01 PUSH 01
003876C3 E8D8020400 CALL CLIB.NLM|Breakpoint
003876C8 83C404 ADD ESP,00000004
003876CB FF3578783800 PUSH dword ptr [00387878]
003876D1 E878620100 CALL CLIB.NLM|readaddr
003876D6 83C404 ADD ESP,00000004
003876D9 89C6 MOV ESI,EAX
003876DB 85C0 TEST EAX,EAX
003876DD 744B JZ 0038772A
003876DF 6A11 PUSH 11
003876E1 E869A40100 CALL CLIB.NLM|malloc
003876E6 83C404 ADD ESP,00000004
003876E9 89C3 MOV EBX,EAX
003876EB 85C0 TEST EAX,EAX
003876ED 7512 JNZ 00387701
#
003876EF 685F783800 PUSH 0038785F
003876F4 E810A00300 CALL CLIB.NLM|printf
003876F9 83C404 ADD ESP,00000004
003876FC E86DDB0100 CALL CLIB.NLM|exit
00387701 8D462C LEA EAX,[ESI+2C]
00387704 50 PUSH EAX
00387705 53 PUSH EBX
00387706 E87FD50300 CALL CLIB.NLM|strcpy
0038770B 83C408 ADD ESP,00000008
0038770E A174783800 MOV EAX,[00387874]
00387713 89430D MOV [EBX+0D],EAX
00387716 891D74783800 MOV [00387874],EBX
0038771C 53 PUSH EBX
0038771D 686D783800 PUSH 0038786D
00387722 E8E29F0300 CALL CLIB.NLM|printf
00387727 83C408 ADD ESP,00000008

```

The boldface line in the disassembly listing above corresponds to the following line in the disassembly listing produced by wdisasm.

```

newNode->next = fileList;
008e al 00 00 00 00 mov eax,fileList

```

Since wdisasm does not know where NetWare will load an NLM into memory, it uses variable names to represent the memory address where the variables will be stored. The internal debugger's disassembly listing shows that fileList, the address of the first node in the linked list, is stored at address 00387874.

Because we may want to refer to the first node of the linked list more than once, we can define a symbol with a value equal to its address:

```
n fileList 00387874
```

To dump just the address of the first file name node, tell the "d" command to dump just the four bytes at 00387874.

```
# d fileList 4
00387874 54 36 38 00 T68.
```

The first node is at memory address 00383654. To begin traversing the linked list, modify the last "d" command to use the value of fileList as the memory address of the first node. The syntax of the debugger's "dl" command is

dl{+linkOffset} address {length}

where linkOffset is offset of the pointer to the next node in the linked list, and length is the number of bytes to be dumped. Nodes in LISTNLMS's linked list have the structure

```
typedef struct filename {
    char    fname[NAME_MAX+1]; /* 12 + 1 bytes */
    struct filename *next;
} FILE_NAME;
```

so the offset of the link to the next node is 13 (0Dh). The total length of the structure is 17 (11h) bytes.

```
# dl+0d [d fileList] 11
Link node 00000001
00383654 4C 49 53 54 4E 4C 4D 53-2E 4E 4C 4D 00 78 36 38
LISTNLMS.NLM.x68
00383664 00 .
```

The square brackets indicate an indirect reference through the specified address. The d in front of fileList specifies that the dword value at fileList is to be used as the address to be dumped.

You can now press <Enter> to traverse through each node of the linked list, until the debugger reaches the node with a null pointer to the next node.


```

#
Link node 00000002
00383678 54 45 53 54 2E 4E 4C 4D-00 00 00 00 FF 9C 36 38 TEST.NLM.....68
00383688 00 .
#
.
.
.
Link node 0000002A
0003C3D0 52 53 50 58 2E 4E 4C 4D-00 00 00 00 00 00 00 00 RSPX.NLM.....
0003C3E0 00 .
#
No more nodes in linked list

```

Conclusion

This AppNote touches only a few of the internal debugger's commands, but by now you should know enough to use the debugger as a tool in your NLM development process. The appendix on the following pages gives a quick reference to all internal debugger commands.

Appendix A: Internal Debugger Quick Reference

The table below summarizes NetWare v3.x internal debugger commands. Optional parameters are given in [square brackets].

Figure 2: NetWare v3.x internal debugger commands.

Command	Description
b	Display all current breakpoints.
bc <i>number</i>	Clear the breakpoint specified by <i>number</i> (0-3).
bca	Clear all breakpoints.
b = <i>address</i> [<i>expression</i>]	Set an execution breakpoint at <i>address</i> . Break will occur if EIP= <i>address</i> , and <i>expression</i> evaluates to TRUE.
br = <i>address</i> [<i>expression</i>]	Set a read/write breakpoint at <i>address</i> . Break will occur if memory at <i>address</i> is referenced, and <i>expression</i> evaluates to TRUE.
bw = <i>address</i> [<i>expression</i>]	Set a write breakpoint at <i>address</i> . Break will occur if memory at <i>address</i> is changed, and <i>expression</i> evaluates to TRUE.
c <i>address</i> [= <i>value(s)</i>]	Change memory at <i>address</i> to the specified <i>value(s)</i> . If <i>value(s)</i> are not specified, debugger prompts for new values.
d <i>address</i> [<i>count</i>]	Dump <i>count</i> bytes (default 256) at <i>address</i> .*
dl[+ <i>linkOffset</i>] <i>address</i> [<i>length</i>]	Dump memory starting at <i>address</i> for <i>length</i> bytes and traverse a linked list by following pointer <i>linkoffset</i> bytes from <i>address</i> (default <i>linkoffset</i> is 0). Press <Enter> to dump the next link node (v3.1/v3.11).
f <i>FLAG</i> = <i>value</i>	Change the <i>FLAG</i> to <i>value</i> (0 or 1), where <i>FLAG</i> is CF, AF, ZF, SF, IF, TF, PF, DF or OF.
g [<i>address(es)</i>]	Begin execution at EIP and set temporary breakpoint(s) at <i>address(es)</i> .
h	Display general help.
hb	Display breakpoint help.
he	Display expression help.
i [<i>size</i>] <i>PORT</i>	Input <i>size</i> from <i>PORT</i> , where <i>size</i> is B (byte), W (word), or D (dword) (default is byte).
m <i>address</i> [L <i>length</i>] <i>byte(s)</i>	Search memory beginning at <i>address</i> for <i>byte(s)</i> (if <i>length</i> is not specified, the rest of memory will be searched). Byte values must be given in hexadecimal and separated by spaces or commas.*
n [<i>symbolName</i> <i>value</i>]	Define new symbol <i>symbolName</i> equal to <i>value</i> . If <i>symbolName</i> and <i>value</i> are not specified, all symbols and values are displayed.
n - <i>symbolName</i>	Remove user-defined symbol <i>symbolName</i> (n-- remove all symbols) (v3.1/v3.11).
o [<i>size</i>] <i>PORT</i> = <i>value</i>	Output <i>size</i> <i>value</i> to <i>PORT</i> , where <i>size</i> is B (byte), W (word), or D (dword) (default is byte).
p	Single step, proceed over CALLs, REPs, and LOOPs.*
q	Quit and exit back to DOS (or reboot if DOS has been removed).

r	Display registers and flags (v3.1/v3.11).
REG=value	Change the specified register to <i>value</i> , where <i>REG</i> is EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP or EIP.
s	Single step, trace into CALLs, REPs or LOOPs.*
t	Same as s.*
u address [count]	Unassemble <i>count</i> instructions starting at <i>address</i> .*
v	View file server screens (press any key to go to next screen).
z expression	Evaluate <i>expression</i> .
? [address]	If symbolic information has been loaded, display the closest symbols to <i>address</i> (default is EIP).
.a	Display the abend or break reason (v3.1/v3.11).
.c	Dump diagnostic data to diskette.
.h	Display dot command help.
.m	Display names and addresses of all loaded modules.
.p [address]	Display <i>address</i> as a process control block (if <i>address</i> is not specified, display all process names and addresses).
.r	Display the process control block for the running process.
.s [address]	Display <i>address</i> as a screen structure (if <i>address</i> is not specified, display all screen names and addresses).
.v	Display server version.

*The d, m, p, s, t, and u commands can be continued or repeated by pressing <Enter> at the # prompt.

Breakpoints

A breakpoint condition can be any expression. If a breakpoint condition is specified, the condition is evaluated when the break occurs. If the condition is not true, execution resumes without entering the debugger.

Figure 3: NetWare v3.x internal debugger expression operators (continued).

There are four breakpoint registers, allowing a maximum of four breakpoints to be set at a time. These breakpoints can be permanent breakpoints (set using the "b" command) or temporary breakpoints (set using the "g" command). In addition, the "p" command also sets a temporary breakpoint if the current instruction can not be single-stepped (a CALL, or one of the REP or LOOP families). Because of the limited number of breakpoint registers, you might not be able to execute a "g" or "p" command when four permanent breakpoints are set.

Here are some examples of commands that use breakpoints:

```
b = PushNode [d esp+4] == 0
    Set a breakpoint at PushNode (which you must have previously
    declared with the n command). Break if the first parameter on
    the stack is NULL.
```

```
bw = 16fe60 eip >= listnlms && eip <= (listnlms+lab)
    Break if any instruction in LISTNLMS tries to modify memory
    address 16fe60.
```

```
g [d esp]
    Execute until the current function returns to its caller.
```

Debugger Expressions

Expressions consist of terms and operators. Terms in expressions can be hexadecimal constants, symbols, or register or flag names. You can use grouping operators to cause terms to be interpreted as indirect memory addresses or hardware port addresses. The following are register and flag names as they are used in debugger expressions.

Registers: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP and EIP
 Flags: FLCF, FLAF, FLZF, FLSF, FLIF, FLTF, FLPF, FLDF and FLOF

The table below gives debugger expression operators in order of precedence.

Figure 3: NetWare v3.x internal debugger expression operators.

Symbol	Description	Precedence
Grouping operators		
(<i>expression</i>)	Evaluate <i>expression</i> at higher precedence.	0
[<i>size expression</i>]	Read <i>size</i> from the memory address specified by <i>expression</i> , where <i>size</i> is B (byte), W (word), or D (dword).	0
{ <i>size expression</i> }	Evaluate <i>expression</i> and resulting value as a port address. The bracketed expression is replaced with the byte, word, or double word input from the port.	0
Unary operators		
!	logical NOT	1
-	2's complement	1

Figure 3: NetWare v3.x internal debugger expression operators (continued).

~	1's complement	1
Binary operators		
*	multiply	2
/	divide	2
%	mod	2
+	add	3
-	subtract	3
>>	bit shift right	4
<<	bit shift left	4
>	greater than	5
<	less than	5
>=	greater than or equal to	5
<=	less than or equal to	5
!=	not equal to	6
==	equal to	6
&	bitwise AND	7
^	bitwise XOR	8
	bitwise OR	9
&&	logical AND	10
	logical OR	11
Ternary operator		
expression1 ? expression2 , expression3	If expression1 is true then the result is the value of expression2, otherwise the result is the value of expression3.	12

***Figure 3: NetWare v3.x internal debugger expression operators
(continued).***