Appendix A: Creating Loadable Modules

Device drivers must be converted from source modules into NetWare Loadable Modules before they can be loaded and installed in the NetWare v3.1x or v4.0 Operating Systems. Appendix D provides a list of compilers and assemblers which may be used to create object modules from driver source modules. The object modules must be processed by a NetWare linker, either NLMLINKP (which makes use of extended memory) or NLMLINKR (which uses only regular memory.) Both linkers require a developer-created definition file (.DEF) that provides configuration information needed to produce the NLM, including the NetWare internal variables and routines the driver will access once loaded.

The NLMLINK(P)(X)(R) command syntax is as follows:

nlmlinkp drivername<Enter>

Where: drivername is the name of the definition file for the desired driver (It is not necessary to enter the .DEF extension in this command).

NLMLINK(P)(X)(R) will find all the required information and directives in the definition file, including the names of all object modules which must be linked to form the driver (see figure A-1 for a sample definition file). NLMLINKP(R) will produce an output file (with a .DSK extension) which is the NLM form of the disk driver, ready to load into an active NetWare environment. Figure A-1 shows a sample definition file which will direct NLMLINKP(R) to find sample.obj, link it, append a file named firmware.com to it, and produce the output file sample.dsk.

.Def File

```
type 2
description "NetWare Sample Disk Driver"
output sample
input  sample
start  InitSampleDriver                    /* public routine in NLM */
exit   RemoveSampleDriver                  /* public routine in NLM */
check  CheckSampleDriver                /* public routine in NLM */
reentrant                               /* allow re-entrant use of driver*/
version  1,0,a                             /* version number */
custom firmware.com                     /* co-processor firmware file */
map                                     /* produces map */
copyright "Sample Copyright stuff"
debug                                   /*  allows  debugger  symbols   -   remove  in  production
                                        version */
import                                  /* externals */
    AlertDevice
    AllocateResourceTag
    Alloc
    Free
    GetHardwareBusType
    GetSectorsPerCacheBuffer
    ParseDriverParameters
    RegisterHardwareOptions
    DeRegisterHardwareOptions
    SetHardwareInterrupt
    ClearHardwareInterrupt
    ScheduleNoSleepAESProcessEvent
    CancelNoSleepAESProcessEvent
    CRescheduleLast
    DelayMyself
    OutputToScreen
    AddDiskSystem
    DeleteDiskSystem
    CheckDiskCard
    AddDiskDevice
    RemoveDiskDevice
    DeleteDiskDevice
    GetIOCTL
    PutIOCTL
    GetRequest
    PutRequest
```

Figure A-1 Sample Definition file

Definition File Keywords

The .DEF file keywords can occur in any order, and the required keywords are indicated below. The following keywords are defined for use in the definition file to direct NLMLINKP(R) in creating NetWare v3.1x & v4.xx Loadable Module device drivers:

TYPE (Required) Specifies the type of loadable module as indicated below, and implicitly determines the extension to append on the output file.

1 = Lan Driver (.LAN)
2 = Disk Driver (.DSK)
3 = Name Space Module (.NAM)
4 = Utility (.NLM)
8 = Host Adapter Module (.HAM)
9 = Custom Device Module (.CDM)

DESCRIPTION (Required) Specifies a string that describes the loadable module to be created. The console command "MODULES" will output this string to describe this module. The description can be 1-127 bytes long, must be enclosed in double quotes and may not include a null, double quote, carriage return, or new-line. The description should contain the indicated fields in the following order format:

"company or product name, description"

OUTPUT (Required) Specifies the name of the output file (the extension will be added by the linker as specified above).

INPUT (Required) Specifies the name of the input .OBJ file(s).

START (Required) Specifies the name of the loadable module's initialization procedure. When the file server supervisor uses the "LOAD" console command to load the module, NetWare calls this procedure.

EXIT (Required) Specifies the name of the loadable module's exit procedure. This procedure is called when the file server supervisor enters the "UNLOAD" console command.

CHECK (Required) Specifies the name of the loadable module's check procedure. The console command "UNLOAD" calls an NLM's check procedure (if it exists) before unloading the module. The check procedure is <u>required</u> for disk drivers since it must indicate to the OS if any disks are locked (module may not be safely unloaded).

COPYRIGHT (Required) Inserts the Novell default copyright or the copyright for a third-party developer.
Usage: COPYRIGHT Novell default.
COPYRIGHT "company" Third party copyright message.

VERSION (Required) Usage: VERSION XX,YY,ZZ

XX - Major Version number,
YY - Minor Version number,
ZZ - Revision number (The zero-based ordinal number of the alphabetic letter to be displayed.)

REENTRANT (Optional) Specifies that the loadable module is reentrant (i.e., two or more processes may use the code at the same time). This keyword is mutually exclusive (both cannot be specified) with the keyword **MULTIPLE**.

MULTIPLE (Optional) Specifies that more than one code image of the loadable module may be loaded into file server memory. This keyword is mutually exclusive (both cannot be specified) with the keyword **REENTRANT**.

CUSTOM (Optional) Specifies that a custom data file is to be appended to the output file. This keyword should be followed by the file name of the custom data file.

MAP (Optional) Directs the linker to create a map file.

IMPORT (Required) Specifies that a list following the keyword will contain variable and procedure names that are external to the object files. These are case sensitive NetWare v3.1x - v4.xx Operating System variables and procedures (or variables and procedures from other loadable modules which must have been previously loaded) which will be linked to the module after it has been loaded and before it begins initialization.

EXPORT (Optional) Specifies that a list following the keyword will contains case sensitive variables and procedure names resident in the loadable module to be made available to other loadable modules.

MODULE (Optional) Specifies loadable modules that must be loaded before the current loadable module is loaded. The loader will attempt to find and load any modules not already in the server memory. If it cannot, the current module will not be loaded.

DEBUG (Optional) Specifies that the linker will include debug information in the output file.

@ operator (Optional) "@" is an operator that can be used with the Input, Import, and Export directives. The @ operator indicates that the list is to be read from a file, and can be nested. The file specifier, including path, must immediately follow the @ operator.

Syntax: IMPORT @file.txt

Appendix B: The NetWare Debugger

Debugger Features

The NetWare Operating System includes an internal (built-in) assembly language-oriented debugger. The following describes some of its feature including:

- Trap into the debugger from an assembler or C program
- Trap into the debugger from the server console keyboard either dynamically or following a server ABEND or GPI
- Identify the cause and point of a program's execution where it trapped into the debugger
- Single step
- Proceed to next instruction
- Go until a specified address is reached
- Set and/or clear breakpoints (including read or write breakpoints with conditional statements.) a maximum of 4 breakpoints can be set at one time
- Un-assemble code
- Display and/or modify registers
- Display and/or modify memory, including the stack
- Read from and write to ports
- Search memory for a byte pattern
- Traverse a linked list that has been built dynamically
- Display modules currently loaded
- Display current server processes
- Display process control blocks
- Display screens, including the file server's screen
- Display debugger help screens
- Exit the debugger and either return to normal file server operation or to DOS

Futher explanation of extended features in the debugger may be found in the on-line help information that may be accessed within the debugger as described below.

Debugger Basics

Note: The debugger displays a pound-sign ("#") for its prompt, as shown in the examples in this section.

Entering The Debugger

Start the debugger in one of the following ways:

From the server console keyboard:

1) Depress <R-Alt>, <R-Shift>, <L-Shift>, and <Esc> simultaneously at the server console keyboard. (This will not work if the server is hung in an infinite loop with interrupts disabled, or if the server console is secured.)

2) v4.xx only - If a driver is loaded using a -D option NetWare will break into the debugger at the beginning of the InitializeDriver code.

   EXAMPLE: load -D DRIVERNAME <CR>

3) After the server abends or GPIs, enter either:

   <Alt>, <R-Shift>, <L-Shift>, and <Esc>
   or
   "**386debug**" on the server console keyboard.

From a driver or NLM:

1) Include the assembly command "INT 3" in the part of the driver code where a break is desired. (Drivers written in C can call the "**EnterDebugger();**" function.)

Manually:

2) Generate a Non-Maskable Interrupt with an NMI board. This will cause the server to Abend, (after which step #2 above may be performed). This may be required if the software being debugged is in an infinite loop with interrupts disabled.

When you start the debugger, it will display the location at which the trap occurred, the cause of the trap into the debugger, and the contents of general registers and the flags.

Debugger Commands

The modules currently loaded and their code and data segment addresses can be displayed by entering

.m

Help Commands

To display a help screen, you may enter any of the following commands:

h	displays help for general commands
hb	displays help for breakpoints
he	displays help for grouping operators
.h	displays help for .COMMANDS

h displays help for general commands (see example below)

```
# h
B                       Breakpoint commands (see HB help screen)
C address               Change memory in interactive mode
C address=number(s)         Change memory to the specified number(s)
C address="text"        Change memory to the specified text ASCII values
D address {length}          Dump memory for optional length
DL{+linkOffset} address {length}
                        Dump memory starting at address for optional length and
                        traverse a linked list (default link field offset is 0)
                        Use <ENTER> to dump the next link node
REG=value               Change the specified register to the new value
                        REG is EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP, or EFL
F Flag=value            Change the FLAG bit to value (0 or 1)
                        where FLAG is CF, AF, ZF, SF, IF, TF, PF, DF or OF
G {break address(s)}    Begin execution at current EIP and set optional temporary
                        breakpoints(s)
H, HB, HE, .H           Display help screens
I {B;W:D} PORT              Input byte, word, or dword from Port (default is byte)
M start {L length} pattern-byte(s)
                        Search memory for pattern (L length is optional and if not
                        specified, the rest of memory will be searched)
N symbolName address    Define a new symbol name at address
N -symbolName           Remove defined symbol name (n-- remove all symbols)
O {B;W;D} PORT=value    Output byte, word, or dword value to PORT
<Press ESC to terminate or any other key to continue>
```

Console screen after entering "**h**" for general command help

```
                        REG is EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP, or EFL
F Flag=value            Change the FLAG bit to value (0 or 1)
                        where FLAG is CF, AF, ZF, SF, IF, TF, PF, DF or OF
G {break address(s)}    Begin execution at current EIP and set optional temporary
                        breakpoints(s)
H, HB, HE, .H           Display help screens
I {B;W:D} PORT              Input byte, word, or dword from Port (default is byte)
M start {L length} pattern-byte(s)
                        Search memory for pattern (L length is optional and if not
                        specified, the rest of memory will be searched)
N symbolName address    Define a new symbol name at address
N -symbolName           Remove defined symbol name (n-- remove all symbols)
O {B;W;D} PORT=value    Output byte, word, or dword value to PORT
P                       Proceed over the next instruction
Q                       Quit and exit back to DOS
R                       Display registers and flags
T or S                  Single step
U address {count}           Unassemble count instructions starting at address
V                       View server screens
Z expression            Evaluates the expression (See HE help screen)
? {address}                 If symbolic information has been loaded, the closest
                        symbols to address (default is EIP) are displayed

Use <ENTER> to continue or repeat the d, dl, m, p, s, t, and u commands
#
```

And after pressing a key other than ESC

hb display help for breakpoint commands (see example below)

```
# hb
B                        display all current breakpoints
BC number                clear the specified breakpoint
BCA                          clear all breakpoints
B  = address {condition}     set an execution breakpoint at address
BW = address {condition}     set a write breakpoint at address
BR = address {condition}     set a read/write breakpoint at address

If a breakpoint condition is specified, the condition will be evaluated
when the break occurs.  If the condition is not true then execution will
be resumed immediately without entering the interactive debugger mode.

A breakpoint condition can be any expression.  For a description of possible
expressions see the HE help screen.

There are 4 breakpoint registers, allowing a maximum of 4 breakpoints to be
set at the same 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 will also set a temporary breakpoint if the
current instruction can not be single stepped.




#
```

Console screen after entering "**hb**" for breakpoint command help

he displays help for grouping operators in expressions (see example below)

```
# he
Grouping operators
  These operators (), [] and {} have precedence 0.  The grouping operators
  can be nested in any combination.  Note that "size is a data size
  specifier B, W, or D.

 (expression)         causes expression to be evaluated at a higher
                      precedence.

 [size expression]       causes expression to be evaluated at a higher
                         precedence
                      and then uses expression as a memory address.  The
                      bracketed expression is replaced with the byte, word
                      or
                      double word at that address.

 {size expression}       causes expression to be evaluated at a higher
                         precedence
                      and then uses expression as a port address.  The
                      bracketed
                      expression is replaced with the byte, word or double
                      word
                      at that address.

Unary operators
  Symbol   Description       Precedence
  !           logical not          1
  -           2's complement       1
  ~           1's complement       1

<Press ESC to terminate or any other key to continue>
```

Console screen after entering "**he**" expression help command

```
Binary operators
  *  multiply         2   >  greater-than       5   != not-equal-to        6
  /  divide               2   <  less-than       5   &  bitwise AND         7
  %  mod              2   >= greater-than           ^  bitwise XOR         8
  +  add              3      or equal-to         5   |  bitwise OR      9
  -  subtract         3   <= less-than or           && logical AND        10
  >> bit shift right  4      equal-to            5   || logical OR     11
  << bit shift left       4   == equal-to            6

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.

All numbers are entered and shown in hex format.  In addition to numbers,
register, flag, and symbol values can be used.

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

#
```

Console screen after pressing **Enter**

.h displays help for dot commands (see example below)

```
# .h
Dot commands
    .A                Display the abend or break reason
    .C                Do a diagnostic memory dump to diskette
    .H                Display the dot help screen
    .M                Display loaded module names and addresses
    .P                Display all process names and addresses
    .P <address>      Display <address> as a process control block
    .R                Display the running process control block
    .S                Display all screen names and addresses
    .S <address>      Display <address> as a screen structure
    .V                Display server version
#
```

Console screen after entering ".h" for dot command help

Breakpoints

B Display all breakpoints that are currently set.

```
# b

Breakpoint 0 write byte at FFF65623
Breakpoint 1 read or write byte at 000653BA
Breakpoint 2 execute at FFF06BA3
```

BC <number> Clear the breakpoint specified by *<number>*.

```
# bc 2

Breakpoint cleared
```

BCA Clear all breakpoints.

```
# bca

All breakpoints cleared
```

B = <address> {condition} Set an execution breakpoint at the address specified when the indicated condition is true.

```
# b = fff8765a

Set as breakpoint 0
```

BW = <address> {condition} Set a write breakpoint at the address specified when the indicated condition is true.

```
# bw = fff665ab

Set as breakpoint 1
```

BR = <address> {condition} Set a read / write breakpoint at the address specified when the indicated condition is true. (see conditions below).

```
# br = 0065acf3

Set as breakpoint 2
```

If a breakpoint condition is specified, the condition will be evaluated when the break occurs. If the condition is not true, then execution will be resumed immediately without entering the interactive debugger. A breakpoint condition can be any expression. There are four breakpoint registers, allowing a maximum of four breakpoints to be set at the same time. These breakpoints can be permanent breakpoints using the B commands above, or temporary breakpoints set using the G command (described below). In addition, the P command will also set a temporary breakpoint if the current instruction cannot be single stepped.

Displaying Memory

D <address> {count} Displays the contents of memory, starting at *<address>*, for <count> number of bytes.

```
# d fff7765e

FFF7765E  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7766E  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7767e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7768e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7769e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF776Ae  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF776Be  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF776Ce  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF776Ee  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF776Fe  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7770e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7771e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7772e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7773e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7774e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
FFF7775e  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................

# d fff7765e 3

FFF7765E  00 00 00
```

Register Manipulations

R Display the registers EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, and EIP registers. It also displays all flags.

```
# R

EAX=99999999  EBX=00005455 ECX=78787878 EDX=00060544
ESI=00000000  EDI=80868086 EBP=00000000 ESP=FFF67876
EIP=FFF56784  FLAGS=00010002
```

<REG> = <value> Change the specified register to the new value. The command is effective with the EAX, EBX, EXC, EDX, ESI, EDI, ESP, EBP, and EIP registers.

```
# eax=8099acb3

Register changed
```

Changing Memory

C <address> Interactively change the contents of memory location *<address>*.

```
# c fff6432a

FFF6432A (00)=33  (34)=98  (5A)=.
```

Note: To end interactive mode type a period.

C <address> = <number or numbers> Change the memory value or values, beginning at *<address>*, to the specified number or numbers.

```
# c fff534c5 = 00,00,12,5a,78

Change successfully completed
```

C <address> = "text" Place the text string specified beginning at <address>.

```
# c fff60db3 = "This is a string."

Change successfully completed
```

F <FLAG> = <value> Change the specified flag to the new value (0 or 1). The command is effective with the CF, AF, ZF, SF, IF, TF, PF, DF, and OF flags.

```
# f pf =0

Flag changed
```

I/O

I {B,W,D} <PORT> Input a byte, word, or double from port.

```
# i 255

Port (255)=ff
```

O {B,W,D} <PORT> = <VALUE> Output a byte, word, or double value to port.

```
# o 255 = 78

Output completed
```

Miscellaneous

G {break address(es)} Begin execution at current EIP and set temporary breakpoint(s) to address(es).

```
# g fff56784

Break at FFF56784 because of go breakpoint
EAX=99999999  EBX=00005455  EXC=78787878  EDX=00060544
ESI=00000000  EDI=80868086  EBP=00000000  ESP=FFF67876
EIP=FFF56784  FLAGS=00010002FFF56784 BB30CE0500     MOV     EBX,0005CE30
```

H, HB, HE, .H Display help screens.

M <START> {L len} <BYTE(S)> Search memory for pattern (the byte pattern) from start until length is reached.

```
# m fff77e50 48 61 72 64

FFF77Ef0  54 48 45 52 4E 45 54 5F-49 49 00 90 00 00 00 00  THERNET_II......
FFF77F00  00 00 00 00 00 00 90 6B-F7 FF 00 00 00 00 00 00  .......kw........
FFF77F10  48 61 72 64 77 61 72 65-44 72 69 76 65 72 4D 4C  HardwareDriverML
```

N <symbolname> <VALUE> Define a new symbol with a value.

```
# n thissym 0f0f
```

P Proceed over next instruction.

Q Quit and return to DOS.

T or S Trace or single step through the program.

U <address> {COUNT} Unassemble count instructions from address.

```
# u FFF87885 2

FFF87885  0000      ADD      [EAX],ALFFF87887  0000      ADD      [EAX],AL
```

V View the screens (will step through the screens sequentially).

Z expression Evaluate the expression.

```
# z 7+8

Evaluates to:  F
```

The D, M, P, S, T, and U commands can be continued or repeated by simply pressing a carriage return at the # prompt.

Dot Commands

.A	Display the abend or break reason
.C	Do a diagnostic memory dump to diskette
.H	Display the dot help screen
.M	Display loaded module names and addresses
.P	Display all process names and addresses
.P <address>	Display <address> as a process control block
.R	Display the running process control block
.S	Display all screen names and addresses
.S <address>	Display <address> as a screen structure
.V	Display server version

Expressions

Grouping Operators — These operators **()**, **[]** and **{}** have a precedence of 0. The grouping operators can be nested in any combination. Note that "size" is a data size specifier of the type B, W, or D.

(expression) — Causes expression to be evaluated at a higher precedence.

[size expression] — Causes expression to be evaluated at a higher precedence and then uses expression as a memory address. The bracketed expression is replaced with the byte, word, or double word at that address.

{size expression} — Causes expression to be evaluated at a higher precedence and then uses expression as a port address. The bracketed expression is replaced with the byte, word, or double word input from the port.

Unary Operators

Symbol	Description	Precedence
!	logical not	1
-	2's compliment	1
~	1's compliment	1
*	multiply	2
/	divide	2
%	mod	2
+	addition	3
-	subtraction	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
==	equal to	6
!=	not 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 expression.

The beginning and length of data and code segments may be found by entering ".m" at the debug prompt. Breakpoints can then be set in the driver code using the addresses in the map file relative to the addresses dumped by the debugger.

Symbolic information may be included in a driver's .DSK file which can be used to access routines or variable by name while in the NetWare v3.xx/v4.xx debugger (the debugger is case sensitive). To access symbolic information, the following steps must be taken:

1) Declare public all desired symbols in the driver

2) Include the keyword "debug" in the driver definition file.

Each of these symbols can now be used in the same way the address they represent would be used. For example, at the debug prompt it is possible to display memory beginning at the address of the label AdapterBdStruct by entering:

#d AdapterBdStruct

Symbols may be dynamically defined by the debugger. If it is necessary to dynamically define more than 10 symbols the server must be loaded with the "-y" option.

Note: Debugging information must be removed before releasing the driver. Including the "debug" keyword in the definition file will cause a message to be displayed on the console when the driver is loaded, indicating that it contains debug information.

Debugger Command Summary

.A	*Display the abend or break reason*
B	*display all current breakpoints*
BC number	*clear the specified breakpoint*
BCA	*clear all breakpoints*
B = address {condition}	*set an execution breakpoint at address*
BW = address {condition}	*set a write breakpoint at address*
BR = address {condition}	*set a read/write breakpoint at address*
C address	*Change memory in interactive mode*
C address=number(s)	*Change memory to the specified number(s)*
C address="text"	*Change memory to the specified text ASCII values*
.C	*Do a diagnostic memory dump to diskette*
D address {length}	*Dump memory for optional length*
DL{+linkOffset}	*address {length}*
	Dump memory starting at address for optional length and traverse a linked list (default link field offset is 0) Use <ENTER> to dump the next link node
REG=value	*Change the specified register to the new value*
	REG is EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP, or EFL
F Flag=value	*Change the FLAG bit to value (0 or 1)*
	where FLAG is CF, AF, ZF, SF, IF, TF, PF, DF or OF
G {break address(s)}	*Begin execution at current EIP and set optional temporary breakpoints(s)*
H	*Display basic debugger command help screen*
HB	*Display breakpoint help screen*
HE	*Display expression help screen*
.H	*Display the dot help screen*
I {B;W:D} PORT	*Input byte, word, or dword from Port (default is byte)*
M start {L length} pattern-byte(s)	
	Search memory for pattern (L length is optional and if not specified, the rest of memory will be searched)
.M	*Display loaded module names and addresses*
N symbolName address	*Define a new symbol name at address*
N -symbolName	*Remove defined symbol name*
N--	*Remove all defined symbols*
O {B;W;D} PORT=value	*Output byte, word, or dword value to PORT*
P	*Proceed over the next instruction*
.P	*Display all process names and addresses*
.P <address>	*Display <address> as a process control block*
Q	*Quit and exit back to DOS*
R	*Display registers and flags*
.R	*Display the running process control block*
S	*Single step*
.S	*Display all screen names and addresses*
.S <address>	*Display <address> as a screen structure*
T	*Single step*
U address {count}	*Unassemble count instructions starting at address*
V	*View server screens*
.V	*Display server version*
Z expression	*Evaluates the expression (See HE help screen)*
? {address}	*If symbolic information has been loaded, the closest symbols to address (default is EIP) are displayed*

Use <ENTER> to continue or repeat the d, dl, m, p, s, t, and u commands

Debugger Limitations

The Internal Debugger currently has the following limitations:

1) Modifying memory on Adapters with Shared Ram (S-RAM) requiring word (16-bit) memory operations (typical with ISA adapters designed for higher performance S-RAM access) will cause garbage data to be written to every other byte. Displaying SRAM will function correctly (valid data displayed), since the high-byte data supplied by the adapter will be ignored by the CPU.

2) Attempting to single-step an instruction which cannot be single-stepped (a breakpoint is normally set by the debugger following the instruction) will fail when 4 breakpoints are already set.

3) The debugger will not allow disassembly of instructions within 20 bytes of the top of defined memory (The GDT will be set to include all defined memory including adapter SRAM). Memory may be displayed to the end of defined memory. Although it is possible to have non-existent memory within the area of defined memory, the debugger will encounter difficulty (the CPU will hang) displaying non-existent memory on machines where a memory time-out which generates a memory ready is not implemented.

Appendix C: Subroutine Calling Conventions

The NetWare v3.1x/v4.xx Operating System support routines provided for drivers, and the driver routines required to be exported to the NetWare OS must follow C subroutine calling conventions. In addition, the driver must have interrupts disabled where required prior to calling NetWare support routines (see Chapter 7). Interrupts will also be disabled upon entry to the driver non-blocking process level entry points (excepting AESProcess event entry points), and to driver Interrupt Service Routines (ISRs). The following sections detail the conventions to which drivers must adhere.

Calling C Support Routines

NetWare requires that calls to and from the operating system use standard C conventions. All parameters are passed on the stack. Drivers are not required to save registers EBP, EBX, ESI, and EDI when making system support routine calls, as these registers are saved and restored by the routine called; however, the state of the other registers are not guaranteed and could be altered by the called routine. All calls are NEAR, due to the flat memory model used by the NetWare 3.1x/v4.xx OS. When exiting a driver routine, the interrupt flag must have the same value as when the routine was called by NetWare. To call a NetWare routine, push all variables on the stack, call the routine, and then adjust the stack pointer upon return. For example, a driver's Initialization routine must call the NetWare support routine AddDiskSystem, which appears to the driver as follows:

```
CardStruct *AddDiskSystem(
        LONG NLMHandle,
        IOConfigStruct *IOStruct,
        LONG (*IOCTLPollRoutine)(),
        LONG (*ScanForDevices)(),
        LONG Reserved0,
        LONG NovellNumber,
        LONG DriverResourceTag,
        LONG CardStructureSize);
```

Figure C-1 C Syntax for AddDiskSystem

As you can see, AddDiskSystem requires eight arguments and returns one value. To call AddDiskSystem, the driver's initialization routine must use code similar to the following:

```
push       CardStructureSize               ;size of CardStructure
push       DriverResourceTag               ;resource tag
push       NovellNumber                 ;Novell assigned number
push       0                            ;Reserved0
push       OFFSET ScanForDevices           ;scan/add routine
push       OFFSET IOCTLPollRoutine         ;IOCTL Poll routine
movzx      eax, CurrentCardNumber
mov           eax, IOConfigList[eax*4]        ;get IOConfig structure
push       eax
push       [esp + Parm0]                ;driver handle
call       AddDiskSystem
lea        esp, [esp + 8*4]             ;adjust stack ptr
```

Figure C-2 Calling a C Routine

The 80386 PUSH instruction pushes DWORD(s) on the stack. Note that the values are pushed in reverse order. Note also that after the initialization routine calls AddDiskSystem, the initialization routine adjusts the stack pointer by 8 * 4. Eight push instructions times four bytes (each push is one dword). Note also that the LEA instruction may not be the only way to adjust the ESP, but it's quick and easy.

Drivers Called by a C Routine

Driver routines called by NetWare are called using "C" subroutine calling conventions. Any parameters passed to the driver routines are pushed on the stack in "C" compatible reverse order. The driver routines called must save registers EBP, EBX, ESI, and EDI on the stack upon entry, and must restore them just prior to returning to the caller. Interrupts may or may not be disabled, depending on the driver routine called. Driver routines that require use of passed parameters must retrieve them from the stack. For example, a NetWare call to a driver's initialization routine could look much like the syntax shown in figure C-3.

Some NetWare support routines require the driver to pass values which are provided by NetWare with the driver initialization call. The required values were pushed on the stack as C-style parameters before calling the Driver initialization procedure, and should be saved by the initialization procedure for later driver reference.

```
LONG InitializeDriver(
        LONG NLMHandle,
        LONG ScreenHandle,
        BYTE *LoadCommandLine,
        LONG Reserved0,
        LONG Reserved1,
        LONG CustomDataFileHandle,
        LONG (*ReadCustomDataRoutine)(
                LONG CustomDataFileHandle, LONG CustomDataOffset,
                BYTE *CustomDataDestination, LONG CustomDataSize),
        LONG CustomDataOffset,
        LONG CustomDataSize);
```

Figure C-3 C Syntax for an Initialization Routine

```
LONG ParseDriverParameters(
        IOConfigStruct *IOConfig,
        LONG Reserved0,
        AdapterOptionStruct *Options,
        LONG Reserved1,
        LONG Reserved2,
        LONG NeedBitMap,
        BYTE *CommandLine,
        LONG ScreenHandle);
```

Figure C-4 Syntax for the ParseDriverParameters call

ParseDriverParameters requires pointers to the command line, an I/O Configuration structure, a bit map, an Adapter Options structure, and a ScreenHandle. To pass command line pointer and screen handle, the driver's initialization routine must have retrieved them from the stack (where NetWare put them).

Given the syntax shown in Figure C-4, a typical initialization routine could call ParseDriverParameters as shown in Figure C-5:

```
push      [esp + Parm1]                        ;screen handle from stack
push      [esp + Parm2]                        ;cmd line pointer
push      NeedsIOPort0Bit + NeedsInterrupt0Bit      ;config requirements
push      0                                    ;Reserved2
push      0                                    ;Reserved1
push      OFFSET Options                           ;Options      structure
                                                   pointer
push      0                                    ;Reserved0
push      OFFSET IOConfig                          ;I/O  Config  structure
                                                   ptr
call      ParseDriverParameters
lea       esp, [esp + (8*4)]                           ;adjust stack ptr
```

Figure C-5 Calling ParseDriverParameters

In Figure C-5 and in all code examples throughout this book, Parm0, Parm1, ..., Parm11 are defined as shown below:

```
ParmOffset    equ    20
Parm0     equ    ParmOffset + 0
Parm1     equ    ParmOffset + 4
Parm2     equ    ParmOffset + 8
Parm3     equ    ParmOffset + 12
Parm4     equ    ParmOffset + 16
Parm5     equ    ParmOffset + 20
Parm6     equ    ParmOffset + 24
Parm7     equ    ParmOffset + 28
Parm8     equ    ParmOffset + 32
Parm9     equ    ParmOffset + 36
Parm10    equ    ParmOffset + 40
Parm11    equ    ParmOffset + 44
```

Figure C-6 Stack Parameter Definitions

In Figure C-6, ParmOffset is defined as 20 in order to represent the 20 bytes that are normally pushed on the stack when a C-style call is made (4 by the call instruction, and 16 by the called routine, upon entry, when saving EBX, EBP, ESI, and EDI). Defining the stack offsets this way is one method that can simplify the retrieval of parameters off the stack. However, the driver can use any method preferred.

Appendix D: Development Tools

NetWare driver developers must use compilers or assemblers which produce native 32-bit code and object modules compatible with the Phar Lap Easy OMF-386, which is the 8086-OMF extension defined by Phar Lap to support the 32-bit addressing modes of the 80386. The NetWare v3.xx Loadable Module Linker (NLMLINKP) must be used to link the modules and requires the above object module format. Compilers and Assemblers currently available for use are:

Assemblers:

The 386|ASM v2.0 (or later) protected mode assembler by Phar Lap Software, Inc.

Compilers:

Novell/WATCOM C Network Compiler/386 v1.0 (or later), available from Novell, Inc.

High C compiler v1.4 (or later) by MetaWare, Incorporated.

Linkers:

NLMLINK, available with NetWare v3.1 - v3.11 from Novell, Inc.

NLMLINKX, available with NetWare v3.1 - v4.xx from Novell, Inc.

NLMLINKR, available with NetWare v3.11 - v4.xx from Novell, Inc.

Novell/WATCOM WLINK, available from Novell, Inc.

Debuggers:

The NetWare debugger (integrated with NetWare)

Novell/WATCOM WVIDEO, available from Novell, Inc.

(Other third-party debuggers are under development)

Appendix E: Version Differences

Prior Version Compatibility

Drivers designed to v3.1x specification are compatible with version 4.xx. There are no major architectural changes to the driver environment in v4.xx. However, the new memory management implementation more strictly enforces memory access specifications. All accesses to memory external to the driver code must either be made to memory that has been registered with NetWare (using *RegisterHardwareOptions*), or made through the *ReadPhysicalMemory* support routine. Also, some I/O Control (IOCTL) functions have been updated to reflect the richer development resources provided for the assorted storage devices such as autochangers, magazines, and tapes. Some API support routines have been replaced with new ones more conducive to the new OS; however, the old API's are still emulated.

Changes To Structures

Chapter 2 shows the fields of structures used by the driver. Changes have been made as follows:

IORequestStructure	The last three parameter names have been changed from NumberOfSectors, SectorNumber , and BufferAddress to generic names (Parameter 1, Parameter 2, etc), to allow for various devices.
IOConfigStructure	Entries have been added to the Flags field, and a CmdLineOptionStr has replaced the Reserved2 field.

New Routines Available With Version 4.xx

Several new **optional** routines have been included in the OS for device, disk, and LAN drivers to help provide hardware independence. The functions accomplished by these optional routines are currently accomplished directly by v3.1x drivers. The new optional routines added are:

*CPSemaphore
*CVSemaphore
ReadPhysicalMemory

* routines that were available in v3.1x but were not fully documented in v1.6 of the Device Driver Specification.

The above routines are described in Chapter 7.

Upgrading Version 3.1x Drivers to 4.xx

The following is a checklist for converting version 3.1x drivers to be compatible with version 4.xx:

1) Replace all occurrences of the *FreeSemiPermMemory* support routine with the new *Free* call. Parameters remain the same.

2) Replace all occurrences of the *AllocSemiPermMemory* support routine with the new *Alloc* call. Parameters remain the same.

3) Replace all occurrences of the *CRescheduleLast* support routine with the new *CYieldIfNeeded* or *CYieldWithDelay* call. Parameters remain the same.

4) Replace all occurrences of the *QueueSystemAlert* support routine with the new *NetWareAlert* call. Parameters have changed.

5) Use the new *ReadPhysicalMemory* support routine for **all** accesses of the BIOS or ROM addressing areas that **have not been registered** with the NetWare OS using the *RegisterHardwareOptions* support routine. Old methods of accessing these system resources should be replaced.

6) All End-of-Interrupt (EOI) calls should be replaced with *CDoEndOfInterrupt.*

7) The OS has been updated to call the new I/O Control (IOCTL) function/sub-function codes. This should not be a problem as displaced old function/sub-function numbers (see Chapter 5, pp. 5-10) were only defined but never implemented.

8) (Optional) Include a line parameter parser in the initialization routine that obtains the server administrator-assigned HBA card number.

Appendix F: Hardware Configuration Information

Obtaining ISA Configuration Information

The ISA BUS does not provide a standardized way to obtain hardware configuration information. Individual slots cannot be queried to determine the adapter type which is installed, nor can adapters be enabled or disabled in a uniform way. Drivers must utilize the parameters passed from the ParseDriverParameters call (parameters supplied in load command line), then verify that the hardware is present and operational as specified. Some adapters may allow all other parameters to be obtained by I/O commands once a primary I/O port is identified, but drivers will still have to interpret the fields thus obtained.

Obtaining EISA Configuration Information

NetWare device drivers on DOS-based servers can obtain EISA information by the following procedures:

Getting the Real Mode Workspace

In order to read EISA configuration information the driver must get a block of memory addressable in both real and protected mode. The EISA machine BIOS uses this block of memory to pass configuration information to the driver.

In order to create this special block of memory, the driver must use the operating system routine *GetRealModeWorkSpace*. Before doing this, however, the driver must allocate five storage locations:

- A double word to hold the size of the block of memory (in bytes)
- A word to hold the offset of the real mode memory address of the block
- A word to hold the segment of the real mode memory address of the block
- A double word to hold the protected mode logical address of the block
- A double word to hold a pointer to a semaphore structure. The driver uses the semaphore to "lock" the memory for its exclusive use while reading the EISA configuration information.

These locations can be reserved in the driver's data area, as the following example shows:

```
WorkSpaceSize                dd  0   ;memory block size in bytes
WorkSpaceRealModeOffset          dw  0   ;block offset
WorkSpaceRealModeSegment         dw  0   ;block segment
WorkSpaceProtectedModeAddress    dd  0   ;block address
WorkSpaceSemaphore               dd  0   ;block semaphore
```

Once the memory has been allocated by the driver, the driver pushes the addresses of the storage locations on to the stack before calling *GetRealModeWorkSpace*.

GetRealModeWorkSpace provides the driver with access to the special block of memory by filling in the storage locations that the driver passed to the operating system on the stack. On return, the driver must clean up the stack. An example of the above call is shown below:

```
push   OFFSET WorkSpaceSize
push   OFFSET WorkSpaceRealModeOffset
push   OFFSET WorkSpaceRealModeSegment
push   OFFSET WorkSpaceProtectedModeAddress
push   OFFSET WorkSpaceSemaphore
call   GetRealModeWorkSpace
add        esp, (5*4)
```

Locking the Memory

During the EISA configuration read operation, the driver must have exclusive use of the special memory block. It must "lock" the memory block by calling the *CPSemaphore* function, as shown in the following example:

```
push   WorkSpaceSemaphore       ;load semaphore
call   CPSemaphore              ;lock work space
add        esp, (1*4)               ;adjust stack
```

Please note that the driver must restore the stack upon return, as always.

Making a Real Mode BIOS Call

In order for the EISA machine BIOS to pass the configuration data for the selected physical card back to the driver, the driver must make a real mode call to the EISA BIOS. The driver must allocate memory for two structures (InputParms and OutputParms) whose format is defined below:

```
InputParamStruct      struc
  IAXRegister             dw  ?
  IBXRegister             dw  ?
  ICXRegister             dw  ?
  IDXRegister             dw  ?
  IBPRegister             dw  ?
  ISIRegister             dw  ?
  IDIRegister             dw  ?
  IDSRegister             dw  ?
  IESRegister             dw  ?
  IIntNumber              dw  ?
InputParamStruct      ends
```

```
OutputParamStruct        struc
  OAXRegister            dw  ?
  OBXRegister            dw  ?
  OCXRegister            dw  ?
  ODXRegister            dw  ?
  OBPRegister            dw  ?
  OSIRegister            dw  ?
  ODIRegister            dw  ?
  ODSRegister            dw  ?
  OESRegister            dw  ?
  OFlags                 dw  ?
OutputParamStruct        ends


InputParms           InputStructure<>
OutputParms          OutputStructure<>
```

Before making the *DoRealModeInterrupt* call, the driver must fill in the InputParms structure in the following way:

IAXRegister — The read configuration parameter 0D801h (See the EISA BIOS call information supplied by the EISA computer manufacturer).

ICXRegister — The adapter slot and block of configuration data to read. CL is the slot and CH is the block.

IDSRegister — The real mode segment address of where to put the block of data. This value was returned in the "WorkSpaceRealModeSegment" variable by *GetRealModeWorkSpace*.

ISIRegister — The real mode memory offset of where to put the block of data. This value was returned in the "WorkSpaceRealModeOffset" variable by *GetRealModeWorkSpace*.

Iintnumber — The interrupt number. In this example, it is interrupt 15h.

After filling out the InputParms structure, the driver pushes the offsets of InputParms and OutputParms and then calls *DoRealModeInterrupt*, as shown below (please note that the driver must restore the stack upon return):

```
push   OFFSET OutputParms    ;ptr to output registers
push   OFFSET InputParms     ;ptr to input registers
call   DoRealModeInterrupt   ;cause switch to real mode, then int
lea        esp, [esp + (2*4)]    ;restore stack
```

Error Checking

To determine if *DoRealModeInterrupt* executed without errors, compare EAX to 0. If the value is 0, the routine executed successfully.

The driver must also detect errors the BIOS routine may have had. It does this by checking the OAXRegister field in the OutputParms structure. To determine if the BIOS routine executed without errors, compare the OAXRegister field to 0. If the value is 0, the routine executed successfully. A sample of the error checking code required follows:

```
cmp        eax, 0                ;was successful?
jne        IntNotValidErrorExit  ;jmp if OS rtn error
cmp        BYTE PTR OutputParms.OAXRegister+1, 0
jne        IntNotValidErrorExit  ;jmp if BIOS error
```

Note: The error handling routines that the driver jumps to must unlock the block of memory by calling *CVSemaphore*.

At this point, the driver has access to the configuration of the adapter set by the user in the EISA configuration utility. The driver accesses this information using the logical address (protected mode address) of the special memory block, which was returned during the *GetRealModeWorkSpace* call. A sample of typical driver processing follows:

```
mov        si, WorkSpaceProtectedModeAddress   ;load pointer to data
movzx  ecx, BYTE PTR [esi+INTERRUPTOFFSET]     ;get int, if any
and        cl, ISOLATEINTMASK                  ;isolate interrupt level
jecxz  NoAddInterrupt                       ;if none skip add
mov        SaveInterrupt, cl                ;save interrupt for later
```

Each configuration block contains different information (interrupts, memory, etc.). If the first block read does not contain the appropriate information, keep reading blocks by incrementing CH in the InputParms structure and calling DoRealModeInterrupt again. Read blocks until the information is obtained or until Int 15h returns an 81h in AL (BYTE PTR OutputParms.OAXRegister+1).

Note: INTERRUPTOFFSET is defined in the EISA specification.

Unlocking the Memory

Finally, the driver must unlock the special memory block that the EISA configuration data is located in. This is accomplished by making a call to the *CVSemaphore* function, as indicated in the following example:

```
push   WorkSpaceSemaphore    ;pass semaphore
call   CVSemaphore           ;unlock workspace
add        esp, (1*4)            ;clean up stack
```

Appendix G: 16-bit Host Adapter Support

Machines supporting more than 16 megabytes of RAM may have problems with 16-bit host adapters originally designed for the 24-bit address bus of IBM AT compatibles, or earlier Micro Channel Architecture Busses which could only address 16 megabytes. If the driver in question does not support a 16-bit Host Adapter using shared RAM or DMA (including Bus-Master DMA) which may be used in systems with more than 16 megabytes, the following information is not relevant.

Potential Problems

EISA Machines utilizing the Intel EISA Bus Controller chip usually disable memory control signals to slots which have a 16-bit ISA card installed if a memory access above 16 Megabytes occurs. However, it may be possible that some machines may not disable control signals to slots with 16-bit Host Adapters for memory accesses above 16 megabytes, either due to an interpretation of the EISA specification or because the motherboard is not properly configured for the slot. This may cause host adapters with shared RAM to respond when they are not being addressed, causing bus contention between memory and the host adapter (and eventual failure of bus drivers on both the motherboard and the host adapter).

This is not a problem with Micro Channel Architecture (MCA) machines since 16-bit Host Adapters designed to specifications look at the "MADE24" signal to determine if the access is above 16 megabytes (and thus beyond the Host Adapters range of addressing). Also, some EISA motherboards may not hold the high-order unused address lines (24-31) to logic zero during Bus-Master DMA transfers for 16-bit Host Adapters (and thus cannot be used with 16-bit Bus-Master DMA host adapters in configurations with more than 16 meg). This problem is **only solvable** if it is caused by improper configuration of the EISA system, and requires a proper reconfiguration to prevent the control signals from being activated in slots with 16-bit adapters for accesses above 16 megabytes.

A 16-bit Host Adapter (ISA or MCA) may also use shared RAM which maps just below the 16 meg boundary. System memory conflicts with adapter shared RAM **must be resolved** (usually memory may be disabled or relocated, or the shared RAM may be mapped below 1 meg (there is typically a hole with at least 16K to 64K somewhere between 000C0000h and 000F8000h).

NetWare drivers supporting 16-bit DMA (MCA only) or Bus-Master DMA Host Adapters on MCA or EISA machines with more than 16 megabytes will encounter another problem when the driver gets an I/O request with the ending address extending into the area above 16 megabytes (the Host Adapter cannot do transfers above 16 meg). **This problem is solvable**. A new memory allocation pool and new allocation routines were added to versions 3.11 and above to address this problem, allowing drivers to request buffers which are specifically below 16 megabytes, which the driver can either copy to or from, and which must be used for all server I/O requests which end above 16 megabytes. A new NLM, BELOW16, has been designed to provide support for these routines in version 3.10. Please note that the number of buffers (cache block size) allocated specifically for this purpose is user-setable from a minimum of 8 to a maximum of 200 (for 3.1x) or 300 (for 4.xx). The default for versions 3.1x and 4.xx and for the BELOW16 NLM is 16.

Procedure

After Loading the Server

Version 3.11 and subsequent versions:

Use the setable parameter "Reserved Buffers Below 16 Meg" to reserve the number of buffers required for all drivers supporting 16-bit adapters. The minimum number of buffers that can be reserved is 8, and the maximum is 200. (The maximum for v4.xx is 300.) The default for this parameter is 16 buffers. Indicate enough buffers for all drivers which will be loaded later and which will require use of these buffers. Do not set this parameter higher than **required**, as it will have a significant effect on server performance.

Version 3.10 (allocate default number of buffers):

Load the driver (the drivers definition file must specify that the BELOW16 NLM module must be loaded first, using the MODULE directive). This will cause the BELOW16 NLM to default to 16 buffers allocated specifically for drivers requiring buffers below 16 megabytes.

Version 3.10 (override default number of buffers):

Load the BELOW16 NLM (must be **first** NLM loaded) using the format "load below16.nlm buffers=nn", where nn is a value between 8 (minimum) and 200 (maximum). The number indicated must be large enough for all drivers requiring intermediate I/O buffers below 16 megabytes, since these buffers will be shared with all drivers requiring them. Do not specify a number larger than required, as indicated below.

Now load any drivers requiring buffers below 16 megabytes.

Driver Initialization

Allocate at least a single buffer for each Host Adapter used potentially for I/O and the Scan for Devices function, using the "AllocBufferBelow16Meg" allocator (see chapter 7). Retain the buffer address in a special list so that it can be identified for use with requests which end above 16mb, also to return to the system for clean-up. The buffers allocated by the driver from this memory pool are intended to be kept by the driver until the driver is unloaded*****, and not allocated dynamically upon demand (the allocation routine is blocking and **may not be called** either from Interrupt Level or from the Driver IOPollRoutine level). Additional memory allocated by the driver for control blocks and other driver purposes must be allocated using the Alloc memory allocation routine.

Driver Scan For Devices

After finding a device using the buffer allocated during initialization (or allocating an additional buffer if the original is busy with I/O) determine the number of outstanding requests that can be active simultaneously with the Host Adapter for the driver, and allocate them using the "AllocBufferBelow16Meg" allocator. It may be necessary for the driver to limit the number of requests that may be outstanding at one time. Many Host Adapters will never require more than one buffer, because many Host Adapters do not support disconnect or in some way are limited to one actual request pending at any time.

* **Note:** The OS memory allocators used by the drivers may not be called at interrupt or process non-blocking level, but are all meant to be used to allocate memory which is to be kept by the driver until the driver is unloaded from the NetWare v3.xx or v4.xx Operating Systems. Using the memory allocators in an interactive environment will degrade performance both of the driver and also the cache, since the blocks are taken from the cache and the cache blocks flushed prior to being returned to the caller.

Driver Active I/O Operation

Initiating I/Os

After acquiring each request* by calling GetRequest:

1- Call MapDataOffsetToAbsoluteAddress to get the real absolute memory address of the request

2- Add the length of the transfer request (or requests if combining adjacent requests) in bytes to the starting real absolute memory address.

3- If the result is below 16 megabytes, proceed to perform I/O normally directly to or from the actual request buffer (this will allow greatest throughput).

4- If the result is above 16 megabytes, use one of the special buffers the driver allocated at initialization below 16 meg. If a write, move the data from the request buffer to the special pre-allocated buffer. Flag the buffer as in use so that the driver will not attempt to use it simultaneously for another request.

* **Note:** Additional memory may be registered above 16 megabytes at any time, requiring drivers to check dynamically all requests to determine if they end above 16 meg.

At interrupt or I/O completion:

1- If the I/O just completed was a read and a special driver pre-allocated buffer below 16 megabytes was used, move the data from the special buffer to the original request buffer above 16 megabytes.

2- For all requests originally above 16 meg, now mark the special buffer available for future driver needs for that adapter.

Driver Unload

a) Wait for all requests to be completed (as normal).

b) Return each buffer previously acquired with the "AllocBufferBelow16Meg" allocator by calling "FreeBufferBelow16Meg" providing the buffer address originally obtained.

Appendix H: NetWare Ready Support

Notice of Discontinuance

Novell has discontinued the NetWare Ready program and will no longer certify products to that standard. It was determined that the original objectives of the program were either being met though other means, or were not consistent with Novell Lab's charter.

The primary objective of the NetWare Ready program was to verify hardware and hardware driver compatibility with the NetWare operating system. Each product was categorized and then tested for compatibility against all complementary products of the same category while running NetWare. A secondary objective tested the hardware for reliability. For example, a SCSI disk drive would be tested for compatibility with several previously-certified drivers and host bus adapters while running in a NetWare file server. The SCSI drive was also stress tested under load comparable to a heavy network environment.

Novell determined that, given the proliferation of products and interface standards available, attempts to classify products would be artificially restrictive. Also, the testing required would demand excessive manpower and resources. A better alternative is provided in the **"Yes, NetWare Tested and Approved"** program. There, strong emphasis is placed on testing the compatibility between NetWare and the NetWare driver software. It is the product manufacture's responsibility to resolve hardware compatibility and reliability issues and designate the configuration to be used.

Appendix I: PCI and PCMCIA Support

This document describes how PCI and PCMCIA are supported on **NetWare v4.10**. The APIs described herein area subset of those to be provided in a later version of NetWare. Although NetWare v4.10 does not fully support a multi-bus architecture, the following APIs allow drivers to be loaded on a PCI or PCMCIA bus that coexist in the same IO space as an ISA, EISA or MCA bus. Comprehensive multi-bus support is now provided through the NWPA and NBI nlms. The interface architecture and APIs are described in the *NetWare Peripheral Architecture (NWPA) Functional Specification and Developer's Guide,* Version 2.1D September 1995 or later.

In versions of NetWare before v4.10, a single driver could be written to support similar adapters designed for different bus architectures. The drive accomplished this by asking the NetWare Operating System (by calling *GetHardwareBusType*) what bus type the machine supported, and then using this information (bit 0 = MCA, bit 1 = EISA) to make the appropriate calls to initialize the driver. In this way, the same driver that was loaded to support an adapter on an EISA bus machine could be loaded to support an adapter on a MCA bus machine, if the driver called *GetHardwareBusType* and initialized the adapter based upon which bit was set. This methodology works fine for machines that support only one bus type. However, with the introduction of PCI and PCMCIA it has become more common for a single machine to support multiple bus types. Thus, if *GetHardwareBusType* were to be extended to define bits for PCI and PCMCIA, this would be insufficient because both PCI and EISA buses may be supported by the machine in question. Worse yet, a single machine may support multiple PCI buses.

PIN NetWare has designed a solution for the larger problem of a single machine supporting multiple buses of the same type, or even of this solution in order to solve the immediate problem of supporting PCI and PCMCIA buses as well as help move drivers in the long term direction that NetWare is heading.

PCI or PCMCIA support is provided in the following manner:

Two new fields have been defined in the IOConfigurationStructure:

LONG CBusTag: Contains a tag that can be used to determine the bus type associated with a particular adapter. *ParseDriverParameters* will now parse for the token 'BUS=(string)' and stuff a tag associated with the specified bus (if supported) into CBusTag. The strings supported by NetWare v4.10 are: ISA, EISA, MCA, PCMCIA and PCI. If no 'BUS=(string)' token is specified, the CBusTag field will not be modified. A NULL bus tag can be used to specify the primary or default system bus.

WORD CIOConfig Version:
Contains a 1. This specifies that the CBusTag field is defined and used by the driver. This number may change in the future to allow the definition of IOConfigurationStructure to be changed again for future enhancements.

```
typedef struct IOConfigurationStructure  {
        LONG Reserved0;
        WORD Flags;
        WORD Slot;
        WORD IOPort0;
        WORD IOLength0;
        WORD IOPort1;
        WORD IOLength1;
        LONG MemoryDecode0;
        WORD MemoryLength0;
        LONG MemoryDecode1;
        WORD MemoryLength1;
        BYTE Interrupt0;
        BYTE Interrupt1;
        BYTE DMAUsage0;
        BYTE DMAUsage1;
        LONG IORTag;
        LONG Reserved1;
        BYTE *CmdLineOptionStr;
        BYTE Reserved3[18];
        LONG LinearMemory0;
        LONG LinearMemory1;
        WORD Reserved4;
        LONG CBusTag;
        WORD CIOConfigVersion;
    } IOConfigStruct;
```

Now that the driver has obtained a bus tag, it can use this tag to ask the NetWare Operating System for the bus type by calling *GetBusType* as defined below:

```
/*****************************************************************************/
LONG GetBusType(
        LONG busTag,
        LONG *busType);
```

LONG busTag: Bus tag for the bus in question. NULL specifies primary bus.

LONG *busType: A place to put a value that indicates the bus type. The defined values are:

0 = PC ISA bus
1 = PC MCA bus
2 = PC EISA bus
3 = PCMCIA bus
4 = PCI bus
TBD = other bus types yet to be defined

Returns: 0 = Successful
4 = Parameter error (invalid bus tag)

Other bus calls supported by the NetWare v4.10 that may be useful are the following:

```
/*****************************************************************************/
LONG GetBusName(
        LONG busTag,
        BYTE **busName)
```

LONG busTag: Bus tag for the bus in question. NULL specifies primary bus.

BYTE **busName: A place to put a pointer to a NULL-terminated string (the name of the specified bus)

Returns: 0 = Successful
4 = Parameter error (invalid bus tag)

```
/*****************************************************************************/
LONG GetBusTag(
        BYTE *busName,
        LONG *busTag)
```

BYTE *busName: Pointer to a NULL-terminated string (the name of the bus).
NULL means return tag for the primary bus.

LONG *busTag: A place to but the bus tag.

Returns: 0 = Successful
6 = Item not present

```
/*****************************************************************************/
LONG ScanBusInfo(
        LONG *scanSequence,
        LONG *busTag,
        LONG *busType,
        BYTE **busName)
```

LONG *scanSequence: Initialized to -1 for the first search and then passed back for subsequent calls.

LONG *busTag: A place to put the bus tag.

LONG *busType: A place to put the bus type.

BYTE **busName: A place to put a pointer to a NULL-terminated string (the name of the specified bus).

Returns: 0 = Successful
4 = parameter error (invalid sequence)
7 = No more items

```
/*****************************************************************************/
LONG DoRealModeInterrupt32(
        InputParametersStructure32 *InputStructure32,
       OutputParametersStructure32 *OutputStructure32)
```

InputParametersStructure32 *InputStructure32: A pointer to a structure holding the pertinent input register values.

OutputParametersStructure32 *OutputStructure32: A pointer a structure where the register values are returned.

Returns: 0 & zero flag set = interrupt vector was called
1 & zero flag cleared = interrupt vector not called (DOS not available)

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

typedef  InputParamStruct32{
    LONG IEAXReg;
    LONG IEBXReg;
    LONG IECXReg;
    LONG IEDXReg;
    LONG IEBPReg;
    LONG IESIReg;
    LONG IEDIReg;
    WORD IDSReg;
    WORD IESReg;
    WORD IFSReg;
    WORD IGSReg;
    BYTE IIntNumber;
    BYTE IDummy32[3];
} InputParametersStructure32;

typedef OutputParamStruct32{
    LONG OEAXReg
    LONG OEBXReg
    LONG OECXReg
    LONG OEDXReg
    LONG OEBPReg
    LONG OESIReg
    LONG OEDIReg
    WORD ODSReg
    WORD OESReg
    WORD OFSReg
    WORD OGSReg
    LONG OFlags32
} OutputParametersStructure32;
```

Appendix J: NetWare CD-ROM Support

NetWare versions 3.12 and 4.xx support CD-ROM device drivers by way of an NLM. The NLM uses the standard ISO9660 and High Sierra file formats to implement read-only volumes on v3.12/v4.xx file servers. The following list describes the details that are necessary for developers to consider when developing NetWare device drivers for CD-ROM devices:

1) Scan for devices
   The device driver must register the device with the OS even if the removable compact disk media is not present in the device. During the driver's scan for devices routine (see chap. 4), the driver can call *AddDiskDevice* with dummy parameters in place of **TotalSize** and **DriveParameters**. When the driver is locating devices that are attached to the adapter card, NetWare only cares that there is a device that exists. At this point, NetWare does not care about details such as the capacity of the removable media.

   The actual dimensions of the storage media will be registered with the OS when the OS calls the *ReturnDeviceStatus*, *ReturnDeviceInfo*, and *ReturnMediaInfo* I/O control (IOCTL) routines (See item 2 below).

   During the scan, the driver must also register the device as ReadOnlyDevice (01h) and RemovableDevice (02h) when the access flags are passed as part of the driveSizes parameter to the *AddDiskDevice* routine (see chap. 7). The flags ensure that the OS will call the correct IOCTL routines.

2) IOCTL function calls
   The OS uses the *ReturnDeviceStatus*, *ReturnDeviceInfo*, and *ReturnMediaInfo* IOCTLs to register the correct capacity of the media (in 512-byte sectors) when it is present in the device. To maintain compatibility with other operating system partitions, the physical dimensions (hd, sect, cyl) of the driver are also registered. The product of these dimensions must equal or exceed the capacity of the media. The driver must also indicate in these IOCTLs that it will suport the NetWare 512-byte sector size and perform any needed sector translation as described in item 3 below. (Also, see the IOCTL descriptions in chapter 5.) If no media is present, the driver should return an error condition to the OS.

   NetWare will call *LogicalDeviceMount* and *LogicalDeviceDismount* at some point during device operation. NetWare uses these IOCTLs to verify that media is present and/or inserted. These functions could also be used to spin up or spin down the device (see chap. 5).

   NetWare will also call *LockDeviceMedia* and *UnlockDeviceMedia* to lock or unlock the media in the device. Disabling the eject button on the device ensures that the media will not be ejected while in use.

3) OS data reads
   NetWare v3.12/v4.xx assumes that all devices that are attached to the file server will return data in 512-byte sectors. Most CD-ROM devices have a file format of 2048-byte sectors. The NetWare driver must translate the CD-ROM device sector size into the 512-byte sectors. All devices that are to be used as NetWare volumes that have a different sector size than the required 512-byte sectors must perform 2048-to-512-byte sector translation.

   For example, if NetWare requested to read an 8-sector block of data starting at sector 64, the translation for 2048-byte sectors would start at physical sector 16 and read 2 sectors.

Appendix K: Sequential Access Device Drivers (Tapes)

There are several new IOCTL functions and sub-functions designed to help customize device drivers for tape devices and other similar devices. In addition to these functions, there are several items that developers should be aware of when developing drivers for sequential access media.

1) The device driver interface was originally designed for developing disk drivers only. Consequently, the names of some system calls use the word "disk." For example, an essential call for registering devices with the OS is AddDiskDevice. Sequential Media drivers use this call to register their devices also.

2) Some commands require the driver to issue two or more commands in order to complete a request. A driver can either use an asynchronous event schedule (AES) thread to issue a series of blocking calls, or the driver must devise a mechanism that issues succeeding calls from within the interrupt handler in order to use a series of non-blocking calls.

3) Drivers can ignore implementation of I/O Control (IOCTL) requests and regular I/O requests that pertain to disks such as RandomRead, RandomWrite, FormatDevice (unless required by the sequential access device) and ReturnBadBlockInfo.

4) When a tape error is detected by the driver, it must dequeue the the corresponding device's request list by issuing *GetRequest*s and *PutRequest*s until the list is exhausted (*GetRequest* returns a zero.) The requests must be returned (using *PutRequest*) with a "Abort - Prior State" CompletionCode.

Appendix L: Installation Information File

Introduction

To facilitate the ability to programmatically install device drivers, installation programs must know the parameters associated with each driver, the interactions that are required from the user, and how to set up the respective configuration file(s). For information describing the syntax of the driver text file used to provide installation utilities with the required information, refer to the *NetWare Peripheral Architecture (NWPA) Functional Specification and Developer's Guide*, Version 2.1D September 1995 or later, Appendix C.

Appendix M: OS/2 Drivers

From a developer perspective, there is no difference in writing OS/2 driver and native NetWare drivers. The following are items for all driver developers to remember that especially apply to OS/2-based drivers:

- In general all system hardware must be registered with the NetWare OS and accessed using API calls when available. Do not access them directly.
- Use *CDoEndOfInterrupt* for tasks involving EOI's.
- Use *CEnableHardwareInterrupt* and *CDisableHardwareInterrupt* to mask and unmask individual interrupts.
- When accessing the DOS address space for BIOS information, use *ReadPhysicalMemory* or register the area to be accessed using *RegisterHardwareOptions*.
- Use *MapDataOffsetToAbsoluteAddress* and *MapAbsoluteAddressToDataOffset* to translate between linear (logical) and physical addresses of memory that has been registered with the NetWare OS.

The API set used to develop device drivers has been generalized to adapt to the OS/2 environment. Logically the driver interface is the same for native NetWare running under OS/2 as for native NetWare.