

## 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 drivename<Enter>
```

Where: drivename 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:

<b>TYPE</b>	(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)
<b>DESCRIPTION</b>	(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"
<b>OUTPUT</b>	(Required) Specifies the name of the output file (the extension will be added by the linker as specified above).
<b>INPUT</b>	(Required) Specifies the name of the input .OBJ file(s).
<b>START</b>	(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.
<b>EXIT</b>	(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.
<b>CHECK</b>	(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).
<b>COPYRIGHT</b>	(Required) Inserts the Novell default copyright or the copyright for a third-party developer. Usage: COPYRIGHT                      Novell default. COPYRIGHT "company"      Third party copyright message.

<b>VERSION</b>	(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.)
<b>REENTRANT</b>	(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 <u>mutually exclusive</u> (both cannot be specified) with the keyword <b>MULTIPLE</b> .
<b>MULTIPLE</b>	(Optional) Specifies that more than one code image of the loadable module may be loaded into file server memory. This keyword is <u>mutually exclusive</u> (both cannot be specified) with the keyword <b>REENTRANT</b> .
<b>CUSTOM</b>	(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.
<b>MAP</b>	(Optional) Directs the linker to create a map file.
<b>IMPORT</b>	(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.
<b>EXPORT</b>	(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.
<b>MODULE</b>	(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.
<b>DEBUG</b>	(Optional) Specifies that the linker will include debug information in the output file.
<b>@ operator</b>	(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:

<b>h</b>	displays help for general commands
<b>hb</b>	displays help for breakpoints
<b>he</b>	displays help for grouping operators
<b>.h</b>	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    5  ^  bitwise XOR       8
 + add         3  or equal-to      5  |  bitwise OR        9
 - subtract     3  <= less-than or  5  && 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, FLSE, 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
FFF77E00 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).

Zexpression 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

<b>Symbol</b>	<b>Description</b>	<b>Precedence</b>
<i>!</i>	<i>logical not</i>	<i>1</i>
<i>-</i>	<i>2's compliment</i>	<i>1</i>
<i>~</i>	<i>1's compliment</i>	<i>1</i>
<i>*</i>	<i>multiply</i>	<i>2</i>
<i>/</i>	<i>divide</i>	<i>2</i>
<i>%</i>	<i>mod</i>	<i>2</i>
<i>+</i>	<i>addition</i>	<i>3</i>
<i>-</i>	<i>subtraction</i>	<i>3</i>
<i>&gt;&gt;</i>	<i>bit shift right</i>	<i>4</i>
<i>&lt;&lt;</i>	<i>bit shift left</i>	<i>4</i>
<i>&gt;</i>	<i>greater than</i>	<i>5</i>
<i>&lt;</i>	<i>less than</i>	<i>5</i>
<i>&gt;=</i>	<i>greater than or equal to</i>	<i>5</i>
<i>&lt;=</i>	<i>less than or equal to</i>	<i>5</i>
<i>==</i>	<i>equal to</i>	<i>6</i>
<i>!=</i>	<i>not equal to</i>	<i>6</i>
<i>&amp;</i>	<i>bitwise AND</i>	<i>7</i>
<i>^</i>	<i>bitwise XOR</i>	<i>8</i>
<i> </i>	<i>bitwise OR</i>	<i>9</i>
<i>&amp;&amp;</i>	<i>logical AND</i>	<i>10</i>
<i>  </i>	<i>logical OR</i>	<i>11</i>

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**

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

## **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 ParseDriverParameter  
s**

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 APIs 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 (InputParams and OutputParams) whose format is defined below:

```
InputParamStruct    struc
    IAXRegister      dw ?
    IBXRegister      dw ?
    ICXRegister      dw ?
    IDXRegister      dw ?
    IBPRegister      dw ?
    ISIRegister      dw ?
    IDIRegister      dw ?
    IDRegister       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
```

