Novell Developer Kit

www.novell.com

February 2008

NLM™ DEVELOPMENT CONCEPTS, TOOLS, AND FUNCTIONS

Novell®

Legal Notices

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.
www.novell.com

Online Documentation: To access the online documentation for this and other Novell developer products, and to get updates, see developer.novell.com/ndk. To access online documentation for Novell products, see www.novell.com/documentation.

Novell Trademarks

For a list of Novell trademarks, see Trademarks (http://www.novell.com/company/legal/trademarks/tmlist.html).

Third-Party Materials

All third-party trademarks are the property of their respective owners.

Contents

About This Guide

NLM Development Concepts, Tools, and Functions provides concept and task information for the NLM developer. This documentation describes many of the concepts and tasks required for NLM development in each phase of development:

In addition, it provides function references and development information for the following services:

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Documentation Updates

For the most recent version of this guide, see NLM and NetWare Libraries for C (including CLIB and XPlat) (http://developer.novell.com/ndk/clib.htm).

Additional Information

For information about other CLib and XPlat interfaces, see the following guides:

- *NDK: Program Management*
- *NDK: NLM Threads Management*

- *NDK: Connection, Message, and NCP Extensions*
- *NDK: Multiple and Inter-File Services*
- *NDK: Single and Intra-File Services*
- *NDK: Volume Management*
- *NDK: Server Management*
- *NDK: Client Management*
- *NDK: Network Management*
- *NDK: Internationalization*
- *NDK: Unicode*
- *NDK: Sample Code*
- *NDK: Getting Started with NetWare Cross-Platform Libraries for C*
- *NDK: Bindery Management*

For CLib and XPlat source code projects, visit Forge (http://forge.novell.com).

For help with CLib and XPlat problems or questions, visit the NLM and NetWare Libraries for C (including CLIB and XPlat) Developer Support Forums (http://developer.novell.com/ndk/devforums.htm). There are two for NLM development (XPlat and CLib) and one for Windows XPlat development.

Documentation Conventions

In this documentation, a greater-than symbol (>) is used to separate actions within a step and items within a cross-reference path.

A trademark symbol (®, ™, etc.) denotes a Novell trademark. An asterisk (*) denotes a third-party trademark.

Getting Started

1

CLib is a C-language library that runs on a NetWare server. It is a legacy library that has been replaced by LibC (http://developer.novell.com/ndk/libc.htm). For new development, you should use LibC unless your application has the following requirements:

- Must run on NetWare 4.x as well as the later versions of NetWare.
- Uses functionality not yet available in LibC.

This getting started section explains how to develop a NetWare server application, an NetWare Loadable Module (NLM). The CLib SDK also contains NetWare client libraries. For information on how to get started with these libraries, see Cross-Platform Libraries.

This documentation describes common tasks associated with writing an NLM:

1.1 Requirements

To use CLib, you need the following:

- A NetWare server, running NetWare 5.1 or later. Although CLib is available on NetWare 3.x and NetWare 4.x, these versions of NetWare are no longer supported and no CLib fixes are being exported to them.
- A development workstation with the CLib SDK installed and a network connection to a NetWare server. You will be copying files from the workstation to the server. NLM development is done on workstation, and the compiled and linked NLM is copied to the NetWare server for loading and testing.

1.2 Installing the CLib SDK

The CLib SDK is part of the Novell® Developer Kit (NDK), and it can be downloaded from the the NDK Web site. The SDK consists of software, sample code, and documentation components, which can be downloaded together or individually. In addition, the documentation and sample code can be viewed on line at the NDK site. LibC library files are installed on the NetWare server during installation and with every support pack.

To install the CLib SDK (http://developer.novell.com/ndk/clib.htm), click the download icon and follow the instructions. During the installation, you must select the destination where you want to

install the CLib components. By default, the executable installs the SDK components to c:\novell in the following subdirectories:

- Software: c:\novell\ndk\nwsdk
- Sample code: c:\novell\ndk\samples\clib_sample
- Documentation: c:\novell\ndk\doc\clib

Because CodeWarrior requires you to install the CLib SDK on the C: drive, we recommend that you install the SDK on that drive.

1.3 Selecting a Compiler

The following compilers are available for developing NLMs:

- Metrowerks CodeWarrior for NetWare (page 14)
- Open Watcom Compiler (page 14)
- GNU and Other Compilers (page 15)

You must link a prelude file into your application. The CLib SDK includes prelude files for the CodeWarrior, Open Watcom, and GNU C/C++ compilers.

1.3.1 Metrowerks CodeWarrior for NetWare

Metrowerks CodeWarrior for NetWare is an Integrated Development Environment (IDE) in which you can edit, compile, and debug your code. CodeWarrior for NetWare will soon support C++ NLM development. It supports the development of server-based applications, such as NLMs. The CodeWarrior for NetWare linker combines object code into NLMs, drivers, and modules.

CodeWarrior also has a source-level debugger for debugging NLMs from a client workstation.

Certain levels of the DeveloperNet Program include a copy of CodeWarrior or allow you to purchase one at a reduced price. For details, see the Novell DeveloperNet Program (http://developer.novell.com/brochure).

1.3.2 Open Watcom Compiler

Open Watcom compiler is a cross-platform compiler that produce object files for multiple operating systems. It has both command-line and IDE interfaces and generates 32-bit protected mode code. It has the usual complement of switches to specify such things as

- Whether to include debug information in the object file
- The name of the object file (if other than the default)
- What directory or directories to get include files from
- The amount and kind of optimization to perform

Watcom was the first compiler to support NLM development. It supports developing C applications for NetWare, but does not support C++ applications for NetWare. The Watcom linker does not support symbol prefixing except through use of the ALIAS link directive. Because "@" is overloaded in linker syntax, the solution is difficult and requires quoting. Do not attempt to include Watcom libraries.For more information, see Open Watcom (http://www.openwatcom.org).

1.3.3 GNU and Other Compilers

The GNU C/C++ compiler is used on many UNIX* and Linux* systems, but it also supports NLM development. For more information, see GNU Compiler Collection (http://gcc.gnu.org).

Other C/C++ compilers such Borland C++ and Microsoft Visual C++ can be used to develop NLMs. You use them to write and compile the code and then use a linker from another vendor, such as Open Watcom.

1.4 Setting Up a Compiler

The following sections explain how to set up various compilers for NLM development.

NOTE: If you have instructions for setting up a compiler for NLM development, which is not included in this section and that you would like to share with other CLib developers, please post these instructions in the CLib newsgroup (http://developer.novell.com/ndk/devforums.htm).

Currently, we have instructions for the following compilers:

- Setting Up CodeWarrior for NLM Development (page 15)
- Setting Up Open Watcom with Borland C++ Builder (page 17)
- Using the WATCOM IDE (page 18)

1.4.1 Setting Up CodeWarrior for NLM Development

You should install the following items on your development machine in the order listed:

- CodeWarrior
- NLM and NetWare Libraries for C (including CLIB and XPlat) (http://developer.novell.com/ndk/clib.htm)
- CodeWarrior PDK for NetWare

See the CodeWarrior documentation for workstation requirements and installation instructions for CodeWarrior and the PDK. The PDK requires a NovellNDK environment variable. For the variable, enter NovellNDK. For its value, enter the location of the CLib SDK (default is c:\novell\ndk\nwsdk). You must either install the CLib NDK on the c: drive or modify some hard coded values in the project's nlm.def file that point to c:\novell\ndk\nwsdk.

The following instructions explain how to create a new stationary project with PDK 4.0 that you can use to compile the sample code.

1. Start CodeWarrior.
2. Select File > New.
3. From the Project tab, select NetWare Stationary.

   NOTE: If you don't select NetWare Stationary, you cannot create a new project.

   ---

4. In the Project name box, enter a name for the project, for example, Sample Code. In the Location box, enter the path where you want to store the project's files, and click OK.
5. In the New Project window, select Server > CLib > Generic NLM C, and then OK.

6. Expand NetWare SDK Libraries. If you do not plan to use the CodeWarrior integrated debugger or C++, delete the mwcrtl.lib and mwcrtld.lib files.

   NOTE: The sample code files will not compile without errors when these libraries are included.

7. Add the clibpre.o file. Right click NetWare SDK Libraries, select Add Files from the drop down menu, browse to the c:\novell\ndk\nwsdk\imports directory, and select clibpre.o.

   If your NLM calls functions in other libraries such as eDirectory functions or Unicode functions, add these import files.

8. Expand Source. You can compile the default program HelloWorld.c or your can select a sample code file.

   For a sample code file, delete HelloWorld.c. Then right click Source and from the drop down menu, select Add Files...

   Browse to the CLib sample code directory (the default location is c:\novell\ndk\samples\clib_sample\nlm), and select one of the sample code files to compile, for example, calendar.c.

   Each sample file is designed to be compiled as a separate NLM.

9. Click the Generic C Build Settings... icon.

10. In the Target category, click NLM Target, modify the fields, and click OK.

    For each NLM that you compile, you need to fill out the following NLM Target fields:

    - Output File—determines the name of the NLM.
    - Screen Name and Initial Screen Name—determines the default screen for the NLM. If the NLM receives input from a user, select User Specified and enter a name in the Initial Screen Name field. If the NLM receives no input from a user, you can select either Console (allows console output of messages) or No Default (prevents console output and input).

      For this sample code project, select User Specified and enter a name that matches the sample file you selected.
    - Initial Thread Name—specifies the prefix that is used to provide thread names. The names appear in the NetWare internal debugger, the CodeWarrior debugger, and the NetWare Remote Manager.
    - Stack Size—determines the maximum size of the stack. For the sample code, the default value of 8192 is adequate.
    - Copyright—enter your company's copyright string.
    - Description—supply a description that fits the NLM. This description appears in various NetWare management utilities.
    - Version—enter a version for the NLM. You must specify at least a major version number.

11. Click the Make icon.

12. When the compile completes, browse to the location of your project files.

13. Copy the compiled *.nlm file to the sys:\system directory of your NetWare server.

    Other locations are possible, but if you create your own directory, you also need to add search paths or include the path in the load command.

14. At the system console, load the NLM.

For more information, see the following:

- Metrowerks manuals, especially the *Targeting the NetWare Operating System PDK 4* manual. This manual explains all the Generic C Build Settings.

  If you prefer a command line interface, the Metrowerks manual explains how to access these development tools and their online documentation.
- Metrowerks newsgroup (http://developer.novell.com/ndk/devforums.htm).

1.4.2 Setting Up Open Watcom with Borland C++ Builder

The following steps explain how to build an NLM by compiling the code with Borland C++ Builder and by linking with WLINK from Open Watcom.

1. Install Open Watcom 1.0 or later.

   Available from http://ww.openwatcom.org
2. Download components from the Novell Developer Kit (cldap_all.exe, clib_all.exe, and ndslib_all.exe) available from http://developer.novell.com.
3. Install the NDK components in c:\Program Files\NDK (or change the following linker definition file to match your location).
4. Using Borland C++ Builder, create a file with the following code.

```
#include <stdio.h>

void main(void)
{
   printf( "Hello, world\n" );
}
```

5. In the project options, turn off the “generate underscores” option.
6. Compile the project. Don't build it, because all you need is the object file (hello.obj).
7. Create a linker definition file for the Watcom Linker. It should look similar to the following:

```
Option OSName = 'Novell NLM'
Format Novell NLM 'Hello World!'

Module CLib
Module Threads

Option CaseExact
Import @C:\Progra~1\NDK\NDK\nwsdk\imports\clib.imp,
@C:\Progra~1\NDK\NDK\nwsdk\imports\threads.imp

Name   hello.nlm
LibF   C:\Progra~1\NDK\NDK\nwsdk\imports\nwpre.obj
File   hello.obj
Option Copyright 'Public domain.  Hello World example'
Option Map=hello.map
Option NoDefaultLibs
Option SymFile=hello.sym
Option Version=1.00.00
```

```
Option Stack=128k
Option ScreenName 'Hello World Example'
Option ThreadName 'HelloWorld_nlm'
```

8 At a command prompt in the directory of the source code and linker definition file, enter

```
wlink @link.lnk
```

You should see the following output from this process:

```
Open Watcom Linker Version 1.0
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public
License.
See http://www.openwatcom.org/ for details.
loading object files
creating map file
creating a Novell NLM executable
```

9 When the linker is finished, copy the NLM to a NetWare 5.1 server or later and load it.

The NLM creates a console screen, called Hello World Example, and prints "Hello, world" to this screen.

1.4.3 Using the WATCOM IDE

This section covers the following:

- Setting Up Your Environment to Use WATCOM IDE (page 18)
- Building HELLO.NLM with WATCOM IDE (page 18)
- Installing Watcom v11 Optimized for NLM Development (page 20)

Setting Up Your Environment to Use WATCOM IDE

1 Install the WATCOM development environment. Include Novell NLM as a target platform.

2 Download and install the software for the *NLM and NetWare Libraries for C*, along with any other component that your NLM development requires, such as the *NDS Libraries for C* or the *Novell Protocol Libraries for C*. Further instructions assume that you install to the default location--C:\NOVELL\NDK.

3 Modify your computer's path statement to include the NDK tools directory (NWSDK/TOOLS).

This must be done so that

- The WATCOM Linker can find the Novell MAKEINIT file.
- The Novell NLM tools (such as MSGLIB.EXE) are accessible

4 Create a MAKEINIT configuration file and an ALL.IMP import file by running MAKEINIT.EXE, located in the NWSDK\TOOLS subdirectory.

Building HELLO.NLM with WATCOM IDE

1 Create a directory for your project. (Example: C:\projects\hello)

2 Activate the WATCOM IDE.

3 From the menu bar, select "File / New Project."

4 Specify the project directory (created in Step 1) and a project name. (Example: HELLO.WPJ)

5 A "New Target" window appears, allowing you to specify a target name and an image type. (By default they should be set similar to the following: Target name = "hello", Image type = "NLM [.nlm]"). Click OK.

6 A "Source Files" window appears, which is initially empty. Press the INSERT key. An "Add File(s) to..." window appears. Enter the name of the project's first `C' source file in the "File Name" field. (Example: HELLO.C). Add other `C' source files as needed.

7 Use the "Add File(s) to..." window to specify the location of the Novell prelude object file:

- Set the "List Files of Type" field to All Files [*.*]
- Set the "Drives" field and "Directories" field to NOVELL\NDK\NWSDK\IMPORTS
- Specify the file CLIBPRE.OBJ and add it to the project

Close the "Add File(s) to..." window.

8 From the menu bar, select "Options / Link for NetWare Switches...," then deselect the "Incremental Linking [op inc]" option. Click OK.

NOTE: In the Watcom 11 IDE, Incremental Linking [op inc] is turned on by default. This feature has been known to make Watcom's Linker (WLINK) appear to be in an endless loop, never completing the link operation.

---

9 Specify import files as explained in the following paragraphs, then click OK:

- From the menu bar select "Options / Link for NetWare Switches..."
- From the scroll bar select "2. Import, Export and Library Switches"
- Check the box labeled "No default libraries [op nod]"
- In the Import files(,):[imp] field, specify.IMP files as needed

  For example, to build the "Hello world" NLM, you must specify the following in the Import files(,):[imp] field):

  ```
  @$(nlm386imp)\threads.imp @$(nlm386imp)\nlmlib.imp
  ```

  Instead of specifying individual import files you can use @$(nlm386imp)\all.imp, which is a compilation of all the import files.

(The macro nlm386imp value is defined in the Novell makeinit file, generated by the MAKEINIT.EXE file in the NWSDK\TOOLS directory.)

10 From the menu bar select "Options / Link for NetWare Switches...," then from the scroll bar select "3. Advanced Switches." Check the box labeled "Case sensitive link[op c]."

This option tells the Watcom Linker to consider case when resolving references to global symbols. For example, this will prevent calling FREE when you wanted to call free.

11 From the menu bar select "Options / C Compiler Switches...." By default, "1. File Option Switches" is selected on the scroll bar. Set the Include directories:[-i] field to $(nlm386hdr).

An IDE Request window might appear asking "Mark all .c files in ` ...' for remake?" If it does, click Yes.

12 From the menu bar select "Options / C Compiler switches...." Select "3. Source Switches." Click the radio button for 1 byte alignment [-zp1], then click OK. If an IDE Request window appears asking "Mark all .c files in '...' for remake?" click Yes.

IMPORTANT: The Watcom 11 compiler structure alignment has changed from a default of 1 byte alignment [-zp1] (Watcom 10.6), to a default of 4 bytes [-zp4]. To build NLMs in Watcom 11, you must specify the 1 byte alignment option [-zp1].

---

13 If necessary, edit one or more of the previously specified `C' files. You can do so by double-clicking one of the files listed in the Source files: window. Here, for instance, is a simple "Hello world" example:

```
#include <stdlib.h>
#include <stdio.h>

void main (void)

{
   printf("Hellp world\n");
   return;
}
```

14 Add the following function to one of your `C' source files (for example HELLO.C):

```
void WATCOM_Prelude(void)

{
   return
}
```

The Watcom 11 compiler constructs OBJ files from C source files. When the Watcom compiler is invoked from the IDE, a reference is automatically implied to an external __WATCOM_Prelude symbol. There is no known method of disabling this behavior from the IDE (even though the reference is not needed or even referenced in the C source). Therefore, your application must provide its own reference as a "shim" or "stub." This function will not be called by the Novell CLIBPRE.OBJ or Dynamic Linked Library NLMs.

The example program HELLO.C now looks something like this:

```
#include <stdlib.h>
#include <stdio.h>

void WATCOM_Prelude(void)

{
   return
}

void main (void)

{
   printf("Hello world\n");
   return;
}
```

15 When source code edits are completed, save the file(s). Then from the menu bar select Targets / Make.

If all went well, you will find a fully functional NLM in the target directory.

Installing Watcom v11 Optimized for NLM Development

This method is specific to the C language environment. It may seem possible to used the Watcom environment to write an NLM in C++, but there are known anomalies that cause NLMs written in

C++ with Watcom to fail Novell NLM certification tests. If you are using C++ to develop NLMs consider using Meterowerks Code Warrior.

1. Insert your Watcom 11 CD into your computer's CDROM drive.

   Click Start > Run > Browse. Select your CDROM drive, then select Setup.exe and click Open > OK.

2. Specify the path at which the Watcom compiler will be installed or click OK to accept the default C:\WATCOM path.
3. Specify the Selective installation type and click OK.
4. Select which components will be installed. Select 32-bit compiler options: (deselect Include 16-bit compilers: if it is selected), uncheck the 32-bit MFC 4.1 support: option and click OK > Target.
5. The next window lets you specify the target operating system. Deselect everything except NetWare and click OK > Other.
6. In the Select other options: screen, deselect everything, and click OK > OK.
7. A dialog appears saying "SETUP32 will now copy any selected files." Click OK. The install program will begin copying files as a bar chart tracks progress.
8. A screen appears indicating that SETUP32 needs to modify the AUTOEXEC.BAT and CONFIG.SYS files. Confirm the modifications.

   If another dialog appears and tells you it will place the original AUTOEXEC.BAT and CONFIG.SYS files in two identified backup files, click OK.

9. A dialog appears stating that you should reboot your computer so that changes will take effect. Click OK.
10. Once the computer reboots, seven subdirectories in the specified installation directory are not needed. If you accepted the default directory, those subdirectories are the following:

    \WATCOM\EDDAT
    \WATCOM\H
    \WATCOM\LIB286
    \WATCOM\LIB386
    \WATCOM\NLM
    \WATCOM\NOVH
    \WATCOM\NOVI

    Using Windows Explorer or the DOS deltree command, remove all seven. When you are finished, the following two will remain:

    \WATCOM\BINNT
    \WATCOM\BINW

1.5 Using a Linker

Linkers are generally supplied with the compiler. CodeWarrior, Watcom, and GNU all supply linkers. Some linkers allow you to specify options and commands on the command line; some don't. All, however, support referencing a linker definition file, which contains the options and commands. Linker commands and options tell the linker how to create your program. For complete information, consult the documentation that comes with your linker.

1.5.1 Specifying a Linker Definition File

A linker definition file is useful when linker input consists of a large number of object files that you do not want to manually enter on the command line each time you link your program. Note that a linker definition file can also include other linker definition files.

The following is a sample linker definition file for the Watcom linker, WLINK:

```
Format Novell NLM '$(COMPOSITE_DESCRIPTION)'
Name$(NLM_NAME).NLM
Option Copyright= '$(COPYRIGHT_STRING)'
Option NLMFlags= $(NLMFLAGS)
Option CaseExact
%if !%defined(OPTIMIZE)
Debug CodeView
Debug Novell
%endif
%foreach FILE in $(SOURCE_OBJECTS)
File$(OBJECT_DIR)\$(FILE)
%endfor
%if %defined(MORE_OBJECTS)
File$(MORE_OBJECTS)
%endif
%if %defined(PROFILE)
File WRuntime.Obj
%endif
Option Start= $(START_FUNC)
Option Exit= $(EXIT_FUNC)
Option Version= $(COMPOSITE_VERSION)
Option Map= $(NLM_NAME).Map
Option SymFile= $(NLM_NAME).Sym
Option NoDefaultLibs
Option Messages= $(MSG_PATH)\$(NLM_NAME).MSG
Import @CLIB.Imp
Export @$(NLM_NAME).Exp
Module $(MODULES)Format Novell NLM $(COMPOSITE_DESCRIPTION)
```

For information about the prelude object file, see Section 2.9, "NLM Startup," on page 48.

The IMPORT directive shown in the example above enables your NLM to import (call) functions in other NLMs. When using the IMPORT directive, you have two choices for specifying the external functions you want to call:

- List each function as an IMPORT entry in the linker definition file.
- Place all the function names in an import file (.IMP) and specify that file as the IMPORT entry in the directive file (as shown with the example above).

For example, part of the NetWare API is CLIB.NLM. It runs in memory, and all the NLMs loaded on the same server can import its functions. To import a NetWare API function from CLIB.NLM, an NLM linker definition file can either list each function it wants to import or specify the CLIB.IMP file, which contains a list of functions exported by CLIB.NLM.

1.5.2 Linker Commands

The following table lists the NLM commands that can be placed in a linker definition file. Not all linkers support all of these commands. Some linkers support additional, linker-specific commands.

Linker Command	CodeWarrior Syntax	Open Watcom Syntax	GNU Syntax
AUTOUNLOAD	AUTOUNLOAD	OPTION AUTOUNLOAD	AUTOUNLOAD
CHECK	CHECK checkProcedure	OPTION CHECK=symbol_name	CHECK< procedure name>
CODESTART	Not supported	Not supported	CODESTART <map file code start offset (decimal or Xhex)>
COPYRIGHT	COPYRIGHT ["String"]	OPTION COPYRIGHT 'string'	COPYRIGHT ["String"]
CUSTOM	Not supported	OPTION CUSTOM=file_name	CUSTOM <custom data file path>
DATASTART	Not supported	Not supported	DATASTART <map file data start offset (decimal or Xhex)>
DATE	DATE month, day, 4-digit year	Not supported	DATE month day 4-digit year
DEBUG	DEBUG	DEBUG NOVELL	DEBUG
DESCRIPTION	DESCRIPTION "String"	FORMAT NOVELL 'description'	DESCRIPTION "String"
EXIT	EXIT exitProcedure	OPTION EXIT=symbol_name	EXIT <exit procedure name>
EXPORT	EXPORT symbolList | @symbolListFile	EXPORT entry_name {,entry_name} | @symbolListFile	EXPORT <symbol list> | @<symbol list file>
FLAG OFF	FLAG OFF value	Not supported	FLAG OFF <decimal value>
FLAG ON	FLAG ON value Flags can be specified one at a time.	OPTION NLMFLAGS=value All flags must be specified in one NLMFLAGS command.	FLAG ON <decimal value>
HELP	HELP filePath	OPTION HELP=help_file	HELP <help file path>
HIDESYM	Not supported	Not supported	HIDESYM <symbol list> | @<symbol list file>
IMPORT	IMPORT symbolList | @symbolListFile	IMPORT external_name {,external_name}	IMPORT <symbol list> | @<symbol list file>
INPUT	Supported only by the command line	FILE obj_file{,obj_file}	INPUT <obj list> | @<obj list file>
LINKORDER	LINKORDER symbolList	Not supported	Not supported

Linker Command	CodeWarrior Syntax	Open Watcom Syntax	GNU Syntax
MAP	Supported only by the command line.	OPTION MAP[=map_file]	MAP [map file name]
MESSAGES	MESSAGES filePath	OPTION MESSAGES=msg_file	MESSAGES <file path>
MODULE	MODULE NLMList | @NLMList	MODULE module_name{, module_name}	MODULE <NLM list>
MULTIPLE	MULTIPLE	OPTION MULTILOAD	MULTIPLE
NAMLEN	Not supported	OPTION NAMELEN=value	NAMELEN value
OS_DOMAIN	OS_DOMAIN	OPTION OSDOMAIN	OS_DOMAIN
OUTPUT	Not supported	NAME exe_file	OUTPUT <target file path>
PATH	Not supported	PATH path_name{;path_name}	PATH [search path; . . .]
PSEUDOPREEMPTION	PSEUDOPREEMPTION	OPTION PSEUDOPREEMPTION	PSEUDOPREEMPTION
REENTRANT	REENTRANT	OPTION REENTRANT	REENTRANT
SCREENNAME	SCREENNAME "name"	OPTION SCREENNAME 'name'	SCREENNAME "name"
STACK	STACK stackSize	OPTION STACK=n	STACK <stack size>
STACKSIZE	STACKSIZE stackSize	Not supported	STACKSIZE <stack size>
STAMPEDDATA	Not supported	Not supported	STAMPEDDATA "Stamp" <data file path>
START	START startProcedure	OPTION START=symbol_name	START <procedure name>
SYNCHRONIZE	SYNCHRONIZE	OPTION SYNCHRONIZE	SYNCHRONIZE
THREADNAME	THREADNAME threadName	OPTION THREADNAME 'thread_name'	THREADNAME <name>
TYPE	TYPE typeNumer	FORMAT NOVELL [number]	TYPE <number>
VERBOSE	Not supported	OPTION VERBOSE]	Not supported
VERSION	VERSION majorVersion minorVersion [revision]	OPTION VERSION=major[.minor [.revision]]	VERSION <major version>, <minor version> [, <revision>]
XDCDATA	XDCDATA rpcFilePath	OPTION XDCDATA=rpc_file	XDCDATA <file path>

AUTOUNLOAD

Specifies that the NetWare operating system should unload the NLM when none of its entry points is in use.

CHECK

Specifies the name of a function in the NLM to be executed when the console operator attempts to unload the NLM using the UNLOAD console command.

CODESTART

Specifies an offset to be added to each code symbol offset in the map file. This directive allows a developer to create a map file that compares closely to the values displayed by the NetWare internal debugger.

COPYRIGHT

Provides the copyright string that is displayed on the console screen when the NLM is loaded. If this option is not used, no copyright information is displayed.

The string must be enclosed in double quotation marks (") for CodeWarrior and GNU, but in single quotation marks (') for Open Watcom.

CUSTOM

Allows you to specify the path to a custom data file for use in the NLM. The size and offset of this file are recorded in the NLM header.

DATASTART

Specifies an offset to be added to the data offset in the map file. This directive allows a developer to create a map file that compares closely to the values displayed by the NetWare internal debugger.

DATE

Provides a timestamp for the NLM. The month must be a value between 1 and 12, the day between 1 and 31, and the year 4-digits between 1900 and 3000.

For GNU, the default value is today. GNU uses whitespace (space, tabs, or carriage returns) to separate the month, day, and year parameters. CodeWarrior uses whitespace or commas to separate the parameters.

DEBUG

Instructs the linker to generate debugging information in the executable file. DEBUG affects only files listed in the linker file after this directive.

The NetWare internal debugger uses this debug information. Other debuggers, such as the CodeWarrior debugger, do not use this information.

The Open Watcom linker allows you do specify two options:

```
db_option ::= ONLYEXPORTS | REFERENCED
```

DESCRIPTION

Provides the description that is displayed on the console screen when the NLM is loaded. The string must be enclosed in double quotation marks (") for CodeWarrior and GNU. For Open Watcom, the string is enclosed in single quotation marks (').

The description can be no longer than 127 characters.

EXIT

Specifies the name of a symbol in the NLM where execution should stop. This procedure makes sure that all resources have been released and all threads have terminated before the NLM unloads. For CLIB NLMs, this should be the _Stop routine from the libcpre.o file.

EXPORT

Specifies a list of symbolic names for all variables and functions the NLM is making available to other NLMs. CodeWarrior and GNU allow the EXPORT directive to be followed by either a list of symbols or the name of a file. Open Watcom accepts only a list of symbols.

- When followed by a list for CodeWarrior and GNU, each symbol must be separated by a comma or a whitespace character (tab, space, or carriage return). Each symbol that appears on a new line must be indented by whitespace. Open Watcom expects symbols to be separated by commas.
- When followed by the name of a file, the filename must be preceded by the at (@) character. The file uses the same format as a symbol list.

FLAG OFF

Specifies how the NLM is loaded by clearing the specified bits in the NLM header. For a list of arguments, see “FLAG ON and FLAG OFF Parameters” on page 30.

FLAG ON

Specifies how the NLM is loaded by setting the specified bits in the NLM header. For a list of arguments, see “FLAG ON and FLAG OFF Parameters” on page 30.

HELP

Specifies the path to an internationalized help file that contains the default help screens for the NLM. The path must end with filename with a .HLP extension.

HIDESYM

Specifies a list of symbols to hide.

IMPORT

Specifies a list of symbolic names for variables and functions that other NLMs have defined and your NLM uses. The IMPORT command can be followed by either a list of symbols or the name of a file. CodeWarrior and GNU allow the IMPORT command to be followed by either a list of symbols or the name of a file. Open Watcom accepts only a list of symbols.

- When followed by a list, each symbol must be separated by a comma or a whitespace character (tab, space, or carriage return). Each symbol that appears on a new line must be indented by whitespace. Open Watcom expects symbols to be separated by commas.
- When followed by the name of a file, the filename must be preceded by the at (@) character. The file uses the same format as a symbol list.

INPUT

Specifies the object files that are to be linked. If no file extension is specified, a file extension of either .obj is assumed or the default for the environment. The INPUT command can be followed by either a list of symbols (supported by GNU and Open Watcom) or the name of a file (supported by GNU).

- When followed by a list, each object file must be separated by a comma or a whitespace character (tab, space, or carriage return). Each file that appears on a new line must be indented by whitespace. Open Watcom expects file to be separated by commas.
- When followed by the name of a file, the filename must be preceded by the at (@) character. The file uses the same format as a object file list.

LINKORDER

Specifies the functions and variables that the linker should link first. You do not need to list all the symbols in your program. Each symbol must be separated by a comma or whitespace character (tab, space, or carriage return).

MAP

Indicates that the linker should create a map file with the specified name. If no filename is specified, the name defaults to the name of the executable and a .map extension.

MESSAGES

Specifies the file path to an internationalized message file that contains the default messages for the NLM. The path must end with a filename with a .MSG extension.

MODULE

Specifies the NLMs that must be loaded before this NLM is loaded. These modules are loaded automatically when this NLM is loaded. An NLM that exports symbols that another NLM requires must be loaded before the dependent NLM is loaded.

Each NLM in the list must be separated by a comma or whitespace character (tab, space, or carriage return). Each NLM name that appears on a new line must be indented by whitespace. Open Watcom supports only a comma as a separator.

CodeWarrior also supports listing the modules in a file. When filename follows the MODULE command, the filename must be preceded by the at (@) character. The file uses the same format as a module list

MULTIPLE

Sets a flag in the NLM header indicating that this NLM can be loaded multiple times. If this flag is not set, the NLM cannot be loaded more than once.

NAMLEN

Specifies the maximum characters required to uniquely identify a symbol name. If any symbol fails to meet this condition, the symbol is treated as if it had been defined more than once.

GNU Note: Default is 31. Zero is no limit.

OS_DOMAIN

Sets a flag in the NLM header indicating that this NLM must be loaded in the memory space of the operating system. This prevent the NLM from being loaded into a protected address space.

OUTPUT

Provides the name of the output file for the linker. If no extension is specified, the linker creates an extension according to the NLM type that is specified with the TYPE command.

PATH

Specifies a search path for files used with the directives that take a path and a filename, such as CUSTOM (page 25), EXPORT (page 26), HELP (page 26), IMPORT (page 26), INPUT (page 27), MESSAGES (page 27), and XDCDATA (page 30). The parameter is a string that can be prepended to a filename to create a complete DOS path to the file. Therefore, the sting must end with a backslash (\). The current directory is not searched unless specified. The directories are searched in the order listed. For example:

```
   PATH .\;..\obj\;
or
   PATH ; ..\obj\;
```

Both of these commands search the current directory and the obj directory which is one level up.

PSEUDOPREEMPTION

Sets a flag in the NLM header indicating that the NetWare operating system can forced the NLM to relinquish control if it does not do so on its own often enough.

The amount of time that is allowed to pass before the NLM is forced to relinquish control is set by the console command Set Pseudo Preemption Count. When the time limit is exceeded, the NLM is forced to relinquish control on the next file read or write system call.

REENTRANT

Sets a flag in the flags field of the NLM header indicating that this NLM is reentrant. If an NLM is reentrant, when it is loaded by the LOAD command more than once, the NLM is not loaded again, but the NLM can now be executed concurrently by multiple threads. In this case, only one copy of the NLM is in memory.

SCREENNAME

Specifies the name of the first screen of an NLM, which is created when the NLM is loaded and to which stdin, stdout, and stderr are wired. The screen name is displayed at the top of the console when Alt is pressed, and displayed in the list of current screens when Alt+Esc is pressed.

The screen name must be 71 characters or less.

If this directive is not used, or if NONE is specified, there is no initial screen. In this case, the developer must call CreateScreen (page 237) to create a screen for the NLM. However, stdin, stdout, and stderr are not wired to this screen.

STACK

Specifies the stack size for the NLM in bytes. The minimum stack size is 2 KB. Over 32 KB is recommended. If no size is specified, the default is 16 KB.

NOTE: Many of the communications libraries require that the NLM threads stack size be increased beyond the 16 KB default to 32 KB or more. Consider increasing `stack size` if you encounter stack overflow abends.

This command is equivalent to the STACKSIZE command.

STACKSIZE

Specifies the stack size for the NLM in bytes. The minimum stack size is 2 KB. Over 32 KB is recommended. If no size is specified, the default is 16 KB.

NOTE: Many of the communications libraries require that the NLM threads stack size be increased beyond the 16 KB default to 32 KB or more. Consider increasing `stack size` if you encounter stack overflow abends.

This command is equivalent to the STACK command.

STAMPEDDATA

Causes the linker to create a custom data structure, which is named by the stamp parameter and filled by the data file.

START

Specifies the name of a symbol in the NLM where execution should start. This procedure tracks the state of the NLM and helps with the final cleanup function. For CLIB NLMs, this should always be _Prelude from the libcpre.o file.

SYNCHRONIZE

Sets a flag in the flags field of the NLM header indicating that when this NLM is loaded, the load process goes to sleep until the NLM calls SynchronizeStart (page 163). This prevents other console commands (particularly LOAD) from being processed while the NLM is loading.

THREADNAME

Specifies a prefix for NetWare to use to name the threads of the NLM. For example, if the prefix was Process, then threads created in the NLM would be named “Process1,” “Process2,” “Process3,” and so on. Thread names can be displayed in the CodeWarrior debugger and by using the .P option in the NetWare internal debugger.

The thread prefix should be 12 characters or less.

TYPE

Specifies the kind of service the NLM provides. For a list of supported values, see “NLM Types” on page 31.

VERBOSE

Causes the linker to produce additional information while linking.

VERSION

Specifies the version of the NLM. The version information is displayed on the console screen when the NLM loads. The major version and minor version numbers are required; the revision is optional.

The major version can be any number. The minor version can be 0 - 99. The revision can be 0 - 26, representing a - z. If the revision is greater than 26, it is set to 0.

XDCDATA

Specifies a path to a file containing Remote Procedure Call (RPC) descriptions for functions in the NLM. XDC data can be used to create an MT-safe NLM, funnel functions to processor 0, declare an NLM as MT unsafe, and mark an NLM as preemptible. You must use the mpkxdc tool to generate the XDC data file used by this command. For more information, see Section 6.4, "Using MPKXDC," on page 98.

FLAG ON and FLAG OFF Parameters

The following parameters can be ORed together. Some linkers allow the linker definition file to have multiple FLAG commands. Some of the FLAG parameters are the same as commands, for example: FLAG ON 0x00000001 is equivalent to REENTRANT. If you have duplicate or conflicting commands in the definition file, the last command entered is used to configure the NLM.

Value	Description
0x00000001	Specifies whether an NLM is reentrant. If this flag is set, more than one thread can execute the code of an NLM at the same time. It is equivalent to the REENTRANT linker command.
0x00000002	Specifies whether the NLM can be loaded multiple times. If set, more than one copy of the NLM can be loaded. It is equivalent to the MULTIPLE linker command.
0x00000004	Specifies whether the console command processor sleeps while this NLM loads. If this flag is set, the load process sleeps until the NLM calls the SynchronizeStart function. Besides the load console command, any other console commands are not processed while this NLM loads. This flag is equivalent to the SYNCHRONIZE linker command.
0x00000008	Sets a flag in the NLM header indicating whether the NetWare operating system can forced the NLM to relinquish control if it does not do so on its own often enough. If set, the OS can force the NLM to relinquish control. If not set, the NLM must relinquish control or cause a CPU Hog abend. This flag is equivalent to the PSEUDOPREEMPTION linker command.
0x00000010	Specifies whether the NLM is forced to load into the memory space of the operating system (ring 0). If set, the NLM cannot be loaded into protected address space. This flag is equivalent to the OS_DOMAIN linker command.

Value	Description
0x00000020	Specifies whether the NLM can share code. If set, the NLM cannot share code with any other NLM.
0x00000040	Specifies whether the NLM is automatically unloaded when none of its entry points are in use. This flag is equivalent to the AUTOUNLOAD linker command.
0x00000080	Specifies whether the NLM is hidden. If set, the NLM does not appear in module lists, such as those generated with the MODULES console command????
0x00000100	Specifies whether the NLM is digitally signed.
0x00000200	Specifies whether the NLM is forced to load into protected address space. If set, the NLM cannot be loaded into the memory space of the operating system (ring 0). It will only load in protected address space (ring 3), and no other command line parameters are needed to make it load protected.
0x00000400	Specifies whether the NLM is a shared library.
0x00000800	Specifies whether the NLM is can be restarted.
0x01000000	Specifies whether main can end and the NLM still stay resident in memory.

NLM Types

The following values can be used with the TYPE linker command to specify the type of service the NLM provides. Use one of the following for the number argument:

Value	Description
0	Generic module (the default and the most common designation)
1	LAN driver
2	Disk driver
3	Name space module
4	NLM utility application
5	Mirror server link module
6	OS module
7	Paged high OS module
8	Host adapter module
9	Custom device module
10	File system module
11	Real mode module
12	Ghost module
13	Normal SMP module
14	NIOS module

Value	Description
15	CIOS CAD module
16	CIOS CLS module
20 - 32	NICI (Novell International Cryptographic Infrastructure) modules

1.6 Writing a Basic NLM

The following steps are basic to NLM development, and will produce an NLM that runs on the server. (Code development can become considerably more complex with the increasing complexity of an NLM's functionality.)

1. Set up the development environment you will use to build, run, and debug the NLM:
   - For CodeWarrior, see "Setting Up CodeWarrior for NLM Development" on page 15.
   - For Open Watcom with Borland C++ Builder, see "Setting Up Open Watcom with Borland C++ Builder" on page 17.
   - For Watcom users, see "Setting Up Your Environment to Use WATCOM IDE" on page 18.
2. Create a subdirectory for the new NLM you are creating.
3. Write the code for the NLM.

   For a starting NLM, it may be best to use standard ANSI functions so you can get a feel for the core NLM development process without the complications of the NetWare CLIB server API set. When you are ready to move on to multithreaded NLMs with fuller functionality that use a variety of functions in the CLIB family of server-based APIs, see Section 4.1, "Developing Multithreaded NLMs," on page 71.

   Bear in mind at this point that when you are developing more complex NLMs that allocate resources to CLIB threads and require CLIB context to free those resources, it will be extremely important to write the code required to allow the NLM to terminate gracefully with resources properly freed. To review that process, see Section 4.2, "Terminating an NLM," on page 71.
4. Compile and debug your source code.
5. Compile and debug the source code as an NLM.
6. Copy the NLM onto the server and load it to see how well it runs. For fully functional NLMs, make sure this step is done on a server used for development rather than a production server.
7. Although it is probably not needed with a simple NLM such as this one, time should be allocated at this point for testing. That time will much more than pay for itself.

1.7 Installing the CLib Files on a NetWare Server

Support packs for NetWare 5.1 and NetWare 6 install newer versions of CLib. However, the NDK often has a newer version than the current support pack. If you need to test with this newest version, complete the following steps to install the CLib files on a test server.

WARNING: NDK versions of CLib should never be used in a production environment. You should always use a support pack to install newer versions of CLib into a production environment.

The following instructions assume that you installed CLib SDK in the c:\Novell\ndk\nwsdk directory and that you installed server.exe into the c:\nwserver directory on your NetWare server.

To install the CLib files for NLMs:

1. Back up the following files in the c:\nwserver directory:

clib.nlm	nit.nlm	requestr.nlm
lib0.nlm	nlmlib.nlm	threads.nlm

2. Copy from the following files from the c:\novell\ndk\nwsdk\lib\nlm directory on your workstation to the c:\nwserver directory on your server:

clib.nlm	nit.nlm	requestr.nlm
lib0.nlm	nlmlib.nlm	threads.nlm

3. (Optional) Copy the message files from the c:\novell\ndk\nwsdk\lib\ nlm\msg directory on your workstation to the c:\nwserver\nls\4 directory on your server.

   This is optional, because if you compare file dates, you'll discover these files are seldom updated.

4. Reboot your server.

Basic NLM Concepts

2

This section covers basic NLM concepts such as basic NLM functionality and how to develop an NLM.

2.1 What NLMs Are

NetWare Loadable Modules (NLMs) are the building blocks used to customize the NetWare® operating system. NLMs are built to run in server memory with the NetWare operating system (starting with NetWare 5.0, they can be loaded into either operating system or protected address space).

You can load or unload NLMs to or from server memory while the server is running. Once loaded on a server, an NLM becomes part of the OS. As such, it can directly access NetWare services provided by that server without using a service protocol such as the Novell NCP service. NLMs can also call functions that use NCP services for access to remote servers.

The server functions that NLMs can access are collectively called the NetWare API. A fundamental part of the NetWare API is a core set of APIs that provide a direct programming link into the services of the NetWare operating system.

2.2 What NLMs Do

NLMs add openness, modularity, and flexibility to the NetWare operating system:

- They can be loaded and unloaded as needed.
- They can allocate and deallocate memory as needed.
- They can link with and access the NetWare operating system and other NLMs.

NLMs, such as print and communication servers and server-based utilities, enable NetWare administrators to extend the flexibility and capability of their networks.

2.3 Misconceptions About NLMs

As NLM development has been discussed over time, two misconceptions have worked their way into rumors:

- NLMs are hard to develop.
- NLMs have no future—they are basically dead.

The following information is a more accurate picture for both of these ideas:

2.3.1 NLMs Are Not Hard to Develop

NLMs can be written with nothing but standard ANSI C, although nearly all NLMs are written as extensions of the NetWare operating system and therefore require use of at least parts of the NetWare API sets.

Every development environment has its challenges, however, and NLMs are no exception. The following are the major challenges, each of which can be fairly readily met:

- Learning the NetWare API set—however, mastering the APIs is part of learning to program to any platform.
- Learning to unload an NLM gracefully—a well-tested method for doing that is documented in this NDK.
- Mastering the NLM development environment—compiling, linking, using NetWare dynamic link libraries, etc. This problem is also fairly easy to overcome.

2.3.2 NLMs Are Not Dead

NLMs are the executables for the NetWare environment, and NetWare sales continue to be strong. This means that the need for new NLMs will continue well into the future.

As 64-bit solutions are being developed by all major software vendors including Novell,a substantial market space serviced by NetWare 5.x and 6.x will exist for years to come, and many of those systems will need NLM solutions. The need for NLMs inside a company and the market for NLMs to outside companies is still strong.

2.4 Developing NLMs

NLM development can become quite complex, as is true of all environments. However, as Section 2.3, “Misconceptions About NLMs,” on page 35 explains, it is possible to write a simple NLM using only ANSI standard functions and to have it running on the server within minutes of setting up the development environment. If you haven't written an NLM and would like to produce one quickly, primarily to learn the process, Section 1.6, “Writing a Basic NLM,” on page 32 explains how to do that.

With experience, practice, and communication with other NLM developers, you can develop NLMs that use not only ANSI C functions, but the rich set of APIs that ship with this NDK. Your NLMs can make use of services that do everything from allocating memory to monitoring and reacting to various file system events. When you learn the connection models of NetWare servers and clients, you can use cross-platform as well as server-specific functions. It's even possible now to write DLLs that run on a NetWare server.

2.5 Loading and Unloading NLMs

NLMs can be loaded and unloaded from the server while the server is running. You can load NLMs from the console screen with the NetWare console command LOAD or with the spawnlp and spawnvp functions that are included in the NetWare API. The NetWare API can be passed parameters (and can pass parameters to an NLM) whenever a NetWare API application is loaded.

This section covers the following topics:

- “Loading Multiple NLMs” on page 41
- “Importing and Exporting NLMs” on page 41
- “Unloading NLMs” on page 42

2.5.1 Using Search Paths

Before loading NLMs, you can specify a search path using the NetWare console command SEARCH. For example:

```
SEARCH ADD C:\SERVER
```

To restrict the loading of NLMs, use the SECURE CONSOLE command. This command prevents anyone from loading unauthorized NLMs. After the console is secured, you can load only NLMs that reside in any search path.

Regardless of how an NLM is loaded, there are certain rules that the NetWare operating system follows to find the NLM:

- If you specified an absolute path, then the search path is not used. An absolute path must contain a NetWare volume name or a drive letter. Absolute paths are not allowed if the console is secured.
- If you specified a relative path, it is appended to each of the entries in the search path until the NLM is found or all of the entries have been tried.

The operating system must be able to find the NLMs when a LOAD command is issued. You can load NLMs from a floppy diskette on the server, from the DOS partition of a hard disk on the server, from the server’s SYS:SYSTEM directory, or from other locations.

For example, to load a utility module named TEST.NLM, you can use the LOAD command at the server console in any of the following ways (on NetWare 5.x and 6.x servers, it's not necessary enter the command word "load"):

- To load from the current working directory (CWD) on disk drive A:, enter the following:
  ```
  LOAD A:TEST
  ```
- To load from the CWD on the DOS partition of the hard disk, enter the following:
  ```
  LOAD C:TEST
  ```
- To load from the root directory on the SYS: volume on the server, enter the following:
  ```
  LOAD SYS:TEST
  ```
- To load from a search path (as specified by the SEARCH command) on the server, enter the following:
  ```
  LOAD TEST
  ```

2.5.2 How NLMs Are Loaded

The executable file that first establishes the NetWare operating system is SERVER.EXE. This file contains two parts: a loader and SERVER.NLM. When you start a NetWare server by running SERVER.EXE, the loader is put into memory first. It then loads SERVER.NLM, which contains the NetWare OS kernel.

Both the loader and SERVER.NLM must be in memory before you can load NLMs. The loader knows how to load NLMs, but it cannot do so without allocating memory for them. Because

SERVER.NLM maintains the list of available memory, the loader and SERVER.NLM must work together to load NLMs.

All NLMs consist of a header and the code and data. The header component of an NLM contains the following three lists:

Autoload list

Includes the prerequisite modules that must be loaded before a given NLM can function.

Import list

Includes the names of services and data that the NLM needs to use in order to function.

Export list

Includes the names of services and data that the NLM provides for use by other modules.

When you load an NLM with the LOAD command, the loader first processes the NLM header, which includes the autoload, import, and export lists. Then the loader loads the actual code and data of the NLM.

As NetWare loads the NLM into the server's memory, it resolves all unresolved externals and initializes the module. The basic steps in loading an NLM are as outlined here:

1. The loader processes the NLM header.
2. The loader loads the code and data of the NLM by requesting memory from SERVER.NLM and loading the code and data at the memory addresses allocated. (If the NLM has been compressed with NLMPACK, the loader unpacks the NLM as it is loaded.)
3. The loader maintains a master list of available services and their addresses. (The loader builds this list by processing the export lists of NLMs as it loads them.) The loader resolves the names in the imported lists of the NLM (that is, substitutes the service's address in memory for its name throughout the NLM code). This process allows NLMs to call services directly by calling their code.

2.5.3 Using the LOAD Command

The NetWare API can be passed parameters whenever a NetWare API application is loaded. The LOAD command can include:

- Command line parameters that are passed to the NLM
- NetWare API runtime parameters

The syntax for the LOAD command is as follows:

```
LOAD loadable-module-name [parameter1 parameter2 ... ]
[(CLIB_OPT)/CLIB-parameter1/CLIB-parameter2 ...]
```

The following summarizes parameters used with the LOAD command.

`loadable-module-name`

(Required) The name of the module to load can be specified with or without a filename extension.

The path information precedes the loadable module name. If you specify an absolute path, then it must begin with a DOS drive letter or a NetWare volume name. If you do not specify path

information, then the server assumes that the loadable module is located in the server's search path.

`parameter1 parameter2 ...`

(Optional) You can pass one or more parameters to the module.

WARNING: Once you pass a redirection command (such as <SYS:/file.ext), you cannot pass any other parameters. All parameters must precede the redirection command.

(CLIB_OPT)/ `CLIB- parameter1 / CLIB-parameter2 ...`

(Optional)You can specify one or more NetWare API runtime parameters (see below).

NOTE: The (CLIB_OPT) parameters must not have any embedded blanks.

CLIB_OPT parameters include the following:

/Pcwd

Specifies the initial CWD. This allows you to specify an initial CWD other than the root for this execution of the NLM.

```
LOAD TESTER (CLIB_OPT)/PVOL1:PDEMO
```

/> `filepath`

Redirects the second-level output, `stdout`, to the specified file path.

The following example executes an NLM called BINDDUMP whose second-level output is to be written to SYS:TEMP\BINDERY.LST.

```
LOAD BINDDUMP (CLIB_OPT)/>SYS:TEMP\BINDERY.LST
```

When specifying paths, use "\" as the path delimiter because "/" is interpreted as a new option. (You can also use redirection in ways similar to how it is done in DOS and UNIX.)

WARNING: Once you pass a redirection command, you cannot pass any other parameters. All parameters must precede the redirection command.

/< `filepath`

Redirects the second-level input, `stdin`, from the specified file path as in this example:

```
LOAD TESTER (CLIB_OPT)/<SYS:TEST\TEST1.SCR
```

When specifying paths, use "\" as the path delimiter because "/" is interpreted as a new option.

WARNING: Once you pass a redirection command, you cannot pass any other parameters. All parameters must precede the redirection command.

See Stream I/O Concepts (Single and Intra-File Services) for more information about second-level I/O. (You can also you redirection in ways similar to how it is done in DOS and UNIX.)

/E `filepath`

Redirects the second-level error stream, `stderr`, to the specified path. (You can also you redirection in ways similar to how it is done in DOS and UNIX.)

/L

There are two /L parameters that can be used when loading an NLM. The command

```
LOAD /L loadable_module
```

loads your NLM code into the data segment. This option is used when the compiler places character constants in your code segment.

The second /L parameter is as follows:

```
LOAD loadable_module [optional parameters] (CLIB_OPT)/L<number>
```

where `number` is the packet signature level as specified by the NCP Packet Signature security enhancement, which protects servers and clients using the NCP by preventing packet forgery. Packet signature level options are:

0-CLIB does not sign packets

1-CLIB signs packets only if the server requests it (server option is 2 or higher)

2-CLIB signs packets if the server is capable of signing (server option is 1 or higher)

3-CLIB signs packets and requires the server to sign packets (or logging in will fail).

The default NCP packet signature level is 1 for clients and 2 for servers. The default level for CLIB is 1.

To change the default packet signature level for all NLMs that use CLIB, use the following command format when you load CLIB:

```
LOAD CLIB/L<number>
```

where `number` is the signature level.

To change the packet signature level for a single NLM, use the following command format when you load the NLM:

```
LOAD loadable_module [optional parameters] (CLIB_OPT)/L<number>
```

/N

Specifies the initial name space. The optional name space specifications are as follows:

```
/N[DOS|MAC|UNIX|FTAM|OS2|LONG]
```

(Internally OS2 and LONG are defined as the same value.)

A load command with the initial name space specified might appear thus:

```
LOAD TESTNLM (CLIB_OPT)/NLONG
```

/S

Specifies a remote server login, as shown here:

```
/S servername\userID [\password]
```

/Y

This option is used when you receive the console error message "An NLM has been loaded that does not allow low priority threads to be run. Set `Upgrade Low Priority Threads' to ON or unload the NLM." This condition occurs when an NLM that does busy waiting or spin locks with ThreadSwitch (instead of with ThreadSwitchWithDelay) is loaded on a NetWare server.

For NLMs that were developed using the NetWare API, you should use the `/Y` option if you receive the above error message. This option causes the NLM to use the handicapped CPU-yielding functions.

See Relinquishing Control (*NDK: NLM Threads Management*) for information on how to write NLMs that do not create this problem.

Multiple options can be used together as show in the following example:

```
LOAD TESTER (CLIB_OPT)/>SYS:OUT/PSYS:SYSTEM
```

2.5.4 Setting Environment Variables

You can set environment variables from the command line by typing the "setenv" command, followed by the name of your variable and an equals sign (=), followed by the value for the variable.

The following example sets a path environment variable:

```
setenv PATH = C:\windows\
```

To clear an environment variable, type the "setenv" command, followed by the name of your variable and an equals sign (=), followed by nothing, as in the following example:

```
setenv PATH =
```

2.5.5 Autoloading Prerequisite NLMs

You can specify more than one prerequisite NLM by including the MODULE command in a linker definition file. You can separate the NLM names with commas or use multiple MODULE commands on separate lines.

If different NLMs can satisfy the same requirements, you can specify those modules by separating their names with a vertical bar (|). The OS searches for these modules in the order specified, loading the first one that it finds.

The list of NLMs that need to be autoloaded are specified with the linker command MODULE as follows:

```
MODULE module_name1, module_name2,..
```

`module_name1` and `module_name2` are the filenames of the NLMs to be autoloaded.

An NLM fails to load if the loader cannot find any of the modules listed in the autoload list of the NLM, or if there is not enough memory to load them.

2.5.6 Loading Multiple NLMs

The NetWare API allows multiple NLMs to run simultaneously. Most NLMs rely on services that other modules provide. This interdependence is why many NLMs must be loaded in a certain order. If an NLM requires services provided by other modules, it must be loaded last. For example, STREAMS.NLM must be loaded before loading CLIB.NLM as follows:

```
LOAD STREAMS
LOAD CLIB
```

Or, because CLIB.NLM autoloads STREAMS.NLM, you could simply enter the following:

```
LOAD CLIB
```

If you try to load an NLM that depends on the services of another NLM that is not present, unless the dependent NLM autoloads the provider NLM, the dependent NLM will not load.

2.5.7 Importing and Exporting NLMs

After processing the autoload list, the loader processes the import list and then the export list. The loader checks the import list of the NLM, name by name, to verify that it can resolve the name of each service. An NLM fails to load if the loader cannot resolve one or more of the names in the import list. (Any names that cannot be resolved are displayed on the system console screen.)

Once the import list of the NLM has been processed successfully, the loader processes the export list (the list of services that the NLM provides). The loader adds each name in the export list to its master table of available services.

2.5.8 Unloading NLMs

NLMs can be unloaded while the server is running. When an NLM is unloaded, it must return all the memory and resources that the server previously allocated to it. The operating system generates a warning for each resource that has not been explicitly released.

You can unload NLMs manually by means of the console command UNLOAD entered at the command line, or they can unload themselves. When an NLM exits by means of UNLOAD, a signal is sent to the signal SIGTERM handler if one is registered (it should be for nearly all NLMs). The signal handler, if properly written, directs all threads in the NLM to free all resources they hold and to terminate themselves, with the exception of the main thread, which returns. A function can be registered with AtUnload to be called, but all resources should have been freed with the signal handler first. The functions registered with atexit can be called as well.

NOTE: A detailed explanation of the unload process is presented in "NLM Unload Process" on page 51.

You can use the UNLOAD command to unload programs. The syntax for the UNLOAD command is as follows:

```
UNLOAD loadable-module-name
```

where `loadable-module-name` is the name of the NLM you want to unload.

2.6 Introduction to CLIB

CLIB is a collection of libraries that provide core server-side functions.

These libraries load automatically when the server is started. You can also load them manually by typing load clib at the command prompt (simply clib for NetWare 5.x or 6.x servers).

When CLIB.NLM is loaded, its dependency NLM libraries, as listed below, also load automatically unless they are already loaded:

THREADS.NLM—Support for NetWare threads

REQUESTR.NLM—NetWare Requester support

NLMLIB.NLM—POSIX and NetWare support

NIT.NLM—Various NetWare server functions, many of which are also offered in cross-platform libraries through CAL32NLM.NLM.

These modules collectively contain NetWare server API functions. Your NLMs can call these functions to perform a wide variety of services, including the following:

- Network security, management, and accounting
- High- and low-level I/O
- String and file manipulations
- Memory allocation
- Thread control, synchronization, and communication

- Data conversion
- Mathematical calculations
- Screen management

In addition, the library contains a set of Advanced Services that include functions for dynamic arrays, event reporting and management, extended attributes, complex memory allocation, and console command registration.

2.6.1 Cross Platform NLM Libraries

In addition to the CLib modules, the CLib SDK also includes cross platform (XPlat) libraries for NLM development. They are called XPlat because these libraries also are available for Windows client development. These library modules include the following:

AUDNLM32.NLM
CALNLM32.NLM
CLNNLM32.NLM
CLXNLM32.NLM
DSAPI.NLM
DSEVENT.NLM
LOCNLM32.NLM
NCPNLM32.NLM
NETNLM32.NLM
UNICODE.NLM

2.6.2 Prelude Object Files

The CLib imports directory contains the following prelude files.

File	Compiler	Description
clibpre.gcc.o	GNU C/C++	Prelude object file
clibpre.o	Metrowerks CodeWarrior	Prelude object file
clibpre.obj	Watcom	Newest version of the prelude object file. NLMs linked with this file and loaded on versions of NetWare prior to NetWare 5.1 SP4 (including NetWare 4.x) must also be linked to a vendor-specific runtime library.
nwpre.obj	Watcom	Older version of the prelude object file.
prelude.obj	Watcom	Oldest version of the prelude object file. For new NLM developement, do not link to this file.

Basically, clibpre.obj is nwpre.obj with a call to the following functions that vendor runtime libraries implement to perform their C++ initialization and clean-up.

```
int __init_environment( void *reserved )
{
   return 0;
}
```

```
int __deinit_environment( void *reserved )
{
   return 0;
}
```

2.6.3 CLIB Manuals

Since the CLIB libraries allow for development of a cross-platform NLM, cross-platform Windows application, and server-only NLM development, the information about these libraries has been divided into the following manuals:

Manual	Cross-Platform	NLM-Only
NLM Development Concepts, Tools, and Functions	None.	NetWare operating system functions for events, resource tags, symbols, screen handling, debugging, settable parameters, etc.
Getting Started with NetWare Cross-Platform Libraries for C	Information about using the Cross-Platform libraries for client and server development.	None.
NLM and NetWare C Program Management	A few date/time and data manipulation functions.	Bit, character type, data manipulation, library, math, memory, string, and time functions.
NLM Threads Management	None.	Thread, semaphore, and process (such as abort and system) functions.
Connection, Message, and NCP Extension	Connection, message, and NCP extension functions.	Connection and message functions.
Multiple and Inter-File Management	Data migration, deleted file, file system manipulation, file system monitoring, name space, path and drive functions.	File engine (FE*) I/O functions, file system manipulation functions, name space functions, path and drive functions, and NIT functions for data migration and file system manipulation.
Single and Intra-File Management	AFP, file locking, semaphores, and extended attribute functions.	Direct file system functions, DOS partition functions, file I/O functions, stream I/O functions, and NIT functions for file locking, semaphores, and extended attributes.
Volume Management	Volume information functions for name, number, statistics, and restrictions.	Volume information functions for name, number, and statistics.
Client Management	Requester version and NWCalls management functions.	None.

Manual	Cross-Platform	NLM-Only
Network Management	Accounting functions, auditing functions, and name retrieval functions (for servers and trees).	Accounting and auditing functions.
Server Management	Server information functions including statistics, console privileges, version. Management functions for logins, set commands, NCF files, loading/unloading NLMs, mounting/dismounting volumes, and TTS.	NIT functions for server information and management. Most NIT functions have an equivalent Cross-Platform function.
Internationalization	Locale and multi-byte functions.	Locale functions.
Unicode	Unicode functions.	None.
Bindery	Bindery functions	NIT bindery functions
Sample Code	Examples of creating CLIB NLMs, with only a few cross-platform examples. Many of these examples are no longer available in the ...\ndk\samples\clib_sample directory. To see what is available as sample code and to access more cross-platform examples, see NLM and NetWare Libraries for C (http://developer.novell.com/ndk/doc/samplecode/clib_sample/index.htm).	

Even though the software for the Novell eDirectory functions is included with the CLIB SDK, the documentation for these functions is in the eDirectory Libraries for C (http://developer.novell.com/ndk/ndslib.htm).

2.7 OS-Related Issues

Some services are available to the programmer because of the design of the OS. The following topics discuss the OS features that are significant for NLM developers.

2.7.1 Preemptive and Nonpreemptive Environment

The NetWare operating system was originally designed without time slicing (preemption). All NLMs had to relinquish control of the CPU by blocking or by explicitly yielding control. Beginning with NetWare 5, the scheduler performs preemption, but only for NLMs that are linked with XDC data. NLMs not linked for preemption are required to relinquish control to the operating system.

For NetWare 5 and later, you can design your application to run with or without preemption. If you select nonpreemption, your threads will run until they knowingly call a function that blocks (relinquishes control to the operating system). Because the operating system waits for threads to block, nonpreemptive NLMs are expected to govern their use of the CPU time so as not to take control of the CPU for indefinite periods of time.

NLM applications must either quickly complete the request, do things to regularly relinquish control (such as I/O requests), or explicitly relinquish control by calling a function such as ThreadSwitch. In general, the NLM should run for about .1 millisecond on a 1.6 GHZ processor and then relinquish control.

If you select preemption, you must still design your application so that it relinquishes control. You cannot completely rely on XDC data and the preemption process to prevent your application from taking control of the CPU for too long of an interval and thus precipitating a CPU hog abend.

2.7.2 Current Working Directory

In the NLM environment, each thread group has its own current working directory (CWD), as well as a current working volume and a current server ID. However, there is no notion of "drive" in this environment when you are referring to the NetWare file system.

CWDs for NLMs can be used by almost all NetWare API functions that take a pathname as an input parameter. Any time a server and volume are specified in a pathname, the pathname is absolute. On the other hand, if the pathname does not contain a server or volume, the path is considered relative to the CWD.

2.7.3 Connection Numbers and Task Numbers

Connection and task numbers are important NLM programming considerations. For information see Connection Numbers and Task Numbers in Connection Number and Task Management Concepts (*NDK: Connection, Message, and NCP Extensions*).

2.7.4 Screens and the NetWare OS

The main screen for the server is the console screen. This screen allows the operator to issue commands directly to the OS. This is also the screen where NLMs are loaded.

NLMs can have zero, one, or multiple screens. These screens display on the same monitor as the console screen. The OS switches between screens when the following happens:

The operator presses Ctrl+Esc

This displays a menu of the names of the currently open screens. The operator then enters the number of the screen to switch to.

The operator presses Alt+Esc

This switches the current screen to the next screen in the list of open screens. This allows the operator to cycle quickly between screens.

A running thread changes the active screen

This is done when a thread calls DisplayScreen (page 242).

The NLM that owns the current screen terminates

When an NLM terminates, its screens are destroyed and the console screen becomes the current screen.

NOTE: Keystrokes are accepted only for the currently displayed screen. An NLM that is waiting for input does not receive it until the operator switches to its screen and enters the needed keystrokes.

For an example of creating and using multiple screens, see Multithreaded Programming (*NDK: NLM Threads Management*).

2.7.5 Screen Types

NetWare 5.x and 6.x servers have the following types of screens:

System Console Screen

Server console commands are entered at the command line of the System Console Screen. This screen is always present. On NetWare 5.x servers, NLMs can write to this screen and receive input from its keyboard. On NetWare 6.x servers, NLMs cannot write to this screen or receive input for it. They can write only to the System Logger Screen or to their application-owned screen.

System Logger Screen

This screen is only present on NetWare 6.x servers. This screen logs all system messages as well as the output from NLMs that write to the system console. NLMs cannot get characters from this screen's keyboard because the screen accepts only a few commands related to scrolling and other such activities.

Debug Screen

The Debug Screen is accessed from within an assembly or C program or through a special key sequence. This screen is hidden unless the server is at a breakpoint.

Router Screen

This screen displays whenever the TRACK ON console command is executed.

Application Screens

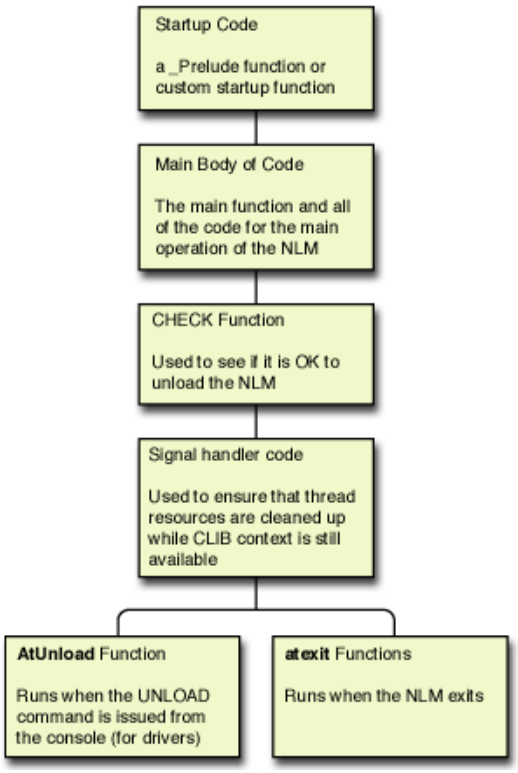
An NLM can have zero or more application screens which are regular or popup. Popup screens, used to present instructional or error messages, are overlaid on regular screens. In some cases, an NLM might not require a screen (a library NLM, for example). An NLM might also write to the System Console Screen (NetWare 5.x), System Logger Screen (NetWare 6.x), or to the screen of another NLM (if the other NLM cooperates).

On NetWare 6.x servers, NLMs can receive input only from an application-owned screen. On NetWare 5.x servers, NLMs can receive input from the System Console Screen or from an application-owned screen.

2.8 Structure of an NLM

An NLM must have initialization code, a main body of code, and termination code. The following figure shows the structure of an NLM.

Figure 2-1 *NLM Structure*



2.9 NLM Startup

When the NLM first loads, an initialization function (_Prelude) performs the following tasks:

- Establishes the context of the NLM for the NetWare API.
- Establishes a default thread group
- Creates a new thread, belonging to the default thread group, and starts the thread executing at the main function.

_Prelude is part of the prelude object file. If the environment is set up correctly, linking in the object file and calling the _Prelude function are both automatic and transparent. For more information about prelude object files, see “Prelude Object Files” on page 43.

2.9.1 Reentrant NLMs

A reentrant NLM can be loaded multiple times, but the server keeps only a single image of the NLM code in memory, rather than a code instance for each load.

Nonreentrant NLMs call the startup function _Prelude each time they are loaded. Reentrant NLMs, on the other hand, call _Prelude only on their initial load. They do not call _Prelude on reentrant loads.

To write a reentrant NLM, create a startup function that checks to see if the NLM has previously been loaded. On the initial load of the NLM, have your startup routine call _Prelude, passing _Prelude the parameters that the OS passed into your startup function. (_Prelude calls the main

function of your NLM.) On subsequent loads of the NLM, do not have your startup routine call _Prelude ; instead, have it handle the reentrant setup and then call the main function of the NLM itself.

In the following sample, the startup function is called MultipleLoadFilter. This function uses a flag called gAlreadyLoaded to indicate whether the current load of the NLM is the first load or a subsequent load.

Reentrant NLM

```
typedef struct resource_list
{
   struct resource_list   *next;
   int                    screenHandle;
} ResourceList;

int            gAlreadyLoaded = 0;
int            gMainThreadGroupID;
ResourceList   *gResList = (ResourceList *) NULL;
typedef void (*PVF) ( void *);

LONG MultipleLoadFilter (
   LoadDefStructPtr   NLMHandle,
   ScreenStructPtr    initErrorScreenID,
   BYTE               *cmdLineP,
   BYTE               *loadDirPath,
   LONG               uninitDataLen,
   LONG               NLMFileHandle,
   LONG               cdecl (*readFunc)())
{
   int   myThreadGroupID;
   if (!gAlreadyLoaded) /* first time through!!!!! */
      return _Prelude(NLMHandle, initErrorScreenID, cmdLineP,
      loadDirPath,uninitDataLen, NLMFileHandle, readFunc);
   /* subsequent times through...*/
   myThreadGroupID = SetThreadGroupID(gMainThreadGroupID);
   BeginThreadGroup((PVF) main, NULL, NULL, cmdLineP);
   SetThreadGroupID(myThreadGroupID);
   return 0L;
}

void main(int argc,char *argv[])
{
   int myThreadGroupID;
   ...
   char **argV;
   if (!gAlreadyLoaded)
   {
      gMainThreadGroupID = GetThreadGroupID();
      RenameThread(gMainThreadGroupID, "Sample-main");
      gAlreadyLoaded = 1;
      firstTime      = TRUE;
      argV           = argv;
      AtUnload(Cleanup);
```

```
    }
    else
    {
       char threadName[17+1+13];
       sprintf(threadName, "Sample-#%d", gAlreadyLoaded);
       myThreadGroupID = GetThreadGroupID();
       RenameThread(myThreadGroupID, threadName);
       gAlreadyLoaded++;
       firstTime = FALSE;
       argV      = args;
    }
    scrH = CreateScreen("Sample Reentrant NLM", 0);
    if (!scrH)
    {
       ConsolePrintf("\nUnable to create screen...");
       goto NoScreenExit;
    }
    LogScreenHandle(scrH);
    SetCurrentScreen(scrH);
    printf("\nSample Reentrant NLM: %d\n", gAlreadyLoaded);
...
}
```

Your startup function must return zero. If it does not, the operating system displays the message "Attempt to reinitialize reentrant module FAILED" even if the NLM successfully loads.

When an NLM is loaded, its startup thread is an operating system thread, which usually doesn't have CLIB context until _Prelude is called. In the example code above, the startup function MultipleLoadFilter calls _Prelude the first time the NLM is loaded, and _Prelude gives the thread CLIB context and creates a default thread group ID. In the example, main saves the default thread group ID in gMainThreadGroupID the first time the NLM is loaded. On subsequent loads of the NLM, MultipleLoadFilter gives the operating system thread CLIB context by setting the thread group ID using the ID stored in gMainThreadGroupID.

You specify that an NLM is reentrant when you link its object modules. In the linker definition file, use the REENTRANT option to specify that the NLM is reentrant. Use the START option to specify the function you want to use as the startup function. The following is an example of a WLINK definition file:

```
form novell nlm 'Reentrant NLM'
name rentrant
option   reentrant
option   start = MultipleLoadFilter
option   case,nodefaultlibs
file     prelude
file     rentrant
import   @clib.imp
```

The REENTRANT option specifies that the NLM is reentrant. The START option specifies the name of the function to call when reentering the NLM. Directive files are discussed in "Specifying a Linker Definition File" on page 22.

For each instance that you use a reentrant NLM, you must load the NLM with the LOAD command. However, every instance of the reentrant NLM is unloaded with a single UNLOAD command. For this reason you must keep track of all of the resources that are used by all instances of the NLM and

free all of them when the NLM is unloaded. (RENTRANT.C shows this by keeping a list of screens.)

2.10 NLM Termination

Using functions from the NetWare API, an NLM can control the termination process. An NLM should include mechanisms to handle the following:

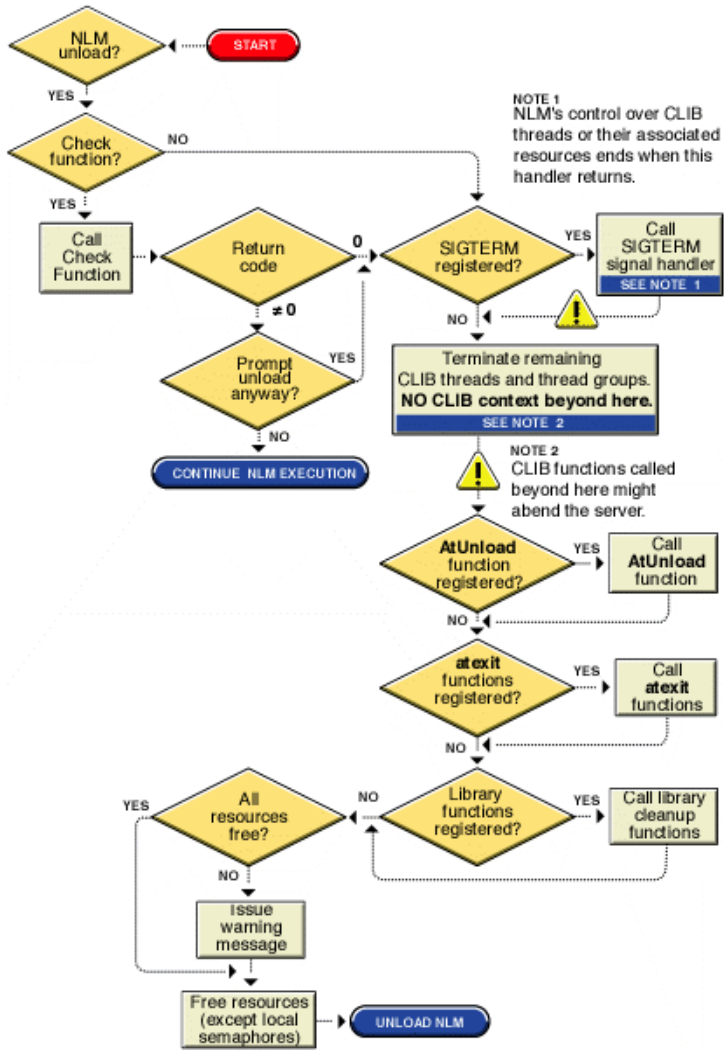
- Unload, such as when the NLM is unloaded at the system console command line, before the NLM exits on its own.
- Self-termination, such as when the NLM exits on its own.
- Abnormal exit, such as when an error causes unexpected shutdown, such as when abort and raise are called.

For information about terminating an NLM gracefully see Section 4.2, “Terminating an NLM,” on page 71.

2.10.1 NLM Unload Process

The NLM unload process, as shown in the following figure, occurs only when an NLM is unloaded from the system console command line. The NLM unload process is a sequence of steps, some performed by the NLM, and others performed by the NetWare OS or by the NetWare API.

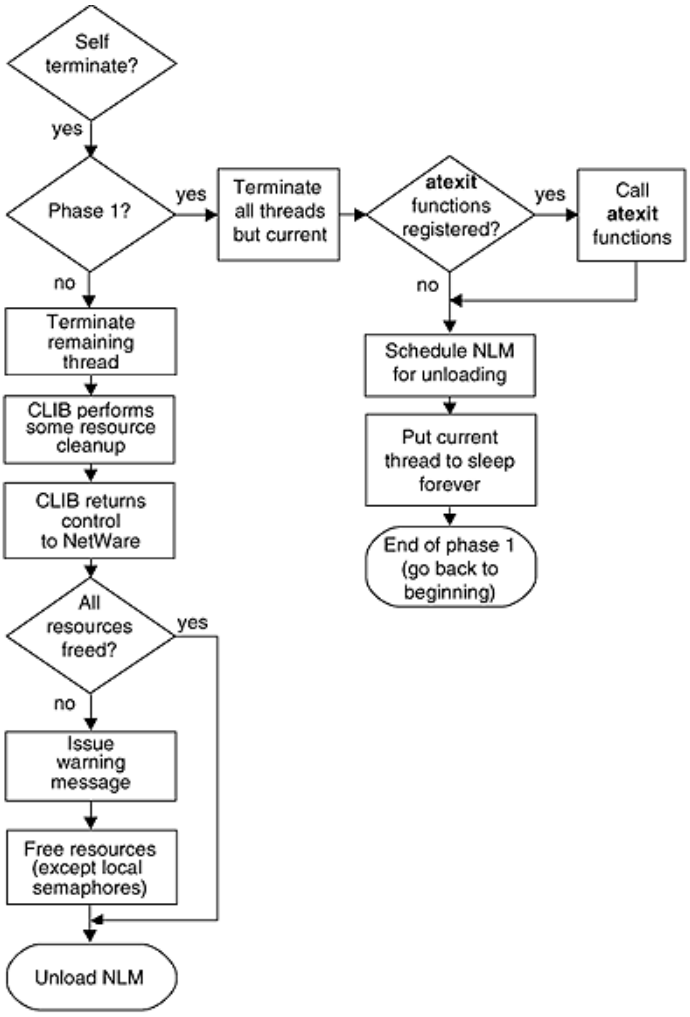
Figure 2-2 *NLM Unload Process*



2.10.2 NLM Self-Termination Process

The NLM self-termination process, as shown in the following figure, occurs when an NLM calls exit or ExitThread or when all threads in an NLM terminate. The NLM self-termination process consists of two phases, with the first phase performed by the thread that caused the termination and the second phase performed by an OS thread.

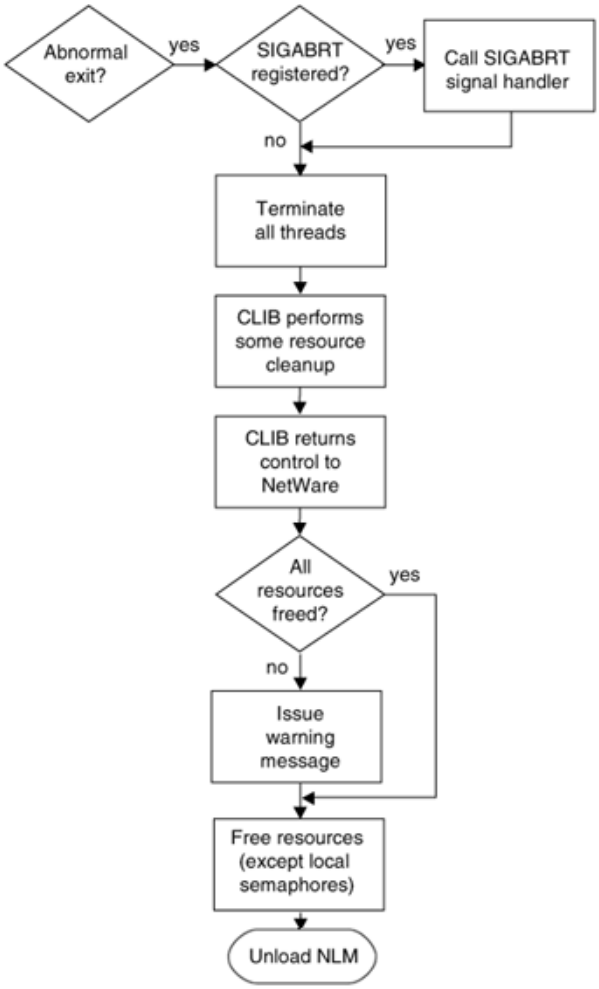
Figure 2-3 *NLM Self-Termination Process*



2.10.3 NLM Abnormal Exit Process

The NLM abnormal exit process, as shown in the following figure, occurs only when the NLM calls the _exit or abort function. The NLM abnormal exit process is a sequence of steps, some performed by the NLM, and others performed by the NetWare OS or by the NetWare API.

Figure 2-4 *NLM Abnormal Abort Process*



2.10.4 Following Exit Steps

The registered function calls and signal raising shown in the figures that show exit steps occur only if the NLM has specified them, using signal, AtUnload, and atexit (*NDK: NLM Threads Management*). If no registered functions or signal handling exists in the NLM, the associated step is skipped.

If a signal handler is implemented properly, use of AtUnload and atexit might not be needed at all (see Section 4.2, “Terminating an NLM,” on page 71).

2.10.5 CHECK Function

In the first step of the unload process, the NLM calls the CHECK function, but only if it has been specified. NLMs specify CHECK functions in the linking phase using directives. For example, if CheckFunction were the name of the function to be called as the CHECK function, the WLINK directive would be as follows:

```
option check = CheckFunction
```

The purpose of the CHECK function is to determine if the NLM is in a state in which it can unload safely. If so, the function should return zero, indicating that the unload process can continue normally.

If the function determines that the NLM cannot be unloaded safely, it should display a warning message on the system console screen and return a nonzero value. If the NetWare operating system receives a nonzero return value from the CHECK function, it issues the following message:

```
Unload module anyway? n
```

If the above message appears, the system console operator can abort the termination process and allow the NLM to continue its normal operation.

NLM termination code should not be placed in the CHECK function.

IMPORTANT: The CHECK function is run by an operating system thread that by default does not have CLIB context in any NetWare version. If your CHECK function calls any NetWare API functions that need CLIB context, you must give the calling thread CLIB context by calling SetThreadGroupID (*NDK: NLM Threads Management*).

The following is a sample CHECK function:

```
int CheckFunction()
{
   /* If you need context information, put it here */
   if( NLMIsBusyRightNow )
   {
      ConsolePrintf(
      "That NLM is currently in use.\r\n");
      return 1;
   }
   return 0;
}
```

Given the sample CHECK function above, if the operator attempted to unload HELLO.NLM while it is busy, the following would be the command and output on the console:

```
:unload hello
That NLM is currently in use.
Unload module anyway? n
:
```

This method does not prevent an operator from continuing with the unload process. It merely provides a meaningful message that the operator can use in deciding whether to continue the unload.

In most situations, you would want to allow the NLM to be unloaded. However, there might be a reason that you do not want anyone to unload the NLM without shutting the server down. In this case, you need to ungetch (page 215) an `n' to the system console. This can be done with the following code:

Causing the Server to Shut Down When an NLM is Unloaded

```
#include <stdio.h>
#include <conio.h>
#include <process.h>

int sysThreadGroupID
```

```
main()
{
   sysThreadGroupID = GetThreadGroupID();
   ...
}

int NWNoUnload()
{
   LONG OldScrID;
   LONG NewScrID;
   int TGID;

   // give the OS thread CLIB context
   TGID = SetThreadGroupID(sysThreadGroupID);

   OldScrID = GetCurrentSceen
   NewScrID = CreateScreen("System Console",0);
   If(OldScrID != NewScrID)
      SetCurrentScreen(NewScrId);
   ungetch('n');
   if(OldScrID != NewScrID)
   {
      SetCurrentScreen(OldScrID);
      DestroyScreen(NewScrID);
   }

   SetThreadGroupID(TGID)
   return -1;
}
```

You can use SetNLMDontUnloadFlag to set an NLM so it cannot be unloaded even if the console operator says it is OK to unload the NLM. Use ClearNLMDontUnloadFlag to allow the NLM to be unloaded after its don't unload flag has been set (*NDK: NLM Threads Management*).

2.10.6 Signal Handling

In general, a signal is raised by the NetWare API when a particular program condition occurs. However, the NLM itself can raise a signal by calling raise (*NDK: NLM Threads Management*).

Signal handling allows previously registered functions to gain control of critical shutdown processes. The following are typical signals your NLM might handle:

SIGTERM

This ANSI standard signal is the most frequently used signal, and is raised when the NLM is unloaded from the console command line. Because the NetWare API raises the SIGTERM signal only when the NLM is unloaded, you must raise the signal yourself when exiting with ExitThread or exit. You can raise the signal using raise (*NDK: NLM Threads Management*).

SIGABRT

This signal is raised only during abnormal exit of the NLM, such as stack overflow. In abnormal exit, the NLM calls abort, which raises the SIGABRT signal (*NDK: NLM Threads Management*).

SIGINT

This signal is raised when the operator presses Ctrl+C during screen output and if the NLM screen's CtrlCharCheckMode is set to TRUE (default). This does not affect an NLM if the current screen is the System Console screen.

In all cases of the termination process, threads are ended before the functions registered with AtUnload and atexit are called. Without signal handling, resources allocated on a local stack, such as local semiphores, cannot be released because thread stacks are freed before the functions registered with AtUnload and atexit are called. (If you want to keep track of these resources after the treads are terminated, you could store them in global or static variables.)

NLMs usually define signal handlers during initialization by calling a function such as the following:

```
signal( SIGTERM,MySignalHandler );
```

The following is a sample signal handler:

NLM Signal Handler

```
int ThreadCounter;/* counts the number of threads running */
int ShutDownFlag; /* the signal handler sets this to TRUE */
#pragma off(unreferenced);
void MySignalHandler(int sigtype) /* sigtype is SIGTERM */
#pragma on(unreferenced);
{
   ShutDownFlag = TRUE;
   printf("Inside signal handler.
      Waiting for threads to stop ...\n");
   while( ThreadCounter > 0 )
   {
      delay( 500 );  /* wait half a second */
   }
   printf("Inside signal handler. Threads have
      stopped.\n");
}
```

By writing a signal handler function that defines a global flag, you can manage resources that are allocated from a local stack. When the signal is raised, it can set a global flag that each thread in the NLM reads. Those threads then can return their own resources and exit.

Whether you require the use of a signal handler depends primarily on whether you have local resources that can be freed only by the thread of execution that allocated them. If your NLM uses any stack-based resources, a signal handler is necessary for proper NLM shutdown.

2.10.7 AtUnload and atexit Functions

IMPORTANT: By the time the AtUnload and atexit functions are called, all NLM thread groups have been destroyed. These threads therefore cannot be given thread group contex. You cannot use any of the NetWare API functions that need context (for example, printf in Single and Intra-File Services) while in the AtUnload and atexit routines. If you do, the server abends.

To ensure that resources owned by NLM threads are properly freed, we highly recommend that you implement a SIGTERM signal handler as explained in Section 4.2, “Terminating an NLM,” on page 71.

During unloading, the AtUnload and the ANSI standard atexit functions are executed if they have been defined. During self-termination, only the atexit functions are executed if they have been defined.

The AtUnload and atexit functions can perform resource cleanup such as freeing memory, closing semaphores, and so on. Each NLM can have a single AtUnload function and up to 32 atexit functions.

NLMs can define the AtUnload function with a call such as the following:

```
AtUnload( NLMUnloadFunction );
```

The following example uses the AtUnload function:

```
char *myMemPtr; /* the pointer for this NLM's memory */

main()
{
   AtUnload( NLMUnloadFunction );
   /* other NLM code would go here */
   printf("You may unload this NLM.\n");
}
void NLMUnloadFunction()
{
   if( myMemPtr != NULL ) free ( myMemPtr );
}
```

The AtUnload function calls one function only. Therefore, the called function should perform all the necessary functions you want implemented at unload time.

NLMs usually specify their atexit functions with calls such as the following:

```
atexit( CloseMyFile );
atexit( CloseMySemaphore );
atexit( FreeMyMemory );
```

Successive calls to the atexit function create a list of functions to be executed on a last-in, first-out basis.

The following example uses the atexit function:

```
FILE *myOpenFile; /* the file this NLM will open     */
LONG mySemaphore; /* the semaphore this NLM will use */
char *myMemPtr;   /* memory this NLM will allocate   */

main()
{
   atexit( CloseMyFile );
   atexit( CloseMySemaphore );
   atexit( FreeMyMemory );
   /* other NLM code would go here */
   printf("You may unload this NLM.\n");
}
```

```
void CloseMyFile()
{
   // still have NLM level context
   if( myOpenFile != NULL)
      fclose ( myOpenFile );
}

void CloseMySemaphore()
{
   if( mySemaphore != NULL )
      CloseLocalSemaphore( mySemaphore );
}

void FreeMyMemory()
{
   if( myMemPtr != NULL )
      free( myMemPtr );
}
```

If you do not handle cleanup through signal handling, you can use the atexit functions to clean up if any of the following conditions are met:

- The NLM calls exit.
- The last thread in the NLM returns from its original function.
- The NLM calls ExitThread with EXIT_NLM as the action code parameter, which causes NetWare to unload the NLM. (If only one thread is running, calling ExitThread with EXIT_THREAD as the action code parameter also terminates the NLM.)
- The NLM is unloaded with the UNLOAD command.

If the NLM never self-terminates, then the only function that needs to be defined is AtUnload or a SIGTERM signal handler. These functions gain control when the operator issues the UNLOAD command.

2.10.8 Freeing Resources upon Exit

NLMs are responsible for freeing the resources they allocate, such as memory, sockets, screens, devices, semaphores, and so on. NLMs should return all allocated resources to the OS during the termination process. If an NLM has not freed all its resources upon program termination, NetWare issues a warning message such as the following:

```
5/24/93 3:30pm: Module did not release 500 resources.
Module: Hello
Resource: Small memory allocations
Description: Alloc Short Term Memory
```

During NLM termination, NetWare and the NetWare API attempt to free all resources that the NLM allocated. Local semaphores are the only resource that cannot be freed; they must be freed with calls to CloseLocalSemaphore. If an NLM does not close all allocated local semaphores upon termination, the server abends.

To avoid these problems, we strongly suggest that you use a signal handler. Most frequently, the NLM is unloaded from the console command line with the unload command, and the SIGTERM signal handler is appropriate (see Section 4.2, “Terminating an NLM,” on page 71).

More Advanced NLM Concepts

3

Besides the issues of the NetWare operating system, you need to understand the structure of NLMs as well as services that are provided by the NetWare API. The following topics give an overview of the features that should be used when writing code for an NLM.

This section covers the following topics:

You should also understand the following concepts which are described in other sections of this manual or in other CLib manuals.

- Threads (*NDK: NLM Threads Management*)
- Context and Thread Groups (*NDK: NLM Threads Management*)
- Multithreaded Programming (*NDK: NLM Threads Management*)
- Context (*NDK: NLM Threads Management*)
- "Structure of an NLM" on page 47
- "NLM Startup" on page 48
- Relinquishing Control (*NDK: NLM Threads Management*)
- Shared Memory (*NDK: NLM Threads Management*)
- "NLM Termination" on page 51

3.1 Data and Parameters in NLMs

The following information describes alignment of data and passing C parameters.

3.1.1 Data Alignment

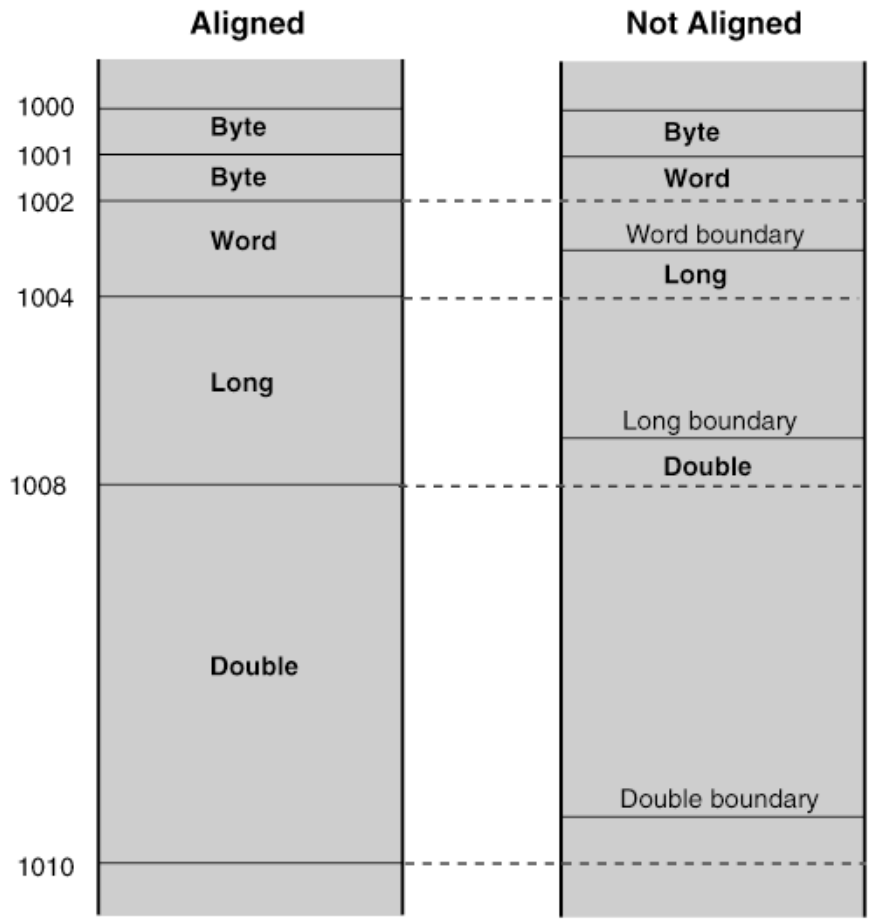
Data alignment describes data that is within the boundaries that are associated with each data type. Each data type, BYTE, WORD, LONG, and DOUBLE have rules where their starting addresses should begin. These rules are summarized in the following table.

Data Type	Starting Address Boundary
BYTE	Addresses that are multiples of one. No alignment issues.
WORD	Addresses that are multiples of two.

Data Type	Starting Address Boundary
LONG	Addresses that are multiples of four.
DOUBLE	Addresses that are multiples of eight.

A variable that does not fit in these boundaries is not aligned. For example, a LONG is four bytes, and should start on an address that is a multiple of four. If it does not start on the correct boundary, it spans boundaries, as shown in the following figure.

Figure 3-1 *Aligned vs. Unaligned Data*

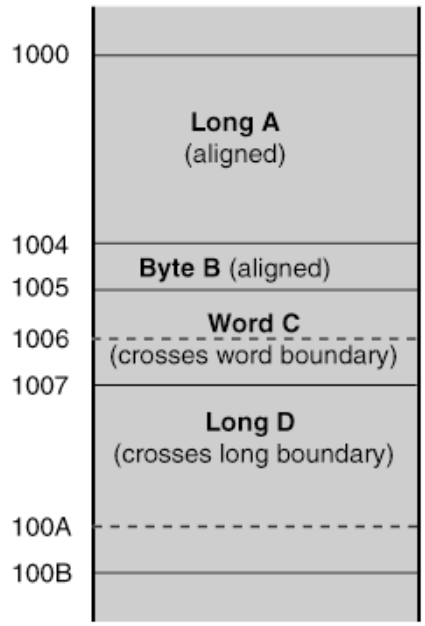


One place where alignment problems are common is in structure definitions. For example, the structure

```
struct {
      LONG A,
      BYTE B,
      WORD C,
      LONG D
} BadStruct;
```

is a poorly-designed structure because the placement of B in the structure causes C and D to cross long boundaries (they are not long-aligned). This is shown in the following figure.

Figure 3-2 *Poorly Aligned Structure*

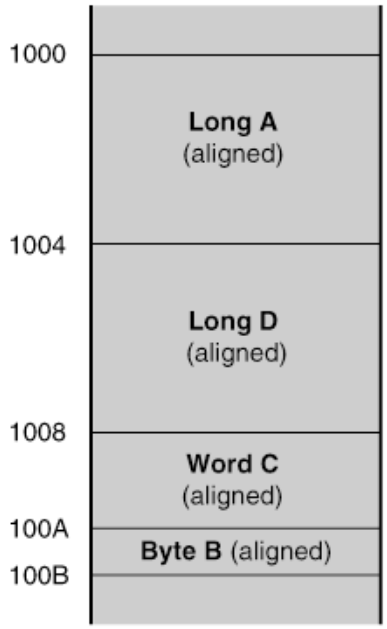


To solve the alignment problem (and to speed up the NLM), you can arrange the structure fields as follows:

```
struct {
      LONG A,
      LONG D,
      WORD C,
      BYTE B,
} GoodStruct;
```

As shown in the following figure, this is a well-designed structure because it does not have alignment problems. The fact that B is not on a long boundary is not an issue; BYTEs do not need to be long aligned because a byte can never cross a boundary as a WORD or a LONG can.

Figure 3-3 *Well-Aligned Structure*



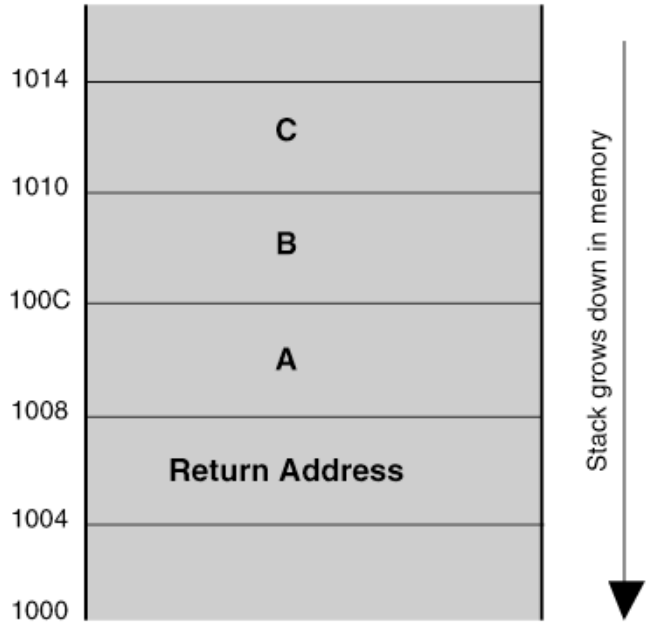
3.1.2 C Parameter Ordering

When C calls a function, it places the function parameters on the stack in the reverse order that they appear in the call. (The rightmost parameter is the first parameter that C pushes on the stack.) For example, when C issues the call

```
MyFunc(A, B, C);
```

it places the parameters on the stack in the order of C, B, and A, as shown in the following figure.

Figure 3-4 *Parameter Ordering*



3.2 Threads, Multithreaded Programming, and Context

The NetWare operating system supports multiple threads running concurrently and has made extensive use of threads. Although it is possible to write a simple NLM that uses only a single thread, an understanding of threads, thread groups, and context is key to successfully creating NLMs that are complex enough to provide very useful services.

Much more complete information about threads, multithreaded programming, and context can be found in Context and Thread Groups and Context in the Threads Concepts chapter of *NDK: NLM Threads Management*.

NOTE: Developers of drivers, stacks, and other low-level code, please refer to Context and Development of Drivers, Stacks, etc. in the same documentation.

3.3 Screen Handling

You can create, switch, and destroy screens from within server-based applications. A single NLM can have multiple screens, one screen, or no screens.

3.3.1 Screen Creation

Screens are created using CreateScreen (page 237), which returns a handle to the new screen, but it does not switch to the new screen. You switch to a screen by passing a screen handle into SetCurrentScreen (page 267). Once the current screen has been set, all output for functions such as printf go to that screen.

The following function shows the creation of a new screen, which is then switched to be the current screen. A new thread is created from within ThreadTwo, so the new thread belongs to the same thread group as ThreadTwo. Therefore, its output goes to the new screen.

```
void ThreadTwo()
{
   int screenHandle;
   int oldScreen;

   screenHandle = CreateScreen("Second Screen", 0);
   oldScreen=SetCurrentScreen(screenHandle);
   BeginThread(ThreadThree, NULL, NULL, "THREAD THREE ");
   printf("This is printing on the second screen\n");
}
```

NOTE: Setting the current screen changes the current screen for all of the threads within the thread group.

3.3.2 Screen Deletion

By default, NLM screens do not close automatically when the NLM terminates. Instead, the following message is displayed at the bottom line of the monitor console when an NLM terminates:

```
<Press any key to close screen>
```

This occurs because some preexisting applications that are being ported to the NetWare environment might have been designed to write out a closing message to the screen and then terminate their execution. The auto-closing feature would immediately destroy the screen, possibly causing the operator to miss an important status message. If an application requires auto-closing, you can turn off the default <Press any key to close screen> screen-closing mode by using SetAutoScreenDestructionMode (page 265). Screens can also be closed within the NLM by calling DestroyScreen (page 239).

For more information about screens and how to write to them, see "Screen Handling Concepts" on page 221.

3.3.3 Input and Output Cursors

An NLM screen has two cursors associated with it: an input cursor and an output cursor. It is possible to position both an input cursor and an output cursor on the NLM screens, giving your application the ability to accept input at one location on the screen and write output at a different location. If you want to mimic the DOS cursor, you can couple the two cursors, causing them to always act as one cursor. (The default setting is to have the cursors coupled.) The cursor coupling is set using SetCursorCouplingMode (page 268).

3.4 NLM Synchronization

The Synchronization Services functions, part of the NetWare API, enable applications to coordinate access to network files and other resources. These services are divided into two categories: locking and semaphores.

3.4.1 Locking

Locking enables a thread to gain exclusive access to a file-related resource, such as a file, physical record, or logical record. Threads lock resources by entering the filename or record location and the size into a log table, then issuing a single call to lock every resource listed in the table. Normally, a thread logs a group of records and then locks them as a set. However, a thread can lock a single record when it is placed in the log table.

This technique of logging files and records as a set and locking them all at once ensures that either all files and records are locked or none are locked. Thus, the developer can prevent *deadlock*, in which two or more applications reach a stalemate trying to access resources locked by the other application.

IMPORTANT: Don't use locking when you are using connection 0 because locking temporarily disables the connection. This disables connection 0 for all modules using the connection. If you are going to use locking, acquire a connection using LoginToFileServer.

3.4.2 Semaphores

Locking allows only one thread to access a file-related resource, but semaphores limit the threads that can access network resources to a configurable number. You can also use semaphores to limit the number of users of a particular resource.

Semaphores can be associated with resources, such as files, structures, and devices. There are two types of semaphores:

Network semaphores

These apply to resources available to servers and workstations on the network. For more information on network semaphores, see Synchronization Concepts (Single and Intra-File Services).

Local semaphores

These apply to resources available only to a single server. For more information on local semaphores, see Interprocess Synchronization (*NDK: NLM Threads Management*).

If your NLM uses resources that could be used by a thread running on another workstation or server on the network, you might use network semaphores.

You can use local semaphores among multiple NLMs; however, an NLM must either pass its semaphore handles to other modules or export a function that returns the semaphore handles to other modules.

If you are using semaphores to communicate between threads in the same NLM, you might use local semaphores. Local semaphores are faster than network semaphores because they are simpler and easier for the NLM to find and implement.

IMPORTANT: Don't use network semaphores when you are using connection 0 because locking temporarily disables the connection. This disables connection 0 for all NLM applications using the connection. If you are going to use network semaphores, acquire a connection using LoginToFileServer.

3.5 Cross-Platform Functions for NLM Development

All cross-platform functions have now been ported to run on the NLM platform through the library CALNLM32.NLM, also included in this NDK.

These functions provide a rich API set for developing NLMs, especially for applications designed as utilities. Such applications can take advantage both of the rich and varied function set provided in the cross-platform libraries and of the speed and server-centric functionality of CLIB.

This module provides the conceptual information necessary to understand NLM development with cross-platform functions. For general development instructions, see Section 4.4, "Developing NLMs with Cross-Platform Functions," on page 76.

Central to understanding the cross-platform world in NLM development are two key concepts:

- Differences in assumptions between cross-platform functions and CLIB functions
- Differences in the connection model between these two environments

3.5.1 Differences in Assumptions

The cross-platform functions are developed under the assumption that a client needs access to services provided by the network, rather than services provided by a specific server. Ultimately, of course, all services are provided by one server or another, but it is entirely possible to be connected to the network, receive the needed services, and not know which server is providing those services. This development assumption is necessary for technologies such as NDS® to be implemented. NDS

provides services through replicas that reside on servers, but the emphasis is on the NDS tree, not on the server on which the replica of that tree resides. This assumption provides for a wide array of services with the capability of enterprise level management and functionality.

In contrast, CLIB has always been designed to provide services on a specific server, often the server on which the NLM is loaded. Because of this emphasis, CLIB does not have the concept of a directory (other than a file system directory). All requests are directed to a specified server, and all communication is server-centric. This assumption provides for very fast, very efficient access to services on a server. It also allows for operations that deal specifically with files and directories residing on a specific server.

3.5.2 Differences in Connection Models

Understanding these differences in connection model is central to successful development of NLMs with cross-platform functions.

- Cross-platform functions use a connection handle.
- CLIB functions use a connection ID.
- Connection handles are valid only for cross-platform functions, and connection IDs are valid only for CLIB functions.

Connection Handles

A connection handle is a number returned from any of a number of cross-platform functions that make an initial connection, such as NWCCOpenConnByName. A connection handle ultimately resolves to a connection on a specific server, but it is possible to make such a connection and not know which server the handle specifies. That would be the case if a connection were opened with NWCCOpenConnByPref, which makes a connection with a preferred transport type and returns a connection handle. However, it does not identify the server to which that connection is made.

IMPORTANT: You need a connection handle to call virtually every cross-platform function, but never attempt to pass a connection handle to a CLIB function as a connection ID.

Connection IDs

A connection ID is a number that refers to a connection on a specific server. A connection ID is returned from any CLIB function that make a connection, such as LoginToFileServer. A connection ID can refer to a connection to the server on which the NLM is loaded or to a connection to a remote server. This model incorporates the concept of a current connection, which remains in force until some function changes the connection to specify a different server, which then becomes the current connection.

IMPORTANT: Before a CLIB function that uses the services on or makes changes to the system of a particular server can be called, a connection ID must be returned by calling a function such as GetCurrentConnection.

3.6 Communicating with Other NLMs

A loaded NLM can call any function in any NLM that exports symbols. To use a function exported by another NLM, the current NLM must include an IMPORT statement in its directive file. In

addition, the NLM that contains the function must include an EXPORT statement in its directive file. For a discussion of directive files, see “Specifying a Linker Definition File” on page 22.

3.7 Introduction to Remote Server Support

Remote server support provides an NLM with the ability to access other servers on the network through the functions in the NetWare API. A *local* server can be defined as the server on which the NLM is loaded. Any other server on the internetwork to which an NLM can attach and log in to is considered a *remote* server.

Servers are identified by a server ID number.

Do not confuse the server ID with the connection number. The server ID identifies a particular server, whereas the connection number indicates a particular connection on a particular server.

NLM and thread group context play an important role in remote server support. Server IDs are placed at the NLM level of context. At this level, all connections are accessible by any threads running within the NLM. If any thread does a logout, all connections for all threads are cleared.

The *current* server, which is the server to which all calls are being directed, is maintained at the thread group level. Any thread within a thread group can directly affect the current server of all threads within that group.

To see if an NLM provides remote support, see the "Remote Servers" notation on its function description.

3.7.1 Accessing Remote Servers

A remote server is accessed by calling LoginToFileServer with a server name attached to the object name (server/object). If the specified server is found, it is assigned the next available server ID and this number is added to a remote session table maintained by the NetWare API. This remote server then has the same server ID for the life of the NLM even if all connections to it are logged out. However, if the NLM terminates and is loaded again, the same server might not have the same server ID.

Server IDs are assigned in the order in which logins are performed to remote servers. If an NLM logs in to server A and then to server B, server A has server ID 1 and server B has server ID 2 for the life of the NLM. If the NLM terminates and is loaded again but first logs in to server B, then server B is assigned server ID 1. The local server (the one that is actually running the NLM) is always assigned server ID 0, even if no login is performed to the local server.

Functions that use a pathname (such as chdir in Multiple and Inter-File Services) and close and open in Single and Intra-File Services) now accept a server name as part of the path (server/volume:path). If no server name is given, the path is assumed to be on the current server.

3.7.2 Changing the Current Server

The current server can be changed by calling SetCurrentFileServerID or chdir (Multiple and Inter-File Services). chdir allows a server name as part of the path. If the specified server is found in the remote session list, the current server ID is set to the specified server.

3.7.3 Logging Out from Remote Servers

An NLM can break connections to remote servers by calling any of the following functions:

- NWDSLogout--Logs an object out of the network leaving all server attachments and other session connections intact.
- Logout--Breaks all connections to all remote servers. This function does not allow an NLM to selectively maintain groups of connections. (Requires bindery context.)
- LogoutFromFileServer--Breaks all connections between a server and all logged objects from the NLM. This function allows an NLM to specifically target those connections that it no longer needs. (Requires bindery context.)
- LogoutObject--Allows an NLM that logged in multiple times to selectively break a connection between a particular logged-in object and a server. (Requires bindery context.)

3.7.4 Remote and Local Server Operations

Not all functions in the NetWare API work on remote servers.

The function descriptions include paragraphs that indicate whether the function supports only remote server or local server operations, or both.

The paragraphs are labelled *Local Server* and *Remote Server*. *N/A* in this paragraph indicates that the function does not support operation on the indicated type of server.

Additionally, for the type of server operations that the function supports, each function is further identified as nonblocking or blocking.

- Nonblocking functions do *not* cause the caller to lose its thread of execution (do not relinquish control).
- Functions that can block might cause the caller to relinquish control.

  For example, a function that is blocking on a remote server would read "*Remote Server:* blocking."
- Finally, some functions can be either blocking or nonblocking depending on the circumstances of the call. These functions are identified as "either blocking or nonblocking." When this is the case, a note is included in the Remarks section to explain the circumstances under which the function blocks.

Advanced NLM Tasks

4

This describes common tasks associated with writing NLMs that make use of the CLIB function libraries, that use multiple threads, and that directly or indirectly allocate memory and other resources for threads owned by the NLM.

4.1 Developing Multithreaded NLMs

1. Set up the development environment as with Section 1.6, "Writing a Basic NLM," on page 32.
2. Plan the threads of execution your NLM will run.

   All NLMs have at least one thread contained in one thread group to accommodate the main function. You can set up other threads to accomplish different tasks. For example, you might set up a thread for each incoming client request.

   NOTE: Threads and thread groups are discussed in Section 3.2, "Threads, Multithreaded Programming, and Context," on page 65.

   ---
3. Organize the threads into groups.

   For example, you can place threads that use the same current screen and current working directory in the same thread group.
4. Write the NLM source code as a server program, using the rules for NLMs in the nonpreemptive environment (see "Preemptive and Nonpreemptive Environment" on page 45).

   The NetWare API contains functions you can use in writing your NLM.
5. Compile, link, and debug the NLM using the procedures discussed in "NLM Development Tool Concepts" on page 79.

4.2 Terminating an NLM

An NLM is not properly finished until it terminates successfully. Most NLMs ordered to termination by entry of the UNLOAD command from the console. The following guidelines provide the principles for bringing an NLM to a clean, complete conclusion.

4.2.1 Clean Up All Resources Allocated Anywhere in an NLM

Not only must an NLM free all resources it has allocated, but each thread created by an NLM must also free any resources it has allocated.

Some NLM developers attempt to write an all-purpose cleanup procedure. However, such procedures are difficult to write successfully because they must free resources they did not allocate. Instead, write the NLM in such a way that every thread frees each resource it allocates when the resource is no longer needed. This method is much more effective, and it ensures that the complete NLM will not leave unfreed resources at termination.

4.2.2 Implement a Signal Handler (SIGTERM)

The advantage of using a SIGTERM handler instead of calling AtUnload or atexit is cleanup—when NetWare executes the handler for these functions, all of the NLM threads have been summarily terminated whether or not they have freed resources they allocated.

1. Using the signal function write a SIGTERM handler. Do not allow the handler to return until your main thread and all other NLM threads have freed allocated resources and terminated.
2. As one method of SIGTERM handler implementation, create two global integer variables:
   - NLM_exiting, initially set to FALSE (0)
   - NLM_threadCnt, initially set to zero (0)
3. For the first statement in the NLM main function, increment NLM_threadCnt.
4. For the last statement in the NLM main function, decrement NLM_threadCnt.
5. In the course of running, ensure that all loops within the NLM monitor the NLM_exiting variable. If NLM_exiting is ever set to TRUE, have all threads free any allocated resources and self terminate.

The following code illustrates one signal handler implementation:

```
void  NLM_SignalHandler (int sig)
    {
      switch (sig)
      {
      case SIGTERM:
         NLM_exiting = TRUE;
         while (NLM_threadCnt != 0)
            ThreadSwitchWithDelay()
         break;
      }
   return;
   }
```

4.2.3 Provide CLIB Context for the SIGTERM Handler if Needed

The SIGTERM handler is executed by an OS thread-the Console prompt thread. (Operating system threads cannot take advantage of most functions offered in the NetWare SDK) While the SIGTERM handler has control over this operating system thread, it accepts and executes commands from the server console screen. That fact makes two things very important to understand:

- The console command prompt will not be available until the signal handler releases it.
- Your SIGTERM signal handler must not destroy the thread executing it because that would destroy the command prompt. Therefore, do not call exit from your SIGTERM handler.

Under some circumstances, your SIGTERM handler might need to call NetWare SDK functions that require full CLIB context. It is possible to borrow a CLIB context, effectively converting the operating system thread to a CLIB thread. The following instructions explain how to do this:

1. Select an active thread in your NLM that already has CLIB context. The NLM main function is generally suitable for this purpose.
2. Call GetThreadGroupID and store the ID of the main function into a global variable such as NLM_mainThreadGroupID.

3. Call GetThreadGroupID again and store the existing operating system context of the SIGTERM handler thread.
4. Call SetThreadGroupID and assign the borrowed CLIB context stored in NLM_mainThreadGroupID to the SIGTERM handler operating system thread.
5. When operations requiring CLIB context are completed, call SetThreadGroupID again to set the SIGTERM handler thread back to its original context.

4.2.4 Allow for Blocked or Suspended Code at UNLOAD

1. Recognize a possible deadlock if an UNLOAD command is issued while a function is blocked.

   In the code below, assume that the body of main includes a function such as getch that blocks (or suspends) thread execution until a character is received from the keyboard. The console operator will likely attempt an UNLOAD routine at some time when the NLM is waiting for keyboard input. In this condition, the SIGTERM handler waits for main to decrement the NLM_threadCnt value to zero before proceeding. However, main does not decrement the value until it receives a character. As a result, the system console screen appears to be hung and the NLM does not unload as requested.

2. If a function is blocked when an UNLOAD command is issued, call a function that can allow processing to continue.

   The SIGTERM handler is responsible for waking up any blocked or suspended threads so that they can become aware of the NLM_exiting value. (In turn, each thread is responsible for checking the NLM_exiting value as often as appropriate.) The SIGTERM handler can help wake up a thread blocked on the getch function by calling the ungetch function and stuffing a character into the keyboard buffer. This character is then read out of the keyboard buffer by the blocked getch and execution can proceed. Other blocking functions to watch out for include gets, t_snd, NWSList, NWSMenu, SuspendThread, and delay.

```
int NLM_mainThreadGroupID;
int NLM_threadCnt = 0;
int NLM_exiting = FALSE;

void NLM_SignalHandler(int sig)
   {
   int handlerThreadGroupID;
   switch(sig)
      { case SIGTERM:
         NLM_exiting = TRUE;
         handlerThreadGroupID = GetThreadGroupID();
         SetThreadGroupID(NLM_mainThreadGroupID);

         /* NLM SDK functions may be called here */

         while(NLM_threadCnt != 0) ThreadSwitchWithDelay();
         SetThreadGroupID(handlerThreadGroupID);
         break;
      }
   return;
   }
void main(void)
   {
   ++NLM_threadCnt;
```

```
   NLM_mainThreadGroupID = GetThreadGroupID();
   signal(SIGTERM, NLM_SignalHandler);

   /* Body of main continues here... */

   --NLM_threadCnt;
   return;
   }
```

4.2.5 Allow for Child Threads and Call-backs

The main function, all child threads, and any call-back routines should use the NLM_threadCnt variable to keep make sure resources are cleaned up for NLM termination.

1. As shown in the above sample, write main to increment the NLM_threadCnt as its first action and to decrement NLM_threadCnt as its last.
2. If your NLM calls BeginThread or any similar function, ensure that the spawned thread increments and decrements the NLM_threadCnt in the same way that main does.
3. If your NLM sets up call-back routines, ensure that each of those routines increments and decrements the NLM_threadCnt as does main.

   Call-back routines might include functions such as NWAddFSMonitorHook, NWRegisterNSPExtension, and RegisterForEvent.

4.2.6 Allow for Normal NLM Termination

When it is appropriate, allow your NLM to terminate normally. The steps below can safeguard against premature termination:

1. Ensure that your NLM does not terminate until the NLM_threadCnt is decremented to 0.
2. Ensure that your main thread never terminates until the NLM_threadCnt has a value of 1, indicating that only main is still running.

Implementing these safeguards allows any thread in your application to shut down the NLM by setting NLM_exiting to TRUE, but it also forces main to stay alive until all other NLM threads have terminated. The following code sample illustrates this:

```
void main(void)
   {
   ++NLM_threadCnt;

   NLM_mainThreadGroupID = GetThreadGroupID();
   signal(SIGTERM, NLM_SignalHandler);

   /* Body of main continues here... */

   while(NLM_threadCnt != 1)
      ThreadSwitchWihDelay();

   --NLM_threadCnt;
   return;
   }
```

4.2.7 Protect Against CTRL-C

Users can break out of your NLM using CTRL-C. You can avoid this in either of two ways:

- Disable CTRL-C functionality by calling SetCtrlCharCheckMode.
- Register a SIGINT signal handler that causes your NLM to ignore CTRL-C, as illustrated in the code example below.

  Notice that the SIGINT signal handler must be reregistered each time a CTRL-C event occurs.

```
int NLM_mainThreadGroupID;
int NLM_threadCnt = 0;
int NLM_exiting = FALSE;

void NLM_SignalHandler(int sig)
   { int handlerThreadGroupID;
   switch(sig)
      { case SIGTERM:
         NLM_exiting = TRUE;
         handlerThreadGroupID = GetThreadGroupID();
         SetThreadGroupID(NLM_mainThreadGroupID);

         /* NLM SDK functions may be called here */

         while(NLM_threadCnt != 0)
            ThreadSwitchWithDelay();

          SetThreadGroupID(handlerThreadGroupID);
          break;

      case SIGINT:
         signal(SIGINT, NLM_SignalHandler);
         break;
      }
   return;
   }

void main(void)
   {
   ++NLM_threadCnt;

   NLM_mainThreadGroupID = GetThreadGroupID();
   signal(SIGTERM, NLM_SignalHandler);
   signal(SIGINT, NLM_SignalHandler);

   /* Body of main continues here... */

   --NLM_threadCnt;
   return;
   }
```

4.3 Designing Client-Server NLMs

When designing a client-server NLM, you should consider several issues, such as division of workload and communication methods. In general, follow these steps to design a client-server NLM:

1. Divide the modules of the program into client-based tasks and server-based tasks.

   Place tasks on the computer that can most efficiently process them. For instance, some I/O operations should be placed on the client because the data or resource is on the client.

   Consider all possibilities in dividing the program, such as all processes on the server or all processes on the client, to help you reach a division of tasks that provides the most computing power for the least computing effort.

   Consider scalability. For example, if you overload the server with CPU-intensive operations, your application might work well for 10 users, but not for 1,000.

   Try to balance processing loads between the client and server, so that neither is overburdened. However, for scalability, the client should take on a slightly larger share of the processing.

2. Choose both an interface and a protocol for communication between client and server programs.

   Choose the lowest-level interface possible without significantly compromising the level of effort required to implement it.

   Consider portability issues. For example, WinSock 32 offers a wide range of portability. Using TLI eases supporting multiple client platforms, whereas IPX™ and SPX™ limit those choices.

3. Determine the maximum number of clients to accept.

   Consider the following:

   - Memory--If the tasks your clients request of the server require large amounts of memory, you might want to limit the number of clients to avoid running out of memory.
   - Performance--The more efficient the server is, the more clients it can support. On the other hand, if the tasks your clients request of the server reduce the performance of the server as a whole, you might want to limit the number of clients.
   - Connection considerations--The number of clients your NLM can accept is limited by the number of connections the server computer accepts. For example, if clients communicate with the server by modem and the server has four modems, your NLM can accept no more than four clients at one time.

4.4 Developing NLMs with Cross-Platform Functions

This general methodology for developing NLMs with cross-platform functions assumes understanding of key concepts, especially the differences between a connection handle and a connection ID. See Section 3.5, “Cross-Platform Functions for NLM Development,” on page 67 for an explanation of those differences.

WARNING: Never attempt to use connection zero in an NLM that uses cross-platform functions. The results would be a broken application, a major security breach, or both.

1. Ensure that library NLMs needed for cross-platform are loaded in the correct order:
   - If your NLM includes calls to the cross-platform libraries, CALNLM32.NLM must be the first NLM loaded. It automatically loads modules on which it has dependencies, including CLIB.NLM and its associated modules.
   - If your NLM does not include calls to the cross-platform libraries, load CLIB.NLM first. Its associated modules will also load automatically.
2. Set up the correct include order for header files.
   - If an NLM makes calls to CALNLM32.NLM, specify the following include order in the make file or in a SET command:

     ```
     NWSDK\INCLUDE
     NWSDK\INCLUDE\NLM
     ```
   - If an NLM makes calls to only to CLIB functions, specify the following include order in the make file or in a SET command:

     ```
     NWSDK\INCLUDE\NLM
     NWSDK\INCLUDE
     ```

   NOTE: Some header files in the NWSDK\INCLUDE directory have the same names as files in the NWSDK\INCLUDE\NLM directory, but the contents of such files are not identical.
3. Keep track of the connection handle and connection ID for each server from which your application requires services.

   One simple way of obtaining both numbers is as follows:
   - Call the cross-platform function NWCCOpenConnByName and save the returned connection handle to a variable or to a table.
   - Immediately call the CLIB function GetCurrentConnection and save the returned connection ID to another variable or table entry.
4. Pass in the appropriate connection handle for each cross-platform function called, and the appropriate connection ID for each CLIB function called.

   IMPORTANT: *Do not pass a connection handle to a CLIB function or a connection ID to a cross-platform function.*
5. Before calling a CLIB run-time function that does not take a connection ID parameter, such as ParsePath or chdir (Multiple and Inter-File Services), make sure the current connection is set to the server on which the operation is being performed.

   If there is any question about what the current connection is, do one of the following:
   - If you know the CLIB connection ID, call SetCurrentConnection and specify the ID for the appropriate server.
   - If you know only collection handles, call NWCCSetCurrentConnection and pass in the handle for the server on which the operation is to be performed. Then call GetCurrentConnection and use the returned connection handle for subsequent calls to CLIB functions.

NLM Development Tool Concepts 5

This section provides an overview of the following tools for NLM development:

- NLM Make Utilities (page 79)
- Debuggers for NLMs (page 82)
- NLM Compression Tools (page 93)
- MPKXDC (page 93)

For information on compilers, linkers, and test tools, see "Getting Started" on page 13.

5.1 NLM Make Utilities

The WATCOM make utility is WMAKE. The makefiles associated with the examples that ship with this NDK are written to be used by WMAKE. To compile an example, you simply move to the directory where the example is located and type:

```
WMAKE
```

To assist you in creating WMAKE-style makefiles for your programs, this NDK ships with the QMK386 utility. QMK386 generates makefiles for WMAKE based on user input and entries in the MAKEINIT file generated by the MAKEINIT.EXE file. You can customize the makefiles after QMK386 creates them. QMK386 is located in the TOOLS directory.

5.1.1 QMK386.EXE

Information about this utility consists of the following sections:

- "Syntax for QMK386.EXE" on page 79
- "Options for QMK386.EXE" on page 79
- "Environment Variables for QMK386.EXE" on page 81
- "Examples for Using QMK386.EXE" on page 81
- "Notes for Using QMK386.EXE" on page 82

Syntax for QMK386.EXE

```
[d:][path]QMK386 <progname> [/options]
```

[d:][path] specifies the drive and path containing the QMK386 command file.

<progname> specifies the name of the modules whose makefile is being created. If this parameter is not set, the default name is "NONAME."

[/options] See Options for QMK386.EXE

Options for QMK386.EXE

To display the QMK386 options, type:

```
QMK386 ?
```

Option	To
/?	Display help screens.
/c<x>	Specify additional objects, imports, exports and modules:
	e<spec> - Export file
	i<file> - Import file
	l<spec> - Library file
	m<spec> - Module dependency
	s<file> - Source file
/f<name>	Specify name of output file. Default: MAKEFILE
/h<dir>	Specify additional INC386 directories.
/i<x>	Include import statement for ...
	a - AIO symbols
	c - CALNLM32 (cross-platform NWCalls symbols)
	d - NETNLM32 (cross-platform NDS symbols)
	e - DSEvent symbolsf - Floating point support symbols
	h - Threads symbolsI - NIT NLM symbols
	l - AFP symbolsn - NLM-specific symbols (NLMLIB)
	o - SOCKLIB symbolsp - Print symbols
	r - Requester symbolss - Streams symbols
	t - TLI symbolsu - Unicode symbols
	v - 3.x Print Services symbols
	w - NWSNUT symbols
	x - CLXNLM32 (cross-platform NWClient symbols)
	y - AUDNLM32 (cross-platform Auditing symbols)
	z - LOCNLM32 (cross-platform NWLocale symbols)
	NOTE: The recommended method is to use the ALL.IMP file to load all symbols. The /i option cannot be used with the ALL.IMP file.
/l<dir>	Specify additional LIB386 directories.
/n<x>	NLM option ...
	m - Load Multiple
	p - PseudoPreemption
	r - ReEntrant
	s - Synchronize

Option	To
/o<x>	Other option ...
	b<path> - shareliB path
	c<api> - Check function
	d<path> - xDc data path
	h<file> - Help file
	m<file> - Message file
	s<api> - Start function
	u<file> - cUstom file
	x<api> - eXit function
	/p
	Run NLMPack after linking.
/s#	Set stack size to `#' KBytes.
/x	No default screen for NLM
/z	Generate an NLMLINK-style linker definition file.

Environment Variables for QMK386.EXE

The following environment variables may be used with QMK386:

SWITCHAR sets the character that QMK386 uses as the delimiter for parameters. By default, the forward slash `/' is used.

QMK386 can be used to set options that are commonly used. For example, if you commonly use /Z and /IDT, set the environment variables as follows to avoid entering these options at the command line:

```
SET QMK386=/z /idt
```

SILENT places the SILENT directive in the makefile when set.

QMKVER allows you to specify the default way to build an application. If QMKVER is not specified, it will default to `d', which uses DEBUG compile switches, and includes debug information in the output file.

CCF386 specifies the command line switches to be used when rebuilding an application.

Examples for Using QMK386.EXE

This section contains some examples of common build options used with QMK386. The parameters can appear in any order, and no spaces are required between options unless the options require a directory path or function name.

To create a WLINK style makefile for a program called HELLO, with an 8K stack, importing from CLIB.IMP, THREADS.IMP and NLMLIB.IMP:

```
F:\> qmk386 hello /ih /in
```

TIP: Import files can be included individually with the /i option (for example, /oh includes the THREADS.IMP file). To inlcude the ALL.IMP file (which includes all the import files), do not use the /i option (ALL.IMP is the default setting).

Using the ALL.IMP file might require more memory than is available in your environment.

To create a WLINK style makefile for a program called TEST1, with a 12K stack, importing from CLIB.IMP and TLI.IMP:

```
F:\> qmk386 /s12 /it test1
```

To create an NLMLINK style makefile for the example above:

```
F:\> qmk386 /s12 /it test1 /z
```

To create an NLMLINK style makefile, and run the NLMPACK utility:

```
F:\> qmk386 /s12 /it test1 /z /p
```

The /CS option allows you to specify source files. You can specify multiple /C<x> options, as well as use wildcards. This allows you to specify multiple dependencies for the first target. For example, to create a MAKEFILE for an NLM called MYNLM, made up of all the .C files in the current directory, importing a file called EXLIB.IMP and specifying a NetWare Loader module dependency on EXLIB.NLM:

```
F:\SRC> qmk386 mynlm /CS*.c /CIexlib /Cmexlib
```

Notes for Using QMK386.EXE

QMK386 MAKEFILEs define the following macros for your .C programs:

- t vMAJ - Major version number
- t vMIN - Minor version number
- t vREV - Revision number

The WMAKE macros which define the values of these are pvmaj, pvmin and pvrev. They are defined at the top of your MAKEFILE, and if you change the values, they are reflected in both the C macros as well as the version number passed to the linker. One way to create a version string in your NLM might be:

```
#define VERSION "Widget Monitor NLM v"vMAJ"."vMIN
```

This would expand to:

```
#define VERSION "Widget Monitor NLM v1.00"
```

if the default values were used.

5.2 Debuggers for NLMs

Sometime during your code development, you will need to use a debugger. The following debugging tools are available to facilitate your development of high-quality software:

- The CodeWarrior debugger allows you to debug NLMs remotely.
- The internal command line debugger that performs symbolic debugging, built into the NetWare operating system (see “NetWare Internal Debugger” on page 83).
- The WATCOM linker (WLINK) can be used to include debugging information in an executable file (see “Linking Debug Information with WLINK” on page 83).

Additionally, the linkers can be used to generate a map file, which is a memory map of your program.

5.2.1 Linking Debug Information with WLINK

You can specify the DEBUG directive in a linker directive file to generate debugging information in the executable file. The following options can be specified with the DEBUG directive to generate the following types of debugging information:

DEBUG ALL

Generates all types of debugging information (global symbol, line numbering, local symbol, and typing).

DEBUG NOVELL

Generates global symbol debugging information that can be processed only by the NetWare Internal Debugger

DEBUG NOVELL ONLYEXPORTS

Generates NetWare global symbol information for exported symbols only.

DEBUG ONLYEXPORTS

Generates WVIDEO global symbol information for exported symbols only.

5.2.2 NetWare Internal Debugger

The NetWare internal debugger is an assembly language debugger that is built into the NetWare operating system. This debugger is a command-line debugger that does not display source code. To use the internal debugger, you should have some knowledge of 80386 assembly language and stack-based parameter passing.

The internal debugger was designed specifically to debug NLMs. It includes a set of supplementary commands that are customized for NLMs, such as .A (display abend or break reason) and .P (display all process names and addresses). These are not part of a typical debugger. The internal debugger allows resident debugging, in which the debugger and the test application run on the same server. In addition, the internal debugger provides a way to debug multiple NLMs concurrently.

NOTE: The NDK includes debug versions of the libraries. Debug records are linked in with each of these NLMs, allowing better visibility to developers using the internal debugger.

You can access the NetWare Internal Debugger in any of the following ways:

- At the server console, simultaneously press the following keys: Left-shift+Right-shift+Alt+Esc.

  NOTE: If the SECURE CONSOLE command is in effect, you cannot access the NetWare Internal Debugger from the keyboard.

- From a C language program, call Breakpoint.
- From an assembly language program, issue an INT 3 instruction.

You can then set execution breakpoints, single-step through program execution, examine the contents of memory, and so on.

Some points to be aware of when using the internal debugger are:

- NetWare runs in the 386 protected mode, using a flat memory model. In a flat memory model, the values in the segment registers do not change once they are initialized by the NetWare operating system. Since they do not change, the internal debugger does not display them.
- The internal debugger supports program global symbolic information; it does *not* support local symbolic information. Any symbols that you want to reference from the internal debugger must be system-wide globals. To access symbolic information, the program must be linked with the DEBUG option.
- The internal debugger is case-sensitive to symbols.
- All numbers are entered and displayed in hexadecimal format.
- Bytes, words, double-words, and pointers are pushed onto the stack as 4-byte parameters.

For more specific information on the debugger, see the following:

- "Debugger Commands" on page 84
- "Setting Breakpoints" on page 90
- "Specifying Expressions" on page 90

Debugger Commands

You can recall commands from the NetWare Internal Debugger's command line buffer by pressing the Up-arrow key. After recalling a command from the command-line buffer, you can edit it. The Right- and Left-arrow keys move the cursor. Insert toggles overwrite. Some of the commands can be repeated by pressing the Enter ; these cases are noted in the command descriptions.

NOTE: If you decide to cancel a command, the Esc key acts like the Enter key. You must use the Delete or Backspace key to erase the command line.

There are four types of help commands in the NetWare Internal Debugger:

- HE-Help on expressions
- HB-Help on breakpoints
- H-General Help
- .H-Help with the supplementary commands

In the command summaries, a pair of square brackets in the Command indicates an optional parameter. The following categories of commands are available:

- "Supplementary Commands" on page 84
- "Breakpoint Commands" on page 86
- "General Debugger Commands" on page 86
- "SFT III Debugger Commands" on page 89

Supplementary Commands

The following table lists supplementary commands.

Command	Description
.a	Displays the abend or break reason.
.c	Does a diagnostic core dump to diskette (this can take a great number of diskettes).
.d [address]	If no address is specified, displays a page directory map for the current debugger domain. When address is specified, displays page entry map for the current debugger domain.
.h	Displays help information about the supplementary commands.
.l offset [offset]	Displays linear address given page map offsets.
.lx address	Displays page offsets and values used for translations.
.m	Displays the names and addresses of the loaded modules.
.p [address]	If no address is specified, displays process (thread) names and addresses. If address is specified, displays address as a process (thread) control block. You can use this command to determine what a particular thread is doing. For example, you can examine the values on the stack, which contain return addresses for called functions, to determine what an inactive task is doing (waiting on a semaphore, waiting on keyboard input, and so on). That is, you can construct a "trail" of functions that have been called. This command now displays the semaphore address when listing processes waiting on a semaphore.
.r	Displays running process (thread) control block. This command displays information about the running thread in the same format as the .p address command.
.s [address]	If address is not specified, displays all screen names and their addresses. If address is specified, displays the specified address as a screen structure. A pointer value obtained by the .s command is used as the address parameter. The command .s address is another way to get information about the current activity of a sleeping thread.
.sem [semaphore address]	If an address is not specified, lists all semaphores that have processes waiting on them. If a semaphore address is specified, displays detailed information about the semaphore.
.t	Toggles the "developer option" on or off.
.v	Displays server version.

Breakpoint Commands

The following lists breakpoint commands.

b

Displays all current breakpoints.

bc `number`

Clears the specified breakpoint.

bca

Clears all breakpoints.

b = `address` [(`condition`)]

Sets an execution breakpoint at address.

Example : Breakpoint if MyFunction is called and the first parameter on the stack is equal to 0:

```
b = MyFunction [desp+4] == 0
```

br = `address` [(`condition`)]

Sets a read or write breakpoint at address.

Example : To check if the code (in the range 14500 to 15500) ever reads or writes to memory location 160FE:

```
br = 160FE EIP >= 14500 && eip <= 15500
```

bw = `address` [(`condition`)]

Sets a write breakpoint at address.

Example : To check if the code (in the range 14500 to 15500) ever writes to memory location 160FE:

```
bw = 160FE EIP >= 14500 && eip <= 15500
```

General Debugger Commands

The following lists the general debugger commands.

c address

Interactively changes memory.

c address=numbers

Changes memory, starting at address, to numbers.

Example : Change byte values starting at 10DFAB to FF, FE, 22.

```
c 10DFAB = FF,FE,22
```

c address = "text"

This command is currently not supported.

d address [length]

Dumps length bytes of memory starting at the address. If length is not specified, 256 (decimal) bytes are dumped.

Example : Dump 16 (decimal) bytes at address 00088F20.

```
d 88F20 10
```

This command can be repeated by pressing Enter. You can visually scan for a string in the ASCII portion of the dump display by dumping a memory location and then repeatedly pressing Enter to display contiguous blocks of memory.

dl[+ `linkOffset`] addr [length]

Traverses a linked list. If length is not specified, 256 (decimal) bytes are dumped.

Example : Suppose the first node in a linked list starts at 50 and the offset of the address of the next node is at offset 4.

To traverse the linked list, displaying 16 (decimal) bytes each time, enter the following command.

```
dl+4 50 10
```

To display each successive node in the list, press Enter.

The default link offset is 0, which indicates the end of the list. Thus, dl 50 10 uses a link offset of 0.

This command can be repeated by pressing Enter. You can dump the first node in a linked list and then dump each successive node by pressing Enter. A NULL link marks the end of the list.

f flag=value

Changes the specified flag. value can be 0 or 1.

Example : To change the specified flag to the new value (0 or 1), where flag is CF, AF, ZF, SF, IF, TF, PF, DF, or OF:

```
f CF = 0
```

g

Specifies a "Go" instruction, starting from the current EIP.

g [break_addresses]

Specifies a "Go" instruction, starting at the current EIP and ending at the break address or addresses.

Example : Suppose a code breakpoint has just occurred at the start of a C function. To resume execution until the function returns to its caller, use the following command:

```
g [desp]
```

h

Displays general help.

hb

Displays breakpoint help.

he

Displays expressions help.

i[b,w,d] port

Inputs a byte, word, or double-word from the specified port. The default is a byte.

Example : To input the value at port 2F0:

```
i 2F0
```

m start [L length] pattern

Memory search is currently not supported.

n

Lists all symbol names, also displaying the NLMs defined them.

n symbolname value

Defines a new symbol name at an address.

Example : To give the value 2D46A5 the name x:

```
n x 2D46A5
```

Now x can be referenced with other commands, such as:

```
b=x, b=x+5, u x.
```

By default, the value is 10. Symbols can be defined with the n command. The y option when the server is started is used to override the default.

n-symbolname

Removes a user-defined symbol name.

n-

Removes all user-defined symbol names.

o[b,w,d] port = value

Outputs byte, word, or double-word to the specified port.

Example : To output 10h to port 320h:

```
o 320=10
```

p

Single-steps through the program code; proceeds past calls. (See the s command for stepping into calls.)

This command can be repeated by pressing Enter. A common usage for this command is to run until you hit a breakpoint, and then single-step by entering the p command and then pressing Enter repeatedly to continue single-stepping. By holding down Enter, you can quickly single-step through the program code.

q

Quits to DOS.

r

Displays the registers and flags.

REG = value

Changes the register to the specified value. The registers are EAX, EBX, ECX, EDX, ESI, EDI, EBP, EIP, and EFL.

s

Single-steps through the program code; steps into a call. (See the p command for stepping past calls.)

This command can be repeated by pressing Enter. You can hit a breakpoint and single-step by entering the s command and then pressing Enter repeatedly to continue single-stepping. By holding down Enter, you can quickly single-step through the program code.

t

Same as s.

u address [count]

Disassembles count instructions. If you type u by itself and press Enter, the starting address is assumed to be the contents of EIP, and 16 (decimal) bytes will be disassembled.

Example : Disassemble 16 (decimal) bytes prior to the current instruction.

```
u eip-10
```

NOTE: A command such as this might not cause the disassembly to fall on an instruction boundary.

This command can be repeated by pressing Enter. You can disassemble starting at any memory location by initially entering the u command and then pressing Enter to continue the contiguous disassembly.

v

Displays the server's screen(s) for viewing. Each time a key is pressed, the next screen is displayed. See the .s command.

x

Exchanges processor stack frames.

z expression

Evaluates the expression.

Example: To display the value at the address computed by adding EBP to EBX shifted right 16 (decimal) times.

```
z [ d EBP + (EBX >> 10) ]
```

? [address]

Displays nearest symbols to address. If address is not given, EIP is used.

Example : To determine the NLM and function owning the current instruction, type the following:

```
?
```

SFT III Debugger Commands

The following table lists the debugger commands that are available for the SFT III™ OS only.

Command	Description
.?	Display server state.
DQ	Dump level 3 queue pointers.
DQ `address`	Dump level 3 queue elements.

Setting Breakpoints

With the NetWare Internal Debugger, you can set execute, write, or read/write breakpoints.

There are four breakpoint registers, allowing a maximum of four simultaneous breakpoints. Breakpoints can be permanent or temporary:

- To set permanent breakpoints, use the b, br, and bw commands. For permanent breakpoints, you can attach a condition that specifies whether to take the breakpoint. If the condition is true, a breakpoint is taken. If it is false, execution continues without stopping.
- To set temporary breakpoints, use the g command. For example, a "go" to a specific address is a temporary breakpoint. The p command can also set a temporary breakpoint if the current instruction cannot be single-stepped.

  If you use all four breakpoints and issue a g [desp] command, the following is displayed:

  ```
  Go out of breakpoints
  ```

  If you use all four breakpoints and attempt to proceed past a function call using the p command, the following is displayed:

  ```
  Breakpoint not available for proceed
  ```

  The assembly repeat instructions (such as REPE), the LOOP instruction, and the CALL instruction also require p to set a temporary breakpoint.

Specifying Expressions

The NetWare Internal Debugger determines the order of execution of an expression in accordance with the following:

- Precedence of grouping operators
- Precedence of unary, binary, and ternary operators
- Common algebraic ordering

The following sections explain the various types of operators:

- "Grouping Operators" on page 90
- "Unary Operators" on page 91
- "Binary Operators" on page 91
- "Ternary Operators" on page 92
- "Registers and Flags" on page 92

Grouping Operators

The grouping operators ( ), [ ], and { } indicate to the debugger the desired grouping of operations. These operators have the highest precedence (0).

( )

(expression)

The terms inside the parentheses are evaluated first. In the case of parenthetical expressions that are nested, evaluation begins with the innermost parenthetical expression.

[ ]

[size expression]

expression is evaluated first and then used as a memory address. The size specifier can be B,W, or D. The expression evaluates to byte, word, or double-word at the specified address.

For example: Suppose the data at memory location 178D10 is the following byte sequence in Intel storage format 38 F9 99 88. Then, using the z command, which evaluates expressions:

Z [D 178D10]	evaluates to	8899F938
Z [W 178D10]	evaluates to	F938
Z [B 178D10]	evaluates to	38

{ }

{size expression}

expression is evaluated first and then used as a port address. The size specifier can be B,W, or D. The expression evaluates to byte, word, or double-word from the port.

Unary Operators

The unary operators have precedence 1.

Symbol	Description
!	Logical not
-	2's complement
~	1's complement

Binary Operators

The binary operators in the following table are ordered from lowest to highest precedence.

Symbol	Description	Precedence
*	Multiply	2
/	Divide	2
%	Mod	2
+	Add	3
-	Subtract	3
>>	Bit shift right	4
<<	Bit shift left	4
>	Greater than	5
<	Less than	5
>=	Greater than or equal to	5
<=	Less than or equal to	5
==	Equal to	6

Symbol	Description	Precedence
!=	Not equal to	6
&	Bitwise AND	7
^	Bitwise XOR	8
|	Bitwise OR	9
&&	Logical AND	10
||	Logical OR	11

Ternary Operators

If expression1 is true, the result is the value of expression2; otherwise, the result is the value of expression3.

```
expression1 ? expression2 , expression3
```

In the following example, a break is taken on

```
myFunction( char *myData )
```

if the carry flag (FLCF) is true and EAX contains 9C, or if the carry flag is false and the first byte of myData is 0:

```
b = myFunction ( FLCF ? eax == 9c, [b [desp+4]] == 0 )
```

Registers and Flags

The 80386 registers, which can be used in expressions, are referenced by the names listed in the following table.

Register	Name
EAX	Accumulator register
EBX	Base register
ECX	Count register
EDX	Data register
ESI and EDI	Index registers
ESP and EBP	Base and stack pointer registers
EIP	Instruction pointer register

The flags register is a 32-bit register that contains a number of status bits. This register is sometimes referred to as the status register. The following table lists the flags register bits.

Flag Bit	Name
FLCF	Carry flag
FLAF	Auxiliary carry flag
FLZF	Zero flag

Flag Bit	Name
FLSF	Sign flag
FLIF	Interrupt flag
FLTF	Trap flag
FLPF	Parity flag
FLDF	Direction flag
FLOF	Overflow flag

5.3 NLM Compression Tools

NLMPACK is an NLM compression utility that you can use to reduce the disk storage size of your NLM files. NLMPACK can be used for NLMs that run on the NetWare 4.x or later. When the NetWare operating system loads an NLM, it checks to see if the NLM is in compressed format. If the NLM has been compressed, the operating system automatically decompresses it as the operating system loads the NLM into memory.

NLMPACK comes in two versions. NLMPACKP runs in protected mode and NLMPACK runs in real mode. For convenience consider references to NLMPACK to apply to NLMPACKP also.

UNPACK is a utility that unpacks NLMs that were packed using NLMPACK. There are two versions of this utility. UNPACK runs in real mode and UNPACKP runs in protected mode.

NLMPACK and UNPACK Syntax

Usage of NLMPACK is as follows:

```
NLMPACK <source name> <target name>
```

`source name` is the name of the NLM that is to be compressed.

`target name` is the name of the file that the compressed NLM is to be placed in.

NOTE: `target name` can be the same as `source name`.

Usage of UNPACK can be seen by entering "UNPACK" with no parameters at the command line. The usage is as follows:

```
UNPACK <source name> <target name>
```

`source name` is the name of the packed NLM that your want to unpack.

`target name` is the destination file name for the unpacked NLM.

WARNING: `target name` cannot be the same as `source name`.

5.4 MPKXDC

MPKXDC.EXE is a tool that enables the threads within an NLM to be identified as to whether or not they can safely take advantage of multiple processors on the same machine. MPKXDC.EXE is provided in the \TOOLS\ directory of this NDK.

For more specific information, see the following sections:

- “Traditional NetWare and Multithreading” on page 94
- “NetWare 4.11 SMP” on page 94
- “NetWare MPK and Funneling” on page 95

5.4.1 Traditional NetWare and Multithreading

Multithreading (MT) allows an application to do multiple tasks concurrently and is an ideal paradigm for programming to symmetric multiprocessing (SMP) machines. By design the threads model shields the programmer from concerns with the details of parallelism. Moreover, if an MT application is written to correctly to run on a uniprocessor system, it will run correctly on an MP system.

From one viewpoint, all NLMs can be considered multithreaded. However, historically NLMs could not take advantage of multiple processors. Although NetWare has always run fast, the basic architecture was uniprocessor and nonpre-emptive.

In a traditional NetWare, a FIFO scheduling policy does not support priorities or time slicing, Traditional NetWare threads are allowed to run until they block or voluntarily yield the processor—a thread is not be pre-empted (that is, time sliced) under any circumstance. Explicit synchronization with other threads are not needed in this uniprocessor environment because a running thread has exclusive access to the processor. Execution order becomes the synchronizing principle. However, because a traditional NLM depends on execution order to run successfully, unmodified traditional NLMs cannot automatically exploit multiple processors.

Thus, traditional NetWare is a multithreaded environment that supports concurrency but not parallelism.

5.4.2 NetWare 4.11 SMP

NetWare 4.11 SMP was the first attempt by Novell to add symmetric multiprocessing support to NetWare through the addition of an NLM-SMP.NLM. Although this enviornment provided SMP support to NetWare, it remained nonpreemptive. It also provided backward compatibility to NLMs that could not handle multiple processors because of the traditional assumptions about execution order.

NetWare 4.11 SMP introduced the concept of NetWare threads, which operated on traitional assumptions and had to run on Processor 0, and MP threads, which were capable of exploiting parallelism by running on any available processor. By default, all threads created as NetWare threads. NetWare threads had to explicitly call an API function to become an MP threads, and MP threads had to call corresponding migration functions to become NetWare threads.

Using the SMPRPC tool at link time, an NLM developer could declare functions in an NLM to be MT (and thus MP) safe, thus certifing that they could run on any processor. Function not so identified were funneled--put to sleep, migrated to Processor 0, awakened and run with traditional assumptions about execution order, put back to sleep, and migrated to the processor from which they were called. Although funneling acts as a kind of synchronization device, it does not enable execution on any processor. Such enabling requires devices that specifically prevent global variables from being written to by more than one thread at a time or being modified as they are being read. Mutexes are examples of true synchronization devices.

5.4.3 NetWare MPK and Funneling

The NetWare 5.x and later operating system used an approach to multiprocessing that was completely different from NetWare 4.11 SMP—the Multiprocessing Kernel (MPK). Although MPK supports traditional uniprocessor NLMs, MPK was conceived, designed, and built for multiprocessing.

With MPK, the concept of a NetWare thread is replaced by the concept of a Common NetWare Binding (CNB). In the CNB, a thread runs only on Processor 0 using traditional execution order assumptions.Unless otherwise provided for, all threads are created in the CNB. However, it is possible to create threads that have all the synchronization needed for execution on any processor.

If a thread depends on execution order for the successful operatoin, its work obviously cannot be divided among available processors because the order of execution cannot be predicted. Such a thread (or such a function) is called MP unsafe. On the other hand, if a thread succcessfully employs synchronization mechanisms such as mutexes, semaphores, and condition variables, An NLM that exports and uses only threads thus enabled does not need to have its functions funneled to processor 0--because they are properly synchronized through the use of synchronization functions like those provided in the NKS API, such threads can run in parallel on any processor.

The MPKXDC.EXE tool allows the functions in an NLM to be declared MT safe or MT unsafe at link time. For instructions, see Section 6.4, “Using MPKXDC,” on page 98.

NLM Development Tool Tasks

6

This describes common tasks associated with tools used to develop NLMs.

6.1 Using MAKEINIT.EXE

MAKEINIT.EXE is a tool supplied by Novell and used for producing a makeinit file and a comprehensive import file. This tool installs into the .../ndk/nwsdk/tools subdirectory, which should be added to your path setting.

1. Enter makeinit.
2. At the first prompt, enter the path from the root of the appropriate drive to the Watcom compiler, for example, c:\watcom\.
3. At the second prompt, enter the path to the directory that contains subdirectories for import files, include files, tools, etc., for example, c:\novell\ndk\nwsdk\.
4. At the third prompt ("Change input?"), enter Y if you want to change what you entered at the first two prompts, and N if you are satisfied with what you entered.

The utility will create a file simply called makeinit in the ../tools subdirectory. That makeinit file is later referenced as the qmk386.exe utility creates a make file for NLM production

6.2 Building a Symbol File for Novell Remote Debugger

This task is optional.

1. From the menu bar select "Options / Link for NetWare Switches...." Check the Debug Codeview [d codeview opt cvp] box.
2. In Options / Link for NetWare Switches..., select 3 Advanced Switches. Make sure the Produce symbol file[op symf] box is checked.
3. From the menu bar select Options / C Compiler switches..., then select 6 Debugging Switches.

   Click the CodeView debugging format [-hc] radio button.

   If an IDE Request window appears asking Mark all .c files in '...' for remake? click Yes.

   Click OK.
4. Replace CVPACK.EXE found in the WACOM\BINNT directory with CVPACK.EXE version 4.26 from Microsoft (CVPACK /? for version confirmation).

IMPORTANT: The Watcom CVPACK has problems that cause it to fail, so use the Miicrosoft CVPACK version 4.26.

6.3 Building HELLO.NLM with WATCOM WMAKE

1. Add the NWSDK tools directory to your search path. This path will depend on where you installed your copy of the NDK.

The following command works if the NDK components were installed in C:\NOVELL\NDK:

```
SET PATH=%PATH%;C:\NOVELL\NDK\NWSDK\TOOLS
```

This command allows the WATCOM compiler to find the MAKEINIT file. It also allows you to invoke QMK386.EXE (page 79) from the different example directories. These files could just as easily have been copied to another directory already in your search path (C:\DOS, for instance), but NWSDK\TOOLS directory also contains other tools you may eventually need.

IMPORTANT: This step assumes that you have already run the NWSDK\TOOLS\MAKEINIT.EXE utility which generates the MAKEINIT and ALL.IMP files.

---

2. Change your current directory to NWSDK\EXAMPLES\NLM\HELLO.
3. Invoke QMK386 to build a makefile for HELLO.C by entering the following command at the DOS prompt:

   ```
   qmk386 hello
   ```

   This command tells QMK386 to build a makefile for the program called "hello". The following message appears after the makefile is created:

   ```
   Creating Wlink-style MAKEFILE for hello ... done!
   ```

   A makefile called MAKEFILE is created in the current directory.

   These steps demonstrate how to build HELLO.NLM using Novell's libraries (CLIB) and startup files (CLIBPPRE.OBJ).
4. Build HELLO.NLM by entering the following command:

   ```
   wmake
   ```

   At this point you should have a file called HELLO.NLM in the current directory.This file is ready to load and run on a NetWare server.

6.4 Using MPKXDC

MPKXDC.EXE creates an XDC file referenced by the loader in a NetWare 5.x or 6.x server. As an NLM loads, the XDC file specifies four designations for an NLM and its exported API functions:

- NLM is MP safe and can run on any processor.
- Designated functions are MP safe, but all others must be funneled to P0.
- NLM is generally MP unsafe and is limited directly to P0.
- NLM is pre-emptable (caution--all functions become potentially blocking).

(For more information on funneling and exported API functions, see "NetWare MPK and Funneling" on page 95.)

To use MPKXDC, do the following:

1. If some but not all exported API functions in an NLM are MP safe (can be run on any processor), create an APILIST.API file that lists either the API functions that are MP safe or those that are not MP safe.

   This file contains a list of API functions, one per line without leading or trailing white spaces or trailing commas. Lines beginning with # are ignored as comments. Blank lines or lines with leading white spaces are also ignored.

2 Prior to compiling, run MPKXDC.EXE using the following syntax:

```
MPKXDC [option] [APILIST.API] FILENAME.XDC
```

The table following these steps explains available choices for [option].

FILENAME is your name for the file to which MPKXDC.EXE outputs XDC data for reference by your NLM at load time.

3 For the WATCOM WLINK linker, include the following line in the linker directive file used in generating an NLM:

```
OPTION XDCDATA=<FILENAME.XDC>
```

The following options can be specified with the MPKXDC tool.

Option	Explanation
-n	Generates XDC data declaring an entire NLM to be MT safe. Threads created for an NLM thus marked do not start in the CNB and can run on any available processor automatically. All routines and exported API functions are also considered MT safe. All new NLMs should be written for this option. This option is not backward compatible with any NetWare version prior to 5.0. Usage: MPKXDC -n FILENAME.XDC
-f	Generates XDC data declaring only a specified set of API functions in an NLM to be MT unsafe. Thus with this option, only API functions listed are funneled, and all other exported API functions are considered MT safe. This option is not backward compatible with any NetWare version prior to 5.0. Usage: MPKXDC -f APILIST.API FILENAME.XDC
-u	Generates XDC data declaring the NLM to be MT unsafe in general, although it has the effect of declaring that the NLM will deal with multithreading/multiprocessing issues itself. This option is solely for performance needs and is discouraged because it carries many risks. Although using this option and declaring some of the exported functions to be MT unsafe are not mutually exclusive, such a combination can have serious implications for the following reason: Even though a given NLM has -u XDC data, a thread executing in that NLM can be outside the CNB. The thread might have entered the NLM through an exported MT safe API function or might have exited the CNB within that NLM. While outside the CNB, if such a thread were to call an MT unsafe function, results would be unpredictable--for performance reasons, the kernel attempts to short circuit the funneling wrapper while resolving the imports of an NLM that has -u XDC data. In most cases, the -u flag is used to simply indicate that the NLM is legacy and does not contain any MT safe code. With very careful scrutiny, some NLMs may be found to benefit from having some of their exports marked MT safe, if performance considerations demand and sufficient examination reveals that the above listed caveats have been carefully considered. This option is not compatible with -n or -p. Usage: MPKXDC -u FILENAME.XDC
-p	Generates XDC data declaring the NLM as being pre-emptible. This means that any thread executing in the code section of the NLM can be pre-empted (assuming the thread has not programmatically entered a critical section). This has to be used carefully--a nonblocking API function in the NLM may become blocking because a thread executing it can be pre-empted.

Option	Explanation
-h	Displays a help screen for the mpkxdc tool.

Memory Protection Concepts

7

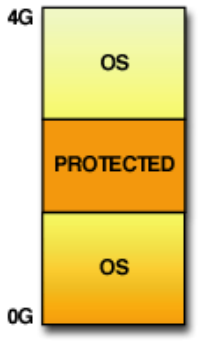
This documentation provides an overview of memory protection in NetWare and explains its use in NLM development.

7.1 NetWare Memory Protection

Memory protection is an integral feature of NetWare 5.x and later that allows an NLM to run in a protected environment. This environment isolates the NLM's memory errors from the operating system and from other running applications. NetWare memory protection can also be used as a valuable development tool.

The foundation of NetWare memory protection is a separation between operating system address space and protected address space. The following illustration represents this separation of memory spaces, although it does not denote the address space locations:

Figure 7-1 *Separation of Protected and OS Memory*



Memory in both the OS address space and in protected address spaces use logical mappings. In addition, NLMs running in protected memory are backed by virtual memory-swapping to and from disk with the appearance of remaining in RAM. (Virtual memory makes the possibility of running out of server memory very unlikely.) Both OS and protected memory also use an improved memory management mechanism that has the effect of defragmenting memory as processes run.

The following is a list of basic features that NetWare memory protection provides:

- Separation of OS address space (running in ring 0) and protected address spaces (running in ring 3)
- Protection of the OS and its resources from NLMs in a protected address space
- Creation of multiple protected address spaces
- Protection of NLMs in one protected address space from NLMs in another
- Checking of all parameters accessing OS resources through a NetWare SDK function
- Fault isolation to one protected address space
- Automatic removal of a protected application (and associated protected address space) that causes a memory fault

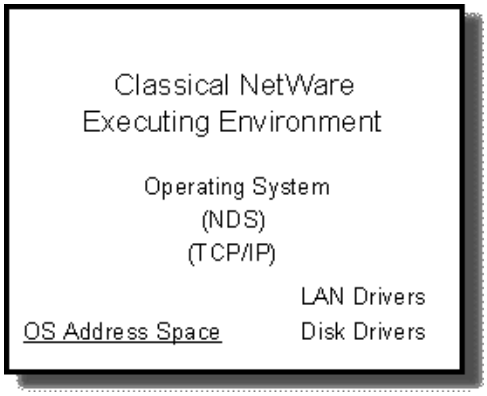
- Optional setting to make the server abend when a faulty parameter is passed to the OS, making debugging more precise

7.1.1 OS Address Space

The OS, hardware device drivers, LAN drivers, and trusted NLM applications run in this space. Applications in OS space have little or no protection from each other.

The NetWare OS has a number of improvements over previous versions, such as a multi-processor aware kernel and native use of IP. However, with regard to memory protection, the OS address space is essentially unchanged, as illustrated:

Figure 7-2 *Classical NetWare Executing Environment*



Regarding memory protection, the OS address space has the following fundamental characteristics:

- Components running in the OS address space are not protected from each other
- Memory has improved logical mapping management and is backed with physical memory
- NLM applications load into this space in the same way as in previous releases
- The environment is still optimized for fast and efficient multi-threaded program execution

7.1.2 Protected Address Spaces

This address space is used for NLMs that are being tested by developers or that have not run in a stable condition for long enough to be completely trusted by system administrators. Multiple protected address spaces can be created, but only one protected address space can have access to OS functionality at one time.

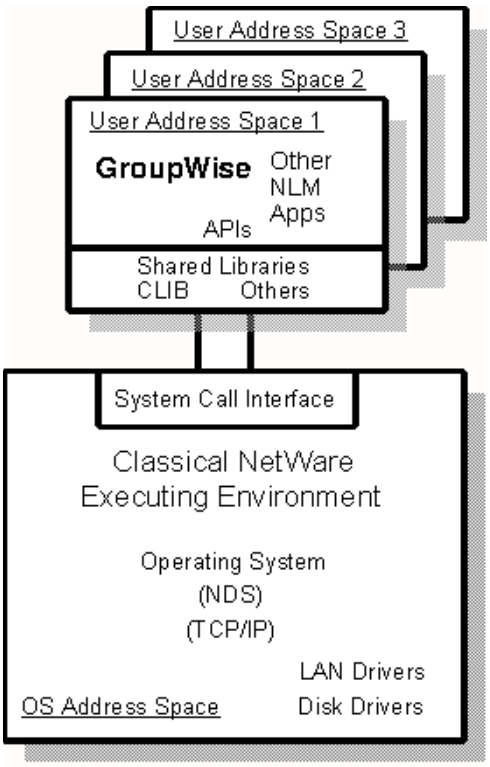
Protected address spaces are new with NetWare, and have the following basic features:

- A protected address space is an insulated functional unit. If a process in one protected address space causes a memory fault, processes in the OS address space and other protected address spaces continue to run normally
- Multiple protected address spaces can be created, but only one protected address space can actively have control of a processor at one time. However, on an multiprocessor machine processes running on different CPUs can be running in different address spaces.
- Applications in a protected space have immediate access to all services provided directly by libraries loaded in that protected address space, including CLIB and other shared libraries.

- NLM applications in a protected address space can have indirect access to functions in the OS address space through use of functions in the NetWare SDK.
- All access from a protected space to the OS is through a system call interface. This interface verifies the validity of function parameters through marshaling code written for each currently supported NetWare API function.
- Because of marshaling, NLMs running in protected address space do not run as fast as NLMs running in the OS address space. The percentage of performance loss depends on the number and kind of OS functions called.
- Virtual Memory technology provides greatly expanded total server memory.

The following illustration represents the basic relationship between protected address spaces and the OS address space:

Figure 7-3 *The Relationship between Protected Address Spaces and the OS Address Space*

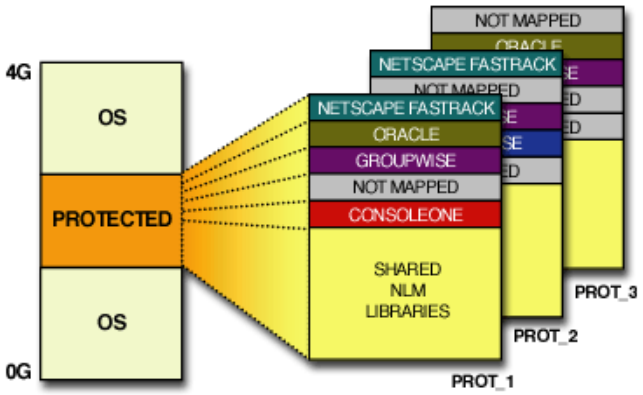


Multiple Protected Address Spaces

NetWare memory protection allows for an indefinite number of protected address spaces. This is possible because the NetWare OS uses logical memory mappings and provides access to Virtual Memory.

Physically, all protected address spaces occupy the same location on the server. In addition, logical mappings of all protected spaces occupy the same address ranges, as illustrated below:

Figure 7-4 *Location of Protected Address Spaces on the Server*

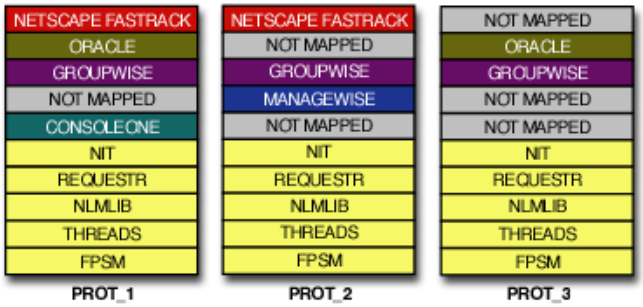


When an application in a protected address space calls a function in a shared library, the library is loaded into the protected address area on the server, and the calling protected address space is mapped to the library. That same range of protected memory is reserved on all other protected address spaces. However, that space is mapped and thus active only in the protected address space that called the library function. Libraries are set up in other protected address spaces as they make calls to library functions.

Likewise, when an NLM application is loaded into a protected address space, the calling address space is mapped to the NLM. The memory range in which the NLM is running is also reserved in all protected address spaces. However, other protected address spaces get access to the loaded NLM only by loading it explicitly, rather than by merely calling one of its functions as with a shared library. Each protected address space that loads the NLM is mapped to the memory range in which the NLM is running.

The following illustration represents two protected address spaces using a variety of NLM libraries and applications. Both are mapped to the libraries CLIB, NLMLIB, and REQUESTR. Only USER_ADDRESS_SPACE1 is mapped NLM B and NLM C. Only USER_ADDRESS_SPACE2 is mapped to NLM A and NLM F. Both are mapped to NLM D, and neither is mapped to an implied NLM E. (NLM E is mapped to USER_ADDRESS_SPACE3.)

Figure 7-5 *Mappings of Protected Address Spaces*



What Loads Automatically into NetWare Protected Address Space

When an address spaces is created by the NetWare 5.x or 6.x OS, a library called userlib.nlm is automatically load into (are mapped to) that space. This library contains some of the OS functions

that are most commonly used, such as those to compare strings. Having those functions readily accessible saves on the overhead of crossing the boundary between a protected address space and the OS address space.

In addition, the CLIB family of libraries loads into protected address space when an NLM loads. After CLIB has loaded into protected space, there are two copies running-one in the OS address space and another in the protected address space. All protected address spaces have access to the same copy of CLIB through protected address space mappings.

NLMs can also be loaded into one or more protected address spaces, but they must be loaded explicitly.

What Applications Can Run in NetWare Protected Space?

Access to functions in the OS address space is provided to NLM applications in the protected address space through the libraries of functions in the NetWare API functions in the NDK. Use only functions provided through those libraries for OS space functionality. OS functions are subject to change without notice or publication, and SDK functions are updated to adapt to changes in the OS. NLMs that are written with functions the NetWare SKD and that follow specifications published with the SDK can be loaded with confidence into protected address space.

This includes NLMs written with functions provided through the CLIB family of libraries, automatically loaded with CLIB.NLM. It also includes NLMs written with the multi-platform functions ported to the NLM platform through CALNLM32.NLM.

7.1.3 System Call Interface

The system call interface is a gate that verifies the validity of parameters into the OS address space. If an invalid parameter is passed, the passing NLM is unloaded without terminating normally, its associated protected address space is deleted, and the memory that the address space occupied is released.

The verification takes place through marshaling code that validates parameters of each fully released NetWare API function, based on two criteria:

- Memory accessed
- Values passed

Addressed memory is verified for validity and length. For example, if an input parameter points to an array of structures, the interface verifies that the starting address is valid and that enough memory is allocated for the array.

Values passed are also verified for validity. For example, if a function needs to write the contents of a buffer to a screen, the interface verifies that the specified screen is on the list of valid screens and that the calling NLM application has access to that screen.

In addition, the interface also verifies the validity of values resulting from operations. For example, if a process were to attempt to divide by zero, the parameter value would be declared invalid.

7.1.4 Memory Protection set Parameters

You can use the following set parameters to control the behavior of memory protection in a 5.x or 6.x server:

- set memory protection fault cleanup

  This parameter allows for cleanup after at attempt to violate memory protection. If set to ON, on an attempt to violate memory protection, the offending protected address space and its loaded NLMs are removed from protected memory and their resource are freed and returned to the system. If set to OFF, no effort is made to handle the fault, and the situation is left to the abend recovery mechanism.

- set memory protection no restart interval

  This parameter prevents a protected address space from restarting if it faults more than once during a specified number of minutes. If set to 0, this parameter is itself disabled. If set to a whole number from 1 to 60, on a fault the offending protected address space is removed along with any loaded NLMs, and the resources associated with that address space are returned to the system. Then a new protected address space with the same name is created, and the same NLMs are loaded into the new space. Thereafter, if the protected address space is restarted more recently than the set number of minutes, the restart feature is disabled.

- set auto restart after abend

  Controls server behavior following and abend. Values are as follows (configure restart times for options 1 and 2 with set auto restart after abend delay time:

  - 0-do not attempt to recover from abend
  - 1-(default) For software abends, NMIs, and machine check exceptions - attempt to recover from the problem, down the server in the configured amount of time, then restart the OS. For other exceptions abends - suspend the faulting process and leave the server up.
  - 2-For all software and hardware abends, attempt to recover from the problem, down the server in the configured amount of time, then restart the OS.
  - 3-For all software and hardware abends, do an immediate restart of the server. (Detailed information about server abends is logged to the abend.log file in the sys:system directory. Note that an abend indicates that the server is not in a valid state, and recovery is not always possible. This option is disabled when the developer option settable parameter is set to ON.)

Memory Protection Tasks

8

This documentation explains how to perform some fundamental tasks to use NetWare 5.x or later memory protection as aids in NLM development.

8.1 Loading an NLM into OS Address Space

Loading NLMs into OS address space does not change with the NetWare 5.x or 6.x OS. The OS address space is the default load location.

1 To load an NLM into OS address space, at the server prompt or in a .ncf file enter the following command:

```
load nlm_a
```

where *nlm_a* is the correct name of the NLM you are loading.

When this command executes, the specified NLM loads in the OS address space and has direct access to the OS. OS memory has little or no protection from NLMs running in the OS address space.

8.2 Loading an NLM into a Protected Address Space

You can load an NLM into protected a address space by using the system default or by loading into an explicitly named protected address space.

1 To load *nlm_a*.nlm into a default protected address space, at the server console type

```
load protected nlm_a
```

where *nlm_a* is the correct NLM name.

The NetWare 5.x or 6.x OS automatically creates USER_ADDRESS_SPACEX. (X is incremented as spaces are created. The first default protected address space created is USER_ADDRESS_SPACE1, the second is USER_ADDRESS_SPACE2, and so forth.)

You can override the system default naming scheme and load an NLM into an explicitly named protected address space.

1 To load *nlm_a*.nlm into a protected address space *space_a*, at the server console type

```
load address space = space_a nlm_a
```

where *space_a* and *nlm_a* are the correct protected address space and NLM names.

With a protected address space name set in the command, the system automatically loads the NLM into protected memory. If protected address space *space_a* does not currently exists, the system creates it and loads nlm_a.nlm into it. If *space_a* exists, the system loads *nlm_a*.nlm into that space.

8.3 Unloading NLMs Protected Address Spaces

The unload command unlinks a loadable module previously linked to the OS with the load command. In the NetWare 5.x or 6.x OS unload also allows you to unload an NLM from a specific protected address space or to unload the entire protected address space.

- To unload an NLM, at the server prompt type

  `unload nlm_a`

  where *nlm_a* is the correct NLM name.
- To unload an NLM from a specific protected address space, at the server prompt type

  `unload address space = space_a nlm_a`

  where *space_a* and *nlm_a* are the correct protected address space and NLM names.
- To unload a protected address space (and all NLMs and libraries running in that protected address space), at the server prompt type

  `unload address space = space_a`

  where *space_a* is the correct protected address space name.

8.4 Using the protection Command

The protection command allows you to check the protection status of any or all protected address spaces, including all applications loaded into that space and any protection options currently set. It also allows you to enable or disable the restart feature for a protected address space.

8.4.1 Checking Protection Status

1. To check the protection status of any or all protected address spaces, type one of the following at the server console:

   ```
   protection
   ```

   (Displays protection status for all protected address spaces)

   ```
   protection space_a
   ```

   (Displays protection status only for protected address space "space_a" and creates that protected address space if it does not already exist)

8.4.2 Enabling/Disabling the Restart Feature

1. To enable or disable the restart feature for a protected address space, type one of the following at the server console:

   ```
   protection restart space_a
   protection no restart space_a
   ```

8.5 Finding Out What is Running in a Protected Address Space

1. Use the protection command as described in the first section of Section 8.4, “Using the protection Command,” on page 108

8.6 Setting a Protected Address Space to Restart after a Fault

By default, when a memory fault occurs, the server removes the NLM that caused the fault, deletes the protected address space (and all its contents), and releases the memory used by the protected address space. However, the server can be set to restart the protected address space after a fault occurs. This setting can be chosen when the NLM is loaded, when a protected address space is created explicitly, or when processes in a protected address space are running.

- To flag a default protected address space as restartable at load time, at the console type

  `load restart nlm_a`

  where *nlm_a* is the correct NLM name.

  The system creates a new protected address space with the next number in the default naming scheme and loads nlm_a.nlm into it.

- To flag an explicitly named protected address space as restartable when you are loading an NLM into it, at the console type

  `load address space = space_a restart nlm_a`

  where *space_a* and *nlm_a* are the correct protected address space and NLM names.

  If protected address space "space_a" exists, the system loads nlm_a.nlm into it and flags the protected address space as restartable. Otherwise, the system creates protected address space "space_a," loads nlm_a.nlm into it, and flags the address space as restartable.

- To flag an existing protected address space as restartable as processes are running or to create a restartable new address space, at the console type

  `protection restart space_a`

  where *space_a* is the correct protected address space name.)

  If protected address space "space_a" exists, the system flags it as restartable. Otherwise, the system creates protected address space "space_a" and flags it as restartable.

- To flag an existing protected address space as nonrestartable as processes are running or to create a restartable new address space, at the console type

  `protection no restart space_a`

  where *space_a* is the correct protected address space name.

  Type help protection at the console screen for more information.

8.7 Setting a Server to Abend for Memory Faults

1. To set the server to abend on a memory fault, at the server prompt or in a .ncf file enter the following commands:

   ```
   set Memory Protection Fault Cleanup = off
   set Auto Restart After Abend = 0
   ```

By default, when a memory fault occurs in a protected address space, a NetWare server shuts down the calling NLM and protected address space without normal termination, releases the memory held by the protected address space, and continues other operations. However, you can set the server to abend immediately at a memory fault. Such an abend allows immediate inspection of the running server, and can thus help greatly with debugging.

To restore the memory fault recovery feature, set Memory Protection Fault Cleanup back to on and set Auto Restart After Abend to 2.

8.8 Loading Memory Fault Isolation

1 At the server console prompt type the following (where someNLM is the NLM you are loading):

LOAD -m someNLM

The -m loader option is a new development tool with NetWare 5, and can be extremely useful in tracking down causes of memory corruption. The option results in each memory allocation being put onto one or more full pages with a guard page at the border of the upper address of the allocated memory. If the memory is overrun and touches the guard page, a page fault occurs, causing a server abend on the instruction that overran the memory.

NOTE: In order for the server to abend while an NLM is running in a protected memory space, system fault protection must be turned off. See Section 8.7, "Setting a Server to Abend for Memory Faults," on page 109.

8.9 Pinpointing Memory Overflows

NetWare memory protection offers two debug options that greatly reduce the time and effort required to find the cause of a memory violation. Both provide a mechanism that causes a server abend precisely at the instruction on which the initial memory violation occurs. This can save hours of searching for something that might have happened five seconds, five minutes, or five hours before the corruption would otherwise become apparent because an abend.

NetWare memory protection provides the options for the following important memory violation problems:

Most server memory problems occur within the normal operation of an NLM, although the violation might not be apparent until an unrelated operation tries to access the corrupted memory considerably after the corruption took place. For instructions on setting the server to abend when normal allocated memory in a specified NLM is overrun, see Section 8.8, "Loading Memory Fault Isolation," on page 110.

8.10 Accessing On-Line Help for Memory Protection

1 To access on-line help for command options and syntaxes, type help followed by the relevant command. For example:

```
help protection
help load
help unload
```

Advanced NLM Function Concepts

9

This documentation describes Advanced Services, its functions, and features.

9.1 Advanced Function List

Function	Description
AllocateDynArrayEntry (page 118)	Allocates an entry in a dynamic array.
AllocateGivenDynArrayEntry (page 120)	Allocates an entry in a dynamic array at a given element index.
AllocateResourceTag (page 122)	Allocates a resource tag for a particular resource.
AsyncRead (page 124)	Allows a file to be read directly from cache memory.
AsyncRelease (page 126)	Releases the cache buffer memory allocated by a call to AsyncRead.
CancelNoSleepAESProcessEvent (page 127)	Cancels a scheduled AES (Asynchronous Event Scheduler) event.
CancelSleepAESProcessEvent (page 128)	Cancels a scheduled AES event.
DeallocateDynArrayEntry (page 129)	Frees the dynamic array entry at a specified index.
GetFileHoleMap (page 130)	Returns a block allocation map for a file.
GetSetableParameterValue (page 132)	Obtains the value of a server parameter.
GetThreadDataAreaPtr (page 133)	Returns the thread switch data area pointer for the current thread.
gwrite (page 134)	Writes multiple buffers to a file.
ImportSymbol (page 136)	Returns a pointer to an exported symbol.
NWAddSearchPathAtEnd (page 137)	Adds a search path to the end of the search path list that the OS uses to determine the location of NLM applications.
NWGarbageCollect (page 139)	Unfragments freed server memory.
NWDeleteSearchPath (page 138)	Deletes a search path from the search path list that the OS uses to determine the location of NLM applications.
NWGetSearchPathElement (page 140)	Returns a search paath from the search path list that the OS uses to determine the location of NLM applications.
NWInsertSearchPath (page 141)	Inserts a search path into the search path list that the OS uses to determine the location of NLM applications.
qread (page 142)	Performs a low-overhead read operation.

Function	Description
qwrite (page 144)	Performs a low-overhead write operation.
RegisterConsoleCommand (page 146)	Registers a console command parsing function.
RegisterForEvent (page 148)	Registers to be notified when a particular event occurs.
SaveThreadDataAreaPtr (page 153)	Sets the thread switch data area pointer for the current thread.
ScanSetableParameters (page 154)	Returns information about server parameters.
ScheduleNoSleepAESProcessEvent (page 158)	Defines a procedure to be called by the asynchronous event scheduler (AES) after a specified delay.
ScheduleSleepAESProcessEvent (page 160)	Defines a procedure to be called by the AES after a specified delay.
SetSetableParameterValue (page 162)	Sets the value of a server parameter.
SynchronizeStart (page 163)	Restarts the NLM start-up process when using synchronization mode.
UnimportSymbol (page 164)	Eliminates the dependency of an NLM on a specified external symbol.
UnRegisterConsoleCommand (page 165)	Cancels a registered console command parsing function.
UnregisterForEvent (page 166)	Cancels a registration for event notification.

9.2 Functions to Handle Dynamic Arrays

The dynamic array functions, listed below, assist the developer in handling arrays that grow dynamically.

- AllocateDynArrayEntry (page 118)
- AllocateGivenDynArrayEntry (page 120)
- DeallocateDynArrayEntry (page 129)

These functions perform some of the housework necessary when expanding in-memory tables.

For example, a database server NLM might maintain a connection table where each entry contains information about one of the server's clients. To avoid limiting the database server to an arbitrary maximum number of clients it can service, this connection table expands whenever new clients are added. Dynamic array functions perform some of the housework for this expansion.

9.3 Dynamic Array Terminology

Dynamic array terms are defined as follows:

entry

Refers to an element in the dynamic array. For example, consider a dynamic array consisting of 20 bytes. If the element size is 4 bytes, then there are 5 entries in the array. Element and entry are used interchangeably. Entries may or may not be in use.

dynamic array

An array whose number of enties can be increased dynamically by the user as needed at run time.

DAB (dynamic array block)

A structure used to control a dynamic array. Every dynamic array must have a DAB associated with it.

grow amount

The user specifies the number of elements to add to the dynamic array when AllocateDynArrayEntry (page 118) is invoked and there are no unused array elements. For example, if the grow amount is 5, the first call to AllocateDynArrayEntry produces an array of 5 elements. The next call increases the number of elements to 10 only if there are no unused array elements in the first 5. The grow amount is not used with AllocateGivenDynArrayEntry (page 120).

reallocation function

This function is used to reallocate memory for use by the dynamic array, and must allow resizing. Currently, only realloc allows resizing. However, users can write their own resizing memory allocation function, as long as the number and definition of parameters is the same as for realloc.

9.4 Dynamic Linkage of Exported Symbols

ImportSymbol (page 136) and UnimportSymbol (page 164) allow you to link and unlink exported module symbols dynamically. Any symbol exported by an NLM may be imported dynamically by another NLM by calling ImportSymbol. This function is especially useful for creating an NLM that doesn't fully rely on a symbol or set of symbols, but can have enhanced functionality if those symbols are present. It is also useful for creating NLM applications that can load on multiple versions of the server, and can take advantage of features that are present in one version but not the other.

The function uses the "handle" of the NLM importing the symbol, and the name (ASCIIZ string) of the symbol being imported. If successful, the function returns the address of the symbol. The module dependency list maintained by the OS reflects the NLM depencency on that symbol. If the symbol is not available for import, the function returns NULL.

Once the symbol is imported, the NLM may freely call or access the symbol as if it had been statically imported at load time. Symbols may be imported from the OS itself or from other NLM applications that have exported symbols.

The reverse of importing symbols is also possible. UnimportSymbol tells the OS that the NLM no longer needs the specified symbol. If UnimportSymbol is successful, the OS removes the NLM depencency on that symbol. This allows the NLM from which the symbol was dynamically imported to unload, providing no other dependencies exist on either the NLM as a whole or any other symbols it exports.

NOTE: If a symbol is un-imported, it must not be accessed.

The return type of ImportSymbol is a void pointer. Generally, you can assign the return value of data symbols to any pointer to object type, although you should be careful to access the data object in ways that are consistent with the type it really is.

It is *not* as generally acceptable in STD C to typecast a void pointer as a function pointer, and you should be careful about this operation in your source code.

9.5 Event Reporting and Management Functions

Use the event reporting functions to obtain and set the thread data area pointer for the current thread, to perform and cancel event notification, and to restart the NLM startup process when using synchronization mode. These functions are listed below:

- GetThreadDataAreaPtr (page 133)
- RegisterConsoleCommand (page 146)
- SaveThreadDataAreaPtr (page 153)
- SynchronizeStart (page 163)
- UnRegisterConsoleCommand (page 165)

NLM applications that manage events can use event management functions to allocate resource tags and process events. Event management functions are listed below:

- AllocateResourceTag (page 122)
- CancelNoSleepAESProcessEvent (page 127)
- CancelSleepAESProcessEvent (page 128)
- ScheduleNoSleepAESProcessEvent (page 158)
- ScheduleSleepAESProcessEvent (page 160)

9.6 File I/O Functions

NLM applications that need faster access to files and information about sparse files can use File I/O functions.

- AsyncRead (page 124)
- AsyncRelease (page 126)
- GetFileHoleMap (page 130)
- gwrite (page 134)
- qread (page 142)
- qwrite (page 144)

Advanced Tasks

10

This documentation describes common tasks associated with Dynamic Array functions in the Advanced function group.

10.1 Using Dynamic Array Functions

When using dynamic array functions, do the following:

1. Create a data structure called a dynamic array block (DAB).

   This structure describes information table, such as entry types and expansion parameters.

2. Use the DAB as an input parameter to one of two functions called whenever the table must expand.

   Which function is used depends on how the indexes to entries in the dynamic array are generated. Initially a dynamic array has no entries. In the database server example, the dynamic array would expand every time a new client requests service.

3. Call another function when a dynamic array entry is no longer being used, so that entry can be reused.

10.2 Generating Dynamic Array Indexes

1. Generate indexes in one of the two following ways:
   - Call AllocateDynArrayEntry (page 118), which generates the index for a new entry.
   - Call AllocateGivenDynArrayEntry (page 120) to specify which index to use when allocating a new entry in a dynamic array.

Advanced Functions

11

This documentation alphabetically lists the advanced functions and describes their purpose, syntax, parameters, and return values.

AllocateDynArrayEntry

Allocates an entry in a dynamic array

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwdynarr.h>

int AllocateDynArrayEntry  (
   T_DYNARRAY_BLOCK    * dabP);
```

Parameters

dabP

(IN) Points to the T_DYNARRAY_BLOCK structure containing the Dynamic Array Block (DAB).

Return Values

This function returns the index of the entry (a value of 0 or greater) if successful. Otherwise, it returns an error code:

-1	EFAILURE

Remarks

Call the AllocateDynArrayEntry function to allocate additional entries in the dynamic array. The dynamic array can be increased in size, but not decreased.

`DABarrayP` is the pointer to the dynamic array. It is referenced as `varName.DABarrayP`. To reference an entry of the dynamic array, the expression, `varName. DABarrayP` [index] is used. If the entry is a structure, one of its fields can be referenced as `varName. DABarrayP` [index]. `field`.

`DABrealloc` is the address of the desired memory allocation function which must allow resizing. This function is normally realloc, but it can be a developer-defined function.

`DABgrowAmount` is the number of elements by which to increase the dynamic array when more elements are needed.

The DAB structure can be declared and initialized by standard C methods, or the following macro can be used:

```
GEN_DYNARRAY_BLOCK(  elementType,  varName, defDec )
```

Where `elementType` is the C type of the element, such as int, struct, and so on. `varName` is the name of the variable declared as a dynamic array. `defDec` can be DECLARE, DEFINE, or INIT, as follows:

DELARE

Declares the `varName` as the type of DAB specified. Generates the following:

```
struct varName##Struct varName
```

DEFINE (realloc , growAmount)

Defines and initializes `varName` as the type of DAB specified. Generates the following:

```
struct varName##Struct
{
   elementType    *DABarrayP;
   int            DABnumSlots;
   int            DABelementSize;
   void           *(*DABrealloc) (void *, size_t);
   int            DABgrowAmount;
   int            DABnumEntries;
} varName = {NULL, 0, elementSize, realloc, growAmount, 0}
```

INIT (realloc , growAmount)

Initializes an already-defined DAB. Generates the following:

```
struct varName##Struct varName =
        {NULL, 0, elementSize, realloc, growAmount, 0}
```

The parameters for DEFINE and INIT are as follows:

realloc

Specifies the reallocation function to use when expanding the dynamic array. Normally, this would be realloc.

growAmount

Specifies the amount to expand the dynamic array by if AllocateDynArrayEntry expands the array.

See Also

AllocateGivenDynArrayEntry (page 120), DeallocateDynArrayEntry (page 129)

AllocateGivenDynArrayEntry

Allocates an entry in a dynamic array at a given element index

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwdynarr.h>

int AllocateGivenDynArrayEntry  (
   T_DYNARRAY_BLOCK   *dabP,
   int                 ndx);
```

Parameters

dabP

(IN) Points to a pointer to the Dynamic Array Block (DAB).

ndx

(IN) Specifies the desired 0-based element index into the dynamic array.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code:

Value	Name	Description
5	ENOMEM	Not enough memory.

Remarks

Use the AllocateGivenDynArrayEntry function to allocate additional entries in the dynamic array.

- The array can be increased in size, but not decreased.
- If the index goes beyond the number of elements in the array, the array is expanded to accommodate it. All intermediate entries are allocated and marked as available.
- If an in-use memory block already exists at the specified index, it is overwritten, and 0 is returned.

DAB is a structure with the following elements:

```
elementType *DABarrayP;
      /* elementType = int,struct,typedef,..*/
int    DABnumSlots;
int    DABelementSize;                  /* user-supplied */
```

```
void   *(*DABrealloc) (void *, size_t);  /* user-supplied */
int    DABgrowAmount;                    /* user-supplied */
int    DABnumEntries;
```

`DABarrayP` is the pointer to the dynamic array. It is referenced as varName.DABarrayP. To reference an entry of the dynamic array, the expression, varName.DABarrayP[index] is used. If the entry is a structure, one of its fields can be referenced as varName.DABarrayP[index].field.

`DABrealloc` is the address of the desired memory allocation function which must allow resizing. This function is normally realloc, but it can be a user-defined function.

`DABgrowAmount` is ignored for this function.

This structure can declared and initialized by standard C methods, or the following macro can be used:

```
GEN_DYNARRAY_BLOCK( elementType, varName, defDec )
```

See AllocateDynArrayEntry (page 118) for more detailed information about this macro.

See Also

AllocateDynArrayEntry (page 118), DeallocateDynArrayEntry (page 129)

AllocateResourceTag

Allocates a resource tag for a particular resource

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

LONG AllocateResourceTag  (
   LONG    NLMHandle,
   BYTE   *descriptionString,
   LONG    resourceType);
```

Parameters

NLMHandle

(IN) Specifies the handle of the NLM™ application for which a resource tag is desired; the NLM handle is obtained by calling GetNLMHandle.

descriptionString

(IN) Points to a string describing the resource tag.

resourceType

(IN) Specifies the type of resource tag desired.

Return Values

This function returns a resource tag if successful or a value of 0 if not successful.

Remarks

The resource tag is used as a parameter to other function calls which allocate resources, and in turn, used by the NetWare® Resource Management System. A list of resource types or resource tag signatures can be found in nwadv.h:

Resource Type	For Use With
AESProcessSignature	ScheduleNoSleepAESProcessEvent
	ScheduleSleepAESProcessEvent
ConsoleCommandSignature	RegisterConsoleCommand

See Also

alloca, __qmalloc (*NDK: Program Management*), GetNLMHandle (*NDK: NLM Threads Management*)

AsyncRead

Reads a file directly from cache memory

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwfileio.h>

int AsyncRead  (
   int                          handle,
   LONG                         startingOffset,
   LONG                         numberOfBytesToRead,
   LONG                        *numberOfBytesActuallyRead,
   LONG                         localSemaphoreHandle,
   T_cacheBufferStructure     **cacheBufferInformation,
   LONG                        *numberOfCacheBuffers);
```

Parameters

handle

(IN) Specifies a handle of the file from which data is to be read.

startingOffset

(IN) Specifies the offset in the file from which the first byte is to be read.

numberOfBytesToRead

(IN) Specifies the number of bytes to read from the file.

numberOfBytesActuallyRead

(OUT) Points to the number of bytes actually read from the file.

localSemaphoreHandle

(IN) Specifies this is used by the AsyncRead Event Service Routine (ESR) to signal completion of all of the requested cache reads for a particular call to AsyncRead. A local semaphore handle is obtained by calling OpenLocalSemaphore. Either WaitOnLocalSemaphore or ExamineLocalSemaphore should be used to determine when the ESR has signalled the semaphore.

cacheBufferInformation

(OUT) Points to an array of structures which contain the cache buffer pointers, lengths, and completion codes.

numberOfCacheBuffers

(OUT) Points to the number of cache buffers required to perform the file read.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

NOTE: If this function returns an error, do not wait on the local semaphore passed. AsyncRead does not signal the semaphore when the function fails.

Remarks

This function reads data from a file and returns pointers to the cache buffers which contain the requested data. The requested data can then be read directly from the cache buffers.

AsyncRead now reads only 64K at a time.

The cache buffer structure has the following form:

```
typedef struct cacheBufferStructure
   {
   char    *cacheBufferPointer;
   LONG    cacheBufferLength;
   int     completionCode;
   } T_cacheBufferStructure;
```

The `cacheBufferPointer` field is the address of the first character for that particular cache buffer. The `cacheBufferLength` field is the number of bytes to be used from that cache buffer. The `completionCode` field is the NetWare error code for that particular cache buffer read operation.

AsyncRelease must be called to release the memory allocated by AsyncRead.

See Also

AsyncRelease (page 126), GetNLMHandle, OpenLocalSemaphore, WaitOnLocalSemaphore (*NDK: NLM Threads Management*)

AsyncRelease

Releases the cache buffer memory allocated by a previous call to AsyncRead

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwfileio.h>

void AsyncRelease  (
   T_cacheBufferStructure   *cacheBufferInformation);
```

Parameters

cacheBufferInformation

(IN) Points to the address of the start of a cache buffer list returned by a call to AsyncRead.

Remarks

It is the responsibility of the user to free the cache memory created by an AsyncRead call if an NLM should terminate before AsyncRelease is called. The atexit, AtUnload, and signal functions can be used to handle this situation. Note that _exit, by definition, does not call atexit, although exit does call atexit.

See Also

AsyncRead (page 124), atexit, AtUnload, signal (*NDK: NLM Threads Management*)

CancelNoSleepAESProcessEvent

Cancels a previously scheduled event

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwthread.h>

extern void CancelNoSleepAESProcessEvent  (
   struct AESProcessStructure   *EventNode);
```

Parameters

EventNode

(IN) Points to an AESProcessStructure which describes the event to be cancelled.

Remarks

The `EventNode` should have been used in a previous call to ScheduleNoSleepAESProcessEvent.

See Also

CancelSleepAESProcessEvent (page 128), ScheduleNoSleepAESProcessEvent (page 158)

CancelSleepAESProcessEvent

Cancels a previously scheduled event

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwthread.h>

extern void CancelSleepAESProcessEvent  (
   struct AESProcessStructure   *EventNode);
```

Parameters

EventNode

(IN) Points to an AESProcessStructure which describes the event to be cancelled.

Remarks

The `EventNode` should have been used in a previous call to ScheduleSleepAESProcessEvent.

See Also

CancelNoSleepAESProcessEvent (page 127)

DeallocateDynArrayEntry

Frees the dynamic array entry at the specified index

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwdynarr.h>

int DeallocateDynArrayEntry  (
   T_DYNARRAY_BLOCK    *dabP,
   int                  ndx);
```

Parameters

dabP

(IN) Points to a Dynamic Array Block (DAB), as described for the function AllocateDynArrayEntry.

ndx

(IN) Specifies the desired 0-based element index into the dynamic array.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns:

Value	Name	Description
-1	EFAILURE	Index exceeds limit or element is already deallocated.

Remarks

The specified element's space is not freed; it is just marked as available.

See Also

AllocateDynArrayEntry (page 118), AllocateGivenDynArrayEntry (page 120)

GetFileHoleMap

Returns a block allocation map for a file

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int GetFileHoleMap  (
   int     handle,
   LONG    startingPosition,
   LONG    numberOfBlocks,
   BYTE   *replyBitMapP,
   LONG   *allocationUnitSizeP);
```

Parameters

handle

(IN) Specifies the pertinent file handle.

startingPosition

(IN) Specifies the 0-based byte offset into the file.

numberOfBlocks

(IN) Specifies the number of file blocks required for a given file. This indirectly specifies the size in bytes of `replyBitMapP`. This can be computed by knowing the file size and the size of a block:

`numberOfBlocks` = (file-size) / (bytes-per-block) bytes-per-block = 8 x (sectors-per-block)

Use filelength and GetVolumeInformation to get the information to compute the number of blocks and round up, if necessary. The number specified must be in 4-byte increments: 4, 8, 12, etc.

replyBitMapP

(OUT) Points to a block of memory that should be considered as a bit-stream. If the bit is set, then the file block is allocated. If it is cleared, then the file block is not allocated.

allocationUnitSizeP

(OUT) Points to the size of each block in bytes.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

Remarks

The `startingPosition` and `numberOfBlocks` specify which part of the file to return information about.

GetSetableParameterValue

Obtains the value of a NetWare server console parameter

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

LONG GetSetableParameterValue  (
   LONG   connectionNumber,
   BYTE  *setableParameterString,
   void  *returnValue);
```

Parameters

`connectionNumber`

(IN) Specifies the connection number of the user who wants to obtain information about server console parameters.

`setableParameterString`

(IN) Points to a NULL-terminated ASCIIZ string representing the name of the server console parameter.

`returnValue`

(OUT) Points to the value of the server console parameter.

Return Values

Returns 0 if successful, or -1 if an invalid setable parameter string was specified.

Remarks

A setable parameter is a NetWare OS parameter that can be set using the SET console command (see the *Utilities Reference* included with the NetWare 5 release).

The value returned in the `returnValue` parameter depends upon the server console parameter passed into the `setableParameterString` parameter. Enough space should be set aside for the return value to be copied to the destination address pointed to by the `returnValue` parameter. The maximum size of a server console parameter is 512 bytes.

See Also

ScanSetableParameters (page 154), SetSetableParameterValue (page 162)

GetThreadDataAreaPtr

Gets the thread switch Data Area Pointer for the current thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

void *GetThreadDataAreaPtr  (void);
```

Return Values

This function returns the thread switch pointer for the current thread.

Remarks

When thread-switch event reporting has been registered, the Data Area Pointer is passed as the parameter to the report routine when a thread switch occurs.

The pointer can point to any user-defined data structure.

See Also

SaveThreadDataAreaPtr (page 153), RegisterForEvent (page 148)

gwrite

Writes multiple buffers to a file with a single call

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwfileio.h>

int gwrite  (
   int                          fildes,
   T_mwriteBufferStructure     *bufferP,
   LONG                         numberOfBuffers,
   LONG                        *numberOfBuffersWritten);
```

Parameters

handle

(IN) Specifies the handle of the file to which data is to be written.

bufferP

(IN) Points to an array of structures of type T_mwriteBufferStructure. Each structure contains a pointer to the buffer to be written and the number of bytes to be written.

numberOfBuffers

(IN) Specifies the number of structures in `bufferP`.

numberOfBuffersWritten

(OUT) Points to the number of buffers actually written.

Return Values

On success, returns the number of bytes written. On failure, returns EFAILURE and sets errno and NWErrno to EBADF for a bad file handle or to other error codes as appropriate.

Remarks

The `bufferP` structure is defined in nwadv.h as:

```
char *mwriteBufferPointer
LONG mwriteBufferLength
int  reserved
```

See Also

qwrite (page 144)

ImportSymbol

Returns a pointer to an exported symbol

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 3.2, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

void *ImportSymbol  (
   int    NLMHandle,
   char  *symbolName);
```

Parameters

`NLMHandle`

(IN) Specifies the handle of the NLM that requires the symbol.

`symbolName`

(IN) Points to the symbol to import.

Return Values

Returns a pointer to the function associated with the symbol upon success. Otherwise, it returns 0.

Remarks

ImportSymbol is useful for resolving external symbol references that do not exist when the NLM requiring those symbols loads. For example, if an NLM calls Network Management functions, that NLM can test whether the needed Network Management symbols are available.

The `NLMHandle` parameter can be obtained by calling GetNLMHandle.

See impsymbl.c (../../../samplecode/clib_sample/nlm/impsymbl/impsymbl.c.html).

See Also

GetNLMHandle (*NDK: NLM Threads Management*), UnimportSymbol (page 164)

NWAddSearchPathAtEnd

Adds a search path to the end of the search path list that the OS uses to determine from where it loads NLM applications

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int NWAddSearchPathAtEnd  (
   BYTE   *searchPath,
   LONG   *number);
```

Parameters

searchPath

(IN) Points to a new path to be added at the end of the search path list.

number

(OUT) Points to a number defining where the new search path falls in the list.

Return Values

The following table lists return values and descriptions.

0	Success
-1	Failure

Remarks

The `number` parameter may be used to delete the search path.

The number of search paths is equivalent to the number listed when the NetWare server console `search' is entered.

See Also

NWDeleteSearchPath (page 138)

NWDeleteSearchPath

Deletes a search path from the search path list the OS uses to determine from where it loads NLM applications

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int NWDeleteSearchPath  (
   LONG   searchPathNumber);
```

Parameters

searchPathNumber

(IN) Specifies the search path number to be deleted.

Return Values

The following table lists return values and descriptions.

0	Success
-1	Failure

Remarks

The number of search paths is equivalent to the number listed when the NetWare server console `search' is entered.

See Also

NWAddSearchPathAtEnd (page 137)

NWGarbageCollect

Unfragments freed server memory

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwmalloc.h>

void NWGarbageCollect  (
   LONG   NLMHandle ;
```

Parameters

`NLMHandle`

(IN) Specifies an NLM handle through which freed server memory will be unfragmented.

Remarks

NWGarbageCollect provides a programmatic way to unfragment server memory before that memory is unfragmented automatically by the OS. If a large number of calls have been made to allocate memory, especially in small pieces, the NetWare 4.x, 5.x, and 6.x OS often fragments server memory, causing subsequent memory allocation calls to fail. A call to NWGarbageCollect with a valid NLM handle unfragments all server memory.

For the `NLMHandle` parameter, pass in the handle returned by a call to GetNLMHandle.

Blocking Information: Although NWGarbageCollect can block in some instances, it does not always do so.

See Also

FindNLMHandle, GetNLMHandle, MapNLMIDToHandle (*NDK: NLM Threads Management*)

NWGetSearchPathElement

Returns a search path from the search path list the OS uses to determine from where it loads NLMs

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int NWGetSearchPathElement  (
   LONG   searchPathNumber,
   LONG  *isDOSSearchPath,
   BYTE  *searchPath);
```

Parameters

`searchPathNumber`

(IN) Specifies the search path number to be returned.

`isDOSSearchPath`

(OUT) Points to a flag indicating whether the search path is for the DOS partition.

`searchPath`

(OUT) Points to a search path corresponding with `searchPathNumber`.

Return Values

The following table lists return values and descriptions.

0	Success
-1	Failure

Remarks

The number of search paths is equivalent to the number listed when the NetWare server console `search' is entered.

NWInsertSearchPath

Inserts a search path into the search path list the OS uses to determine from where it loads NLM applications

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int NWInsertSearchPath  (
   LONG   searchPathNumber,
   BYTE  *searchPath);
```

Parameters

searchPathNumber

(IN) Specifies the search path number to be entered.

searchPath

(IN) Points to a new search path to be added to the search path list.

Return Values

The following table lists return values and descriptions.

0	Success
-1	Failure

Remarks

The number of search paths is equivalent to the number listed when the NetWare server console `search' is entered.

qread

Performs a low-overhead read operation

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwfileio.h>

int qread  (
   int    fildes,
   void  *buffer,
   LONG   len,
   LONG   position);
```

Parameters

handle

(IN) Specifies the pertinent file handle.

buffer

(OUT) Points to a buffer where the data is to be received.

len

(IN) Specifies the number of bytes to read.

position

(IN) Specifies the byte offset in the file at which to start reading.

Return Values

If successful, this function returns the number of bytes read. If an error occurs, it returns -1 (EFAILURE) and `errno` and/or `NWErrno` can be set to:

Value	Name	Description
4	EBADF	Bad file number.
		Other error codes as appropriate

Remarks

The qread function does not:

- Perform parameter/context validation.
- Maintain file position.

This function does not support:

- Standard I/O
- Semaphore use of the handle
- Streams
- BSD Sockets

See Also

qwrite (page 144)

qwrite

Performs a low-overhead write operation

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwfileio.h>

int qwrite  (
   int     fildes,
   void   *buffer,
   LONG    len,
   LONG    position);
```

Parameters

handle

(IN) Specifies the pertinent file handle.

buffer

(IN) Points to a buffer which contains the data.

len

(IN) Specifies the number of bytes to write.

position

(IN) Specifies the byte offset at which to start writing.

Return Values

If successful, this function returns the number of bytes written. If an error occurs, it returns -1 (EFAILURE) and `errno` and/or `NWErrno` can be set to:

Value	Name	Description
4	EBADF	Bad file number.
		Other error codes as appropriate

Remarks

The qwrite function does not:

- Perform parameter/context validation.
- Maintain file position.

This function does not support:

- O_APPEND
- Standard I/O
- Semaphore use of the handle
- Streams
- BSD Sockets

See Also

qread (page 142)

RegisterConsoleCommand

Registers a console command parsing function

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

LONG RegisterConsoleCommand  (
   struct commandParserStructure   *newCommandParser);
```

Parameters

`newCommandParser`

(IN) Points to a command parsing function.

Return Values

If RegisterConsoleCommand is successful, it returns 0. Otherwise, it returns 0xFFFFFFFF.

`newCommandParser.parseRoutine` returns the following values:

0	The command was handled and does not allow any subsequently registered command parser to be envoked.
Nonzero	The command was not handled. The console command thread looks for other command parsers to handle the command. If none does, NetWare displays "??? Unknown Command ???"

Remarks

The command parsing function is called by the operating system whenever an unrecognized console command is entered. The parsing function is called with two parameters: a `screen ID` and a pointer to the complete console command line (an ASCIIZ string).

The commandParserStructure can be found in the nwadv.h header file and has the following definition:

```
struct commandParserStructure
   {
   struct commandParserStructure *Link;
      /* Set by RegisterConsoleCommand */
   LONG (*parseRoutine) (  /* Parsing routine (user-defined) */
      LONG screenID,
      BYTE *commandLine);
```

```
struct ResourceTagStructure *RTag;/* Set to resource tag */
};
```

The required resource tag is obtained with a call to AllocateResourceTag using the ConsoleCommandSignature constant (defined in nwadv.h) as the signature value.

The function registered by RegisterConsoleCommand runs as a callback (an OS Thread), which is not able to call most of the NetWare API functions, unless it is given CLIB context.

For 3.11 NLM applications, you must manually create the thread group context in your command parser, by calling SetThreadGroupID and passing a valid thread group ID. Before this thread returns, it should reset its context to its original context, by setting the thread group ID back to its original value.

For 4.x, 5.x, and 6.x NLM applications, the context that is given to the callbacks when they are registered is determined by the value in the registering thread's context specifier. You can set the context specifier to one of the following options:

- NO_CONTEXT-Callbacks registered with this option are not given CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context the callback is only able to call NetWare API functions that manipulate data or manage local semaphores.

  Once inside of your callback, you can manually give your callback thread CLIB context by calling SetThreadGroupID and passing in a valid thread group ID. If you manually set up your context, you need to reset its context to its original context, by setting the thread group ID back to its original value.
- USE_CURRENT_CONTEXT-Callbacks registered with a thread that has its context specifier set to USE_CURRENT_CONTEXT have the thread group context of the registering thread.
- A valid thread group ID-This is to be used when you want the callbacks to have a different thread group context than the thread that schedules them.

When a new thread is started with BeginThread, BeginThreadGroup or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

You can determine the current setting of the registering thread's context specifier by calling GetThreadContextSpecifier. Use SetThreadContextSpecifier to set the registering thread's context specifier to one of the above options.

For more information on using CLIB context, see Context Problems with OS Threads (*NDK: NLM Threads Management*).

See Also

AllocateResourceTag (page 122), UnRegisterConsoleCommand (page 165)

RegisterForEvent

Registers an operation to be called when the specified event occurs

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

extern LONG RegisterForEvent  (
   LONG   eventType,
   void (*reportProcedure)
   (
      LONG   parameter,
      LONG   userParameter
   ),
   LONG (*warnProcedure)
  (
      void (*OutputRoutine)
      (
          void  *controlString,  ...
      ),
      LONG   parameter,
      LONG   userParameter)
  ));
```

Parameters

eventType

(IN) Specifies an event type.

reportProcedure

(IN) Points to the events that occurred.

warnProcedure

(IN) Points to a warn procedure (optional).

Return Values

RegisterForEvent returns a nonzero event handle if successful. Otherwise, it returns EFAILURE (-1).

Remarks

The warning procedure pointed to by `warnProcedure` returns zero or nonzero. If nonzero, you are given a choice to unload.

The function registered by RegisterForEvent runs as a callback-an OS Thread-and is not able to call most of the NetWare APIs (unless it is given a CLIB context).

For 3.11 NLM applications, you must manually create the thread group context in your command parser by calling SetThreadGroupID and passing a valid thread group ID. Before this thread returns, you should reset its context to its original context, by setting the thread group ID back to its original value.

For 4.x, 5.x, and 6.x NLM applications, the context that is given to the callbacks when they are registered is determined by the value in the registering thread's context specifier:

- NO_CONTEXT The registered callback function is not given a CLIB context. Without a CLIB context, the callback function is able only to call NetWare APIs that manipulate data or manage local semaphores.
- USE_CURRENT_CONTEXT The registered callback function have the thread group context of the registering thread.
- A valid thread group ID is used when you want the callbacks to have a different thread group context than the thread that schedules them.

When a new thread is started by calling BeginThread, BeginThreadGroup or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

You can determine the current setting of the registering thread's context specifier by calling GetThreadContextSpecifier. Call SetThreadContextSpecifier to set the registering thread's context specifier to one of the above options.

What `parameter` is set to, whether the event calls a warn routine, whether or not the event can sleep, and each event's description follow:

#	Event: Description	Warn routine	Sleep
0	EVENT_VOL_SYS_MOUNT: `parameter` is undefined. Report Routine is called immediately after the SYS: volume has been mounted.	No	Yes
1	EVENT_VOL_SYS_DISMOUNT: `parameter` is undefined. Warn Routine and Report Routine are called before the SYS:volume is dismounted.	Yes	Yes
2	EVENT_ANY_VOL_MOUNT: `parameter` is a volume number. Report Routine is called immediately after any volume is mounted.	No	Yes
3	EVENT_ANY_VOL_DISMOUNT: `parameter` is a volume number. Warn Routine and Report Routine are called before any volume is dismounted.	Yes	Yes
4	EVENT_DOWN_SERVER: `parameter` is undefined. Warn Routine and Report Routine are called before the server is shut down.	Yes	Yes
7	EVENT_EXIT_TO_DOS: `parameter` is undefined. Report Routine is called before the server exits to DOS (NetWare 4.x, 5.x, and 6.x only).	No	Yes

#	Event: Description	Warn routine	Sleep
8	EVENT_MODULE_UNLOAD: `parameter` is a module handle. Warn Routine and Report Routine are called when a module is unloaded from the console command line. Only Report Routine is called when a module unloads itself (see Using the MODULE_UNLOAD Event: Example (*NDK: Sample Code*)).	Yes	Yes
9	EVENT_CLEAR_CONNECTION: `parameter` is a connection number. Report Routine is called before the connection is cleared.	No	Yes
10	EVENT_LOGIN_USER: `parameter` is a connection number. Report Routine is called after the connection has been allocated.	No	Yes
11	EVENT_CREATE_BINDERY_OBJ: `parameter` is an object ID. Report Routine is called after the object is created and entered in the bindery.	No	No
12	EVENT_DELETE_BINDERY_OBJ: `parameter` is an object ID. Report Routine is called before the object is removed from the bindery.	No	No
13	EVENT_CHANGE_SECURITY: `parameter` is a pointer to EventSecurityChangeStruct. Report Routine is called after a security equivalence change has occurred.	No	No
14	EVENT_ACTIVATE_SCREEN: `parameter` is a screen ID. Report routine is called after the screen becomes the active screen (NetWare 4.x, 5.x, 6.x only).	No	No
15	EVENT_UPDATE_SCREEN: `parameter` is a screen ID. Report routine is called after a change is made to the screen image (NetWare 4.x, 5.x, and 6.x only).	No	No
16	EVENT_UPDATE_CURSOR: `parameter` is a screen ID. Report routine is called after a change to the cursor position or state occurs (NetWare 4.x, 5.x, and 6.x only).	No	No
17	EVENT_KEY_WAS_PRESSED: `parameter` is undefined. Report routine is called at interrupt time whenever a key on the keyboard is pressed (including shift/alt/control).	No	No
18	EVENT_DEACTIVATE_SCREEN: `parameter` is a screen ID. Report routine is called when the screen becomes inactive (NetWare 4.x, 5.x, and 6.x only).	No	No
19	EVENT_TRUSTEE_CHANGE: `parameter` is a pointer to EventTrusteeChangeStruct (page 175). Report Routine is called everytime there is a change to a trustee in the file system.	No	No
20	EVENT_OPEN_SCREEN: `parameter` is the screen ID for the newly created screen. Report Routine is called after the screen is created (NetWare 4.x, 5.x, and 6.x only).	No	Yes
21	EVENT_CLOSE_SCREEN: `parameter` is the screen ID for the screen that is to be closed. Report Routine is called before the screen is closed (NetWare 4.x, 5.x, and 6.x only).	No	Yes
22	EVENT_MODIFY_DIR_ENTRY: `parameter` is a pointer to EventModifyDirEntryStruct (page 172) which contains the modify information. Report Routine is called right after the entry is changed and before the directory entry is unlocked.	No	No

#	Event: Description	Warn routine	Sleep
23	EVENT_NO_RELINQUISH_CONTROL: `parameter` is the running process. Report Routine is called when the timer detects that a process is hogging the processor (NetWare 4.x, 5.x, and 6.x only).	No	No
25	EVENT_THREAD_SWITCH: `parameter` is the thread's ID that was executing when the thread switch occurred. Report Routine is called when the new thread begins executing (applies only to threads in the calling NLM).	No	No
27	EVENT_MODULE_LOAD: `parameter` is module handle. Report Routine is called after a module has loaded (NetWare 4.x, 5.x, and 6.x only).	No	Yes
28	EVENT_CREATE_PROCESS: `parameter` is the PID of the process being created. Report Routine is called after the process is created (NetWare 4.x, 5.x, and 6.x only).	No	No
29	EVENT_DESTROY_PROCESS: `parameter` is the PID of the process being destroyed. Report Routine is called before the process is actually destroyed (NetWare 4.x, 5.x, and 6.x only).	No	No
32	EVENT_NEW_PUBLIC: `parameter` is a pointer to a length preceded string containg the name of the new public entry point (NetWare 4.x, 5.x, and 6.x only).	No	No
33	EVENT_PROTOCOL_BIND: `parameter` is a pointer to EventProtocolBindStruct. This event is generated every time a board is bound to a protocol (NetWare 4.x, 5.x, and 6.x only).	No	Yes
34	EVENT_PROTOCOL_UNBIND: `parameter` is a pointer to EventProtocolBindStruct. This event is generated every time a board is unbound from a protocol (NetWare 4.x, 5.x, and 6.x only).	No	Yes
37	EVENT_ALLOCATE_CONNECTION: `parameter` is a connection number. Report Routine is called after the connection is allocated (NetWare 4.x, 5.x, and 6.x only).	No	Yes
38	EVENT_LOGOUT_CONNECTION: `parameter` is a connection number. After NetWare 5.1, any NLM that is registered for EVENT_LOGOUT_CONNECTION will still be called; but the connection will be logged out at that point.	No	Yes
39	EVENT_MLID_REGISTER: `parameter` is a board number. Report Routine is called after the MLID software is registered (NetWare 4.x, 5.x, and 6.x only).	No	No
40	EVENT_MLID_DEREGISTER: `parameter` is a board number. Report Routine is called before the MLID is deregistered (NetWare 4.x, 5.x, and 6.x only).	No	No
41	EVENT_DATA_MIGRATION: `parameter` is a pointer to EventDateMigrationInfo (page 171). This event is generated when a file's data has been migrated (NetWare 4.x, 5.x, and 6.x only).	No	No
42	EVENT_DATA_DEMIGRATION: `parameter` is a pointer to EventDateMigrationInfo (page 171). This event is generated when a file's data has been de-migrated (NetWare 4.x, 5.x, and 6.x only).	No	No

#	Event: Description	Warn routine	Sleep
43	EVENT_QUEUE_ACTION: `parameter` is a pointer to EventQueueNote. This event is generated when a queue is activated, deactivated, created, or deleted (NetWare 4.x, 5.x, and 6.x only).	No	No
44	EVENT_NETWARE_ALERT: `parameter` is a pointer to EventNetwareAlertStruct (page 173). SystemAlert, QueueSystemAlert, INWSystemAlert, and INWQueueSystemAlert (all NetWare 3.x only) call NetWareAlert which generates this NetWare 4.x, 5.x, and 6.x event.	No	Yes
50	EVENT_CLOSE_FILE: `parameter` is a pointer to EventCloseFileInfo (page 170) (NetWare 4.x, 5.x, and 6.x only).	No	No
51	EVENT_CHANGE_TIME	No	No
56	EVENT_MODULE_UNLOADED	No	Yes
57	EVENT_REMOVE_PUBLIC	No	Yes
60	EVENT_SFT3_SERVER_STATE	No	No
61	EVENT_SFT3_IMAGE_STATE	No	No
62	EVENT_SFT3_PRESYNC_STATE	No	Yes

Use event 44, EVENT_NETWARE_ALERT, in place of event 24, EVENT_SYS_ALERT.

To register for NDS events, call NWDSERegisterForEvent (NDK: eDirectory Event Services).

See Also

UnregisterForEvent (page 166), NWDSEUnRegisterForEvent (NDK: eDirectory Event Services)

SaveThreadDataAreaPtr

Sets the thread switch Data Area Pointer for the current thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

void SaveThreadDataAreaPtr  (
   void    *threadDataAreaPtr);
```

Parameters

threadDataAreaPtr

(IN) Points to the user-defined thread switch Data Area Pointer.

Return Values

This function does not return a value.

Remarks

When thread switch event reporting has been registered, the Data Area Pointer is passed as the parameter to the report routine when a thread switch occurs.

The pointer can point to any user-defined data structure.

See Also

GetThreadDataAreaPtr (page 133), RegisterForEvent (page 148)

ScanSetableParameters

Returns information about NetWare server console parameters

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

LONG ScanSetableParameters  (
   LONG    scanCategory,
   LONG   *scanSequence,
   BYTE   *rParameterName,
   LONG   *rType,
   LONG   *rFlags,
   LONG   *rCategory,
   void   *rParameterDescription,
   void   *rCurrentValue,
   LONG   *rLowerLimit,
   LONG   *rUpperLimit);
```

Parameters

scanCategory

(IN) Specifies the category for which to return setable parameter information.

scanSequence

(IN/OUT) Points to this parameter is used for calling this function iteratively. On the first call, this parameter should be set to 0. On subsequent calls, use the value returned in this parameter. When all information has been returned, this function returns -1 (unsuccessful).

rParameterName

(IN/OUT) Points to or receives the name of a setable parameter (an ASCIIZ string). (Input if `scanCategory` is -2 or -5.)

rType

(OUT) Points to the type of the setable parameter.

rFlags

(OUT) Points to the setable parameter flags.

rCategory

(OUT) Points to the setable parameter category.

rParameterDescription

(OUT) Points to the description of a setable parameter (an ASCIIZ string).

rCurrentValue

(OUT) Points to the value (a number or string, depending on `rType`) to which the setable parameter is currently set. Receives the size of the current value, rather than the value itself if `scanCategory` is set to -2.

rLowerLimit

(OUT) Points to the lower limit of the setable parameter.

rUpperLimit

(IN/OUT) Points to the upper limit of the setable parameter. (Input if `scanCategory` is -4 or -5; must be at least 512 bytes.)

Return Values

This function returns 0 if successful, or a negative value if unsuccessful.

Remarks

This function returns information about setable parameters. A setable parameter is a NetWare OS parameter that can be set using the SET console command.

The `scanCategory` parameter defines what information the function returns. This parameter can have one of the following values:

Value	Description
0	Scan category by number. Replace 0 with a category number, for example 2 for FILE CACHE. To scan all parameters in a category, set `scanSequence` to 0 on the first call.
-1	Scan all categories. To scan all parameters in all categories, set `scanSequence` to 0 on the first call.
-2	Selected set parameter (`rParameterName` is input and points to a parameter name string)
-3	Return category names (the `scanSequence` parameter is input and points to a value of a category name for which the name string is returned in the `rParameterName` pointer.)
-4	Fill a buffer pointed to by `rCurrentValue` with information about the next parameter by sequence number as pointed to by `scanSequence`. To return information iteratively about all parameters, set `scanSequence` to 0 on the first call. (See below for explanation of the buffer.)
-5	Fill a buffer pointed to by `rCurrentValue` with information about a parameter as specified by name with the `rParmaterName` pointer. (See below for explanation of the buffer.)

If `scanCategory` is -4 or -5, this function returns information into a buffer pointed to by `rCurrentValue`. The buffer must be at least 512 bytes. Novell does not provide a parser for this buffer, which is filled in the following order:

```
long          paramType
long          category
long          flags
```

```
string        parameterName  /* Null terminated string */
string/long  parameterValue /*Either long or null-terminated string */
```

The `paramType` segment contains a value that corresponds to those of the `rType` parameter, explained below.

The `category` segment contains a value that corresponds to those of the `rCategory` parameter, explained below.

The `flags` segment contains a value that corresponds to those of the `rFlags` parameter, explained below.

The `parameterName` segment contains a string that names the parameter, as explained about the `rParameterName` parameter below.

The `parameterValue` segment contains either a long or a string, depending upon the parameter type as returned in the `paramType` segment.

The `rParameterName` parameter is the name of the setable parameter, such as "Cache Buffer Size".

The `rType` parameter receives the type of the setable parameter:

0	number
1	boolean
2	time ticks
3	block shift
4	offset
5	string
6	trigger
7	boolean with a uint32 data type

The "trigger" type is a level at which an event would happen. The "Minimum File Cache Buffer Report Threshold" is an example of a trigger type.

The `rFlags` parameter defines properties of the parameter, such as when it can be set:

0x0001	startup only
0x0004	advanced parameter
0x0008	startup or later
0x0010	not secured console-that is, the parameter cannot be set if the console is secured

The `rCategory` parameter can be one of the following categories:

0	COMMUNICATIONS
1	MEMORY

2	FILE CACHE
3	DIR CACHE
4	FILE SYSTEM
5	LOCKING
6	TTS
7	DISK
8	TIME
9	NCP
10	MISCELLANEOUS

See Also

GetSetableParameterValue (page 132), SetSetableParameterValue (page 162), "SET" in *Supervising the Network*

ScheduleNoSleepAESProcessEvent

Defines a procedure that is to be called by the Asynchronous Scheduler (AES) after a specified delay

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwthread.h>

extern void ScheduleNoSleepAESProcessEvent (
   struct AESProcessStructure   *EventNode);
```

Parameters

EventNode

(IN) Points to a structure that defines the event.

Remarks

The defined procedure must not go to sleep when it runs. An abend results if the procedure sleeps. The event is called at process time.

The AESProcessStructure is defined as follows:

```
struct AESProcessStructure
  {
  struct AESProcessStructure *ALink; /*Set by AES*/
  LONG AWakeUpDelayAmount;           /*Set to # ticks to
                                       wait*/
  LONG AWakeUpTime;                  /*Set by AES*/
  void (*AProcessToCall) (void *);   /*Set to function to
                                       call*/
  LONG ARTag;                        /*Set to resource tag */
  LONG AOldLink;                     /*Set to NULL*/
  }
```

Fields that are not set by AES must be set by the user as specified in the above structure definition.

When the defined procedure is called, the AESProcessStructure pointer is passed to it as the only parameter. By adding fields to the end of the structure, the user can pass information to the procedure.

If the event procedure reschedules itself, the function can be made to execute periodically. The scheduled event can be cancelled before time is up by calling CancelNoSleepAESProcessEvent.

The procedure registered by ScheduleNoSleepAESProcessEvent runs as a callback (an OS Thread), which is not able to call most of the NetWare API functions, unless it is given CLIB context.

For 3.11 NLM applications, you must manually create the thread group context in your procedure, by calling SetThreadGroupID and passing a valid thread group ID. Before this thread returns, it should reset its context to its original context, by setting the thread group ID back to its original value.

For 4.x, 5.x, and 6.x NLM applications, the context that is given to the callbacks when they are registered is determined by the value in the registering thread's context specifier. You can set the context specifier to one of the following options:

- NO_CONTEXT-Callbacks registered with this option are not given CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context the callback is only able to call NetWare API functions that manipulate data or manage local semaphores.

  Once inside of your callback, you can manually give your callback thread CLIB context by calling SetThreadGroupID and passing in a valid thread group ID. If you manually set up your context, you need to reset its context to its original context, by setting the thread group ID back to its original value.
- USE_CURRENT_CONTEXT-Callbacks registered with a thread that has its context specifier set to USE_CURRENT_CONTEXT have the thread group context of the registering thread.
- A valid thread group ID-This is to be used when you want the callbacks to have a different thread group context than the thread that schedules them.

When a new thread is started with BeginThread, BeginThreadGroup or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

You can determine the current setting of the registering thread's context specifier by calling GetThreadContextSpecifier. You use SetThreadContextSpecifier to set the registering thread's context specifier to one of the above options.

For more information on using CLIB context, see Context Problems with OS Threads (*NDK: NLM Threads Management*).

See Also

AllocateResourceTag (page 122), CancelNoSleepAESProcessEvent (page 127), ScheduleSleepAESProcessEvent (page 160)

ScheduleSleepAESProcessEvent

Defines a procedure that is to be called by the Asynchronous Scheduler (AES) after a specified delay

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwthread.h>

extern void ScheduleSleepAESProcessEvent (
   struct AESProcessStructure   *EventNode);
```

Parameters

EventNode

(IN) Points to the AESProcessStructure, which defines the event.

Remarks

The defined procedure can go to sleep when it runs. The event is called at process time.

The AESProcessStructure is defined as follows:

```
struct AESProcessStructure
  {
  struct AESProcessStructure *ALink; /*Set by AES*/
  LONG AWakeUpDelayAmount;           /*Set to # ticks to
                                       wait*/
  LONG AWakeUpTime;                  /*Set by AES*/
  void (*AProcessToCall) (void *);   /*Set to function to
                                       call*/
  LONG ARTag;                        /*Set to resource tag */
  LONG AOldLink;                     /*Set to NULL*/
  }
```

Fields that are not set by AES must be set by the user as specified in the above structure definition.

When the defined procedure is called, the AESProcessStructure pointer is passed to it as the only parameter. By adding fields to the end of the structure, the user can pass information to the procedure.

If the event procedure reschedules itself, the function can be made to execute periodically. The scheduled event can be cancelled before time is up by calling CancelSleepAESProcessEvent.

The procedure registered by ScheduleSleepAESProcessEvent runs as a callback (an OS Thread), which is not able to call most of the NetWare API functions, unless it is given CLIB context.

For 3.11 NLM applications, you must manually create the thread group context in your procedure, by calling SetThreadGroupID and passing a valid thread group ID. Before this thread returns, it should reset its context to its original context, by setting the thread group ID back to its original value.

For 4.x, 5.x, and 6.x NLM applications, the context that is given to the callbacks when they are registered is determined by the value in the registering thread's context specifier. You can set the context specifier to one of the following options:

- NO_CONTEXT-Callbacks registered with this option are not given CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context the callback is only able to call NetWare API functions that manipulate data or manage local semaphores.

  Once inside of your callback, you can manually give your callback thread CLIB context by calling SetThreadGroupID and passing in a valid thread group ID. If you manually set up your context, you need to reset its context to its original context, by setting the thread group ID back to its original value.
- USE_CURRENT_CONTEXT-Callbacks registered with a thread that has its context specifier set to USE_CURRENT_CONTEXT have the thread group context of the registering thread.
- A valid thread group ID-This is to be used when you want the callbacks to have a different thread group context than the thread that schedules them.

When a new thread is started with BeginThread, BeginThreadGroup or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

You can determine the current setting of the registering thread's context specifier by calling GetThreadContextSpecifier. Use SetThreadContextSpecifier to set the registering thread's context specifier to one of the above options.

For more information on using CLIB context, see Context Problems with OS Threads (*NDK: NLM Threads Management*).

See Also

AllocateResourceTag (page 122), CancelSleepAESProcessEvent (page 128), ScheduleNoSleepAESProcessEvent (page 158)

SetSetableParameterValue

Sets the value of a NetWare server console parameter

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

LONG SetSetableParameterValue  (
   LONG   connectionNumber,
   BYTE  *setableParameterString,
   void  *newValue);
```

Parameters

`connectionNumber`

(IN) Specifies the connection number of the user who wants to modify server console parameters.

`setableParameterString`

(IN) Points to a NULL-terminated ASCIIZ string representing the name of the server console parameter.

`newValue`

(IN) Points to the new value of the server console parameter.

Return Values

Returns 0 if successful, or -1 if an invalid setable parameter string was specified.

Remarks

A setable parameter is a NetWare OS parameter that can be set using the SET console command (see the *Utilities Reference* included with the NetWare 5 release).

The value returned in the `newValue` parameter depends upon the server console parameter passed into the `setableParameterString` parameter.

See Also

GetSetableParameterValue (page 132), ScanSetableParameters (page 154)

SynchronizeStart

Restarts the NLM startup process when using synchronization mode

Local Servers: blocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

void SynchronizeStart  (void);
```

Remarks

This function is used in synchronization mode to restart the startup process, which is put to sleep to make sure that another NLM is not loaded before the current NLM application's mainline is reached. Synchronization mode is selected at link time by using the SYNCHRONIZE keyword in the link directive file.

NOTE: If an NLM is using synchronization mode, it should include a call to SynchronizeStart as early in the code as possible. Synchronize mode causes the console command process to go to sleep until SynchronizeStart is called.

If you specify the SYNCHRONIZE keyword, the loader does not proceed until your NLM calls SynchronizeStart. Without SYNCHRONIZE, the previously loaded NLM might not have executed any of its code before the loader executes the next command in the AUTOEXEC.NCF file. Use this technique if you have an NLM that must establish some conditions to be used by some subsequent command or NLM in your AUTOEXEC.NCF file. It prevents the loader from proceeding until after you have called SynchronizeStart.

See syncstrt.c (../../../samplecode/clib_sample/nlm/syncstrt/syncstrt.c.html).

UnimportSymbol

Eliminates dependency of an NLM on the specified external symbol

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 3.2, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int UnimportSymbol  (
   int    NLMHandle,
   char  *symbolName);
```

Parameters

NLMHandle

(IN) Specifies the handle of the NLM for which to unimport the symbol.

symbolName

(IN) Points to the symbol to unimport.

Return Values

This function returns 0 if successful. Otherwise, it returns an error code.

Remarks

UnimportSymbol reverses the effect of ImportSymbol, ending your the dependency of your NLM on the NLM that exports the symbol specified by `symbolName`. The `NLMHandle` parameter can be obtained by calling FindNLMHandle or GetNLMHandle.

See Also

ImportSymbol (page 136)

UnRegisterConsoleCommand

Unregisters a console command parsing function

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

LONG UnRegisterConsoleCommand  (
   struct commandParserStructure   *commandParser);
```

Parameters

commandParser

(IN) Points to the command parsing function that is to be unregistered.

Return Values

This function returns a value of 0 if successful. If the specified command parsing function is not found (has not been registered), it returns a value of -1.

Remarks

This function should be called to unregister a command parsing function previously defined with RegisterConsoleCommand.

See Also

RegisterConsoleCommand (page 146)

UnregisterForEvent

Cancels a previous registration for event notification

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Advanced

Syntax

```
#include <nwadv.h>

int UnregisterForEvent  (
   LONG   eventHandle);
```

Parameters

eventHandle

(IN) Specifies the event handle that was returned by RegisterForEvent.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

See Also

RegisterForEvent (page 148)

Advanced Structures

12

This documentation alphabetically lists the Advanced structures and describes their purpose, syntax, and fields.

AESProcessStructure

Defines a process to be called by the Asynchronous Scheduler (AES)

Service: Advanced

Defined In: nwadv.h

Structure

```
struct AESProcessStructure {
   struct AESProcessStructure   *ALink ;
   LONG                          AWakeUpDelayAmount ;
   LONG                          AWakeUpTime ;
   void                          (*AProcessToCall) (void *);
   LONG                          ARTag ;
   LONG                          AOldLink ;
}
```

Fields

`Alink`

Points to set by AES.

`AWakeUpDelayAmount`

Specifies the number of ticks to wait (developer-defined).

`AWakeUpTime`

Specifies set by AES.

`AProcessToCall`

Points to the function to call (developer-defined).

`ARTag`

Specifies the resource tag (developer-defined).

`AOldLink`

Specifies set this field to NULL.

commandParserStructure

Contains information about a developer-defined console command parsing function

Service: Advanced

Defined In: nwadv.h

Structure

```
struct commandParserStructure
{
   struct commandParserStructure    *Link ;
   LONG                            (*parseRoutine) (
      LONG   screenID,
      BYTE   *commandLine);
   LONG                             RTag ;
};
```

Fields

`Link`

Points to set by RegisterConsoleCommand.

`parseRoutine`

Points to a developer-defined parsing routine

`RTag`

Specifies a resource tag (developer-defined).

EventCloseFileInfo

Returns when a file is closed

Service: Advanced

Defined In: nwadv.h

Structure

```
struct EventCloseFileInfo {
   LONG   fileHandle;
   LONG   station;
   LONG   task;
   LONG   fileHandleFlags;
   LONG   completionCode;
};
```

Fields

`fileHandle`

Specifies the handle of the file that was closed.

`station`

Specifies the connection number that closed the file.

`task`

Specifies the task number of the connection that closed the file.

`fileHandleFlags`

Specifies the attributes of the file handle. See `fileHandleFlags` in nwadv.h for a list of the flags.

`completionCode`

Specifies the outcome of the close file operation.

EventDateMigrationInfo

Returns for EVEBT_DATA_MIGRATION and DEMIGRATION

Service: Advanced

Defined In: nwadv.h

Structure

```
struct EventDateMigrationInfo {
   LONG   FileSystemTypeID;
   LONG   Volume;
   LONG   DOSDirEntry;
   LONG   OwnerDirEntry;
   LONG   OwnerNameSpace;
   BYTE   OwnerFileName[256];
};
```

Fields

FileSystemTypeID

Specifies the file system type (NETWARE386FILESYSTEM, NETWARENFSFILESYSTEM, NETWARECDROMFILESYSTEM, IBM_SMB_LAN_SERV_FS-see nwadv.h).

Volume

Specifies on which volume the entry is located.

DOSDirEntry

Specifies the directory number of the entry in the DOS name space.

OwnerDirEntry

Specifies the directory number of the entry in an other than DOS name space (if applicable).

OwnerNameSpace

Specifies the name space number of this entry (see Name Space Flag Values in Multiple and Inter-File Services).

OwnerFileName

Specifies the name of entry in the OwnerNameSpace name space (255 + 1 len byte).

EventModifyDirEntryStruct

Returns for EVEBT_MODIFY_DIR_ENTRY

Service: Advanced

Defined In: nwadv.h

Structure

```
struct EventModifyDirEntryStruct {
   LONG                     primaryDirectoryEntry;
   LONG                     nameSpace;
   LONG                     modifyBits;
   struct ModifyStructure   *modifyVector;
   LONG                     volumeNumber;
   void                     *reserved;
};
```

Fields

primaryDirectoryEntry

Specifies the directory number of the entry being modified.

nameSpace

Specifies the name space in which the modification is occurring (see Name Space Flag Values in Multiple and Inter-File Services).

modifyBits

Specifies the fields of the directory entry that are being changed (see nwdir.h).

modifyVector

Points to the structure that contains the updated fields of the directory entry (see nwdir.h).

volumeNumber

Specifies on which volume the entry is located.

reserved

Reserved.

EventNetwareAlertStruct

Contains information about an alert event.

Syntax

```
struct EventNetwareAlertStruct {
   LONG   alertFlags;
   LONG   alertId;
   LONG   alertLocus;
   LONG   alertClass;
   LONG   alertSeverity;
   LONG   targetStationCount;
   LONG   targetStationList[32];
   LONG   targetNotificationBits;
   LONG   alertParmCount;
   void  *alertDataPtr;
   void  *NetWorkManagementAttributePointer;
   LONG   alertUnused[2];
   LONG   alertControlStringMessageNumber;
   BYTE   alertControlString[256];
   BYTE   alertParameters[256+256];
   BYTE   alertModuleName[36];
   LONG   alertModuleMajorVersion;
   LONG   alertModuleMinorVersion;
   LONG   alertModuleRevision;
};
```

Fields

`alertFlags`

Specifies the flags set at the time of the event (see Section 13.2, “Alert Flag Values,” on page 179).

`alertId`

Specifies the ID of the alert (see Section 13.3, “Alert ID Values,” on page 180).

`alertLocus`

Specifies the location of the alert (see Section 13.4, “Alert Location Values,” on page 181).

`alertClass`

Specifies the class of the alert (see Section 13.1, “Alert Class Values,” on page 179).

`alertSeverity`

Specifies the severity of the alert (see Section 13.5, “Alert Severity Values,” on page 182).

`targetStationCount`

Specifies the number of valid entries in `targetStationList`.

`targetStationList`

Specifies the first 32 stations that were notified of the event.

targetNotificationBits

Specifies the notifications that are generated by the alert (see Section 13.6, “Target Notification Bit Values,” on page 182).

alertParmCount

Is not implemented currently.

alertDataPtr

Is not implemented currently.

NetWorkManagementAttributePointer

Is reserved.

alertUnused

Is not implemented currently.

alertControlStringMessageNumber

Is not implemented currently.

alertControlString

Is not implemented currently.

alertParameters

Is not implemented currently.

alertModuleName

Specifies the name of the NLM that generated the alert.

alertModuleMajorVersion

Specifies the major version of the module that generated the alert.

alertModuleMinorVersion

Specifies the minor version of the module that generated the alert.

alertModuleRevision

Specifies the revision number of the module that generated the alert.

EventTrusteeChangeStruct

Returns for EVENT_TRUSTEE_CHANGE

Service: Advanced

Defined In: nwadv.h

Structure

```
struct EventTrusteeChangeStruct {
   LONG   objectID ;
   LONG   entryID ;
   LONG   volumeNumber ;
   LONG   changeFlags ;
   LONG   newRights ;
};
```

Fields

objectID

Specifies the bindery object ID of the trustee being changed.

entryID

Specifies the directory number of the file or directory that is having the trustee changed.

volumeNumber

Specifies on which volume the entry is located.

changeFlags

Specifies the type of change:

1 EVENT_NEW_TRUSTEE
2 EVENT_REMOVE_TRUSTEE
4 EVENT_TRUSTEE_RIGHTS_MODIFIED

newRights

Specifies the new trustee's rights.

T_cacheBufferStructure

Contains cache buffer information returned by an asynchronous read

Service: Advanced

Defined In: nwadv.h

Structure

```
typedef struct cacheBufferStructure
{
   char   *cacheBufferPointer ;
   LONG    cacheBufferLength ;
   int     completionCode ;
} T_cacheBufferStructure;
```

Fields

`cacheBufferPointer`

Points to the address of the first character for the cache buffer.

`cacheBufferLength`

Specifies the number of bytes to be used from the cache buffer.

`completionCode`

Specifies the NetWare® error code for the buffer read operation.

T_DYNARRAY_BLOCK

Defines a dynamic array block (DAB)

Service: Advanced

Defined In: nwdnarr.h

Structure

```
typedef struct tagT_DYNARRAY_BLOCK
{
   void   *DABarrayP ;
   int     DABnumSlots ;
   int     DABelementSize ;
   void   *(*DABrealloc) (void *, size_t);
   int     DABgrowAmount ;
   int     DABnumEntries ;
} T_DYNARRAY_BLOCK;
```

Fields

DABarrayP

Points to the dynamic array.

DABnumSlots

Specifies the initial number of elements in the dynamic array

DABelementSize

Specifies the size (in bytes) of each element in the dynamic array

DABrealloc

Points to a memory allocation function. This function is normally realloc, but you can define your own function.

DABgrowAmount

Specifies the number of elements by which to increase the dynamic array when more elements are needed.

DABnumEntries

Specifies the number of entries in the dynamic array.

T_mwriteBufferStructure

Contains information about a buffer to be used by gwrite (page 134)

Service: Advanced

Defined In: nwfileio.h

Structure

```
typedef struct mwriteBufferStructure
{
   char  *mwriteBufferPointer;
   LONG   mwriteBufferLength;
   int    reserved;
} T_mwriteBufferStructure;
```

Fields

`mwriteBufferPointer`

Points to a buffer.

`mwriteBufferLength`

Specifies the size of the buffer.

Advanced Values

13

This documentation describes values associated with Advanced.

13.1 Alert Class Values

`alertClass` can have the following values:

Value	Name
0	CLASS_UNKNOWN
1	CLASS_OUT_OF_RESOURCE
2	CLASS_TEMP_SITUATION
3	CLASS_AUTHORIZATION_FAILURE
4	CLASS_INTERNAL_ERROR
5	CLASS_HARDWARE_FAILURE
6	CLASS_SYSTEM_FAILURE
7	CLASS_REQUEST_ERROR
8	CLASS_NOT_FOUND
9	CLASS_BAD_FORMAT
10	CLASS_LOCKED
11	CLASS_MEDIA_FAILURE
12	CLASS_ITEM_EXISTS
13	CLASS_STATION_FAILURE
14	CLASS_LIMIT_EXCEEDED
15	CLASS_CONFIGURATION_ERROR
16	CLASS_LIMIT_ALMOST_EXCEEDED
17	CLASS_SECURITY_AUDIT_INFO
18	CLASS_DISK_INFORMATION
19	CLASS_GENERAL_INFORMATION
20	CLASS_FILE_COMPRESSION
21	CLASS_PROTECTION_VIOLATION

13.2 Alert Flag Values

`alertFlags` can have the following values:

Value	Name
0x00000001	QueueThisAlertMask
0x00000002	AlertIDValidMask
0x00000004	AlertLocusValidMask
0x00000008	AlertEventNotifyOnlyMask
0x00000010	AlertNoEventNotifyMask
0x00010000	AlertMessageNumberValid
0x00400000	AlertNoRingBell
0x00800000	AlertIDNotUniqueBit
0x01000000	OldStyleSystemAlertMask
0x02000000	OldStyleINWSystemAlertMask
0x10000000	NoDisplayLocusBit
0x20000000	NoDisplayAlertIDBit
0x40000000	OverrideNotificationBits
0x80000000	TargetStationIsAPointer
AlertLocusValidMask OR QueueThisAlertMask OR NoDisplayAlertIDBit OR AlertIDValidMask	QAlert320Mask
AlertLocusValidMask OR NoDisplayAlertIDBit OR AlertIDValidMask	Alert320Mask
NOTIFY_ERROR_LOG_BITOR NOTIFY_CONSOLE_BIT	StandardNotify

13.3 Alert ID Values

AlertIDs for NetWare have been organized to a two-fold ID:

- Upper 16 bits contain the value (MASK) for an NLM or a product as assigned by Novell
- Lower 16 bits contain the values 0x80000000-0xFFFF0000 that uniquely identify each MASK.

`alertId` can have the following MASK values:

Value	Name
0x01020000	ALERT_BINDERY
0x01030000	ALERT_OS
0x01040000	ALERT_LLC
0x01050000	ALERT_SDLC
0x01060000	ALERT_REMOTE

Value	Name
0x01070000	ALERT_MLID
0x01080000	ALERT_QLLC
0x01090000	ALERT_UPS
0x010A0000	ALERT_DS
0x010B0000	ALERT_DOMAIN
0x010C0000	ALERT_RSPX
0x010D0000	ALERT_R232

13.4 Alert Location Values

`alertLocus` can have the following values:

Value	Name
0	LOCUS_UNKNOWN
1	LOCUS_MEMORY
2	LOCUS_FILESYSTEM
3	LOCUS_DISKS
4	LOCUS_LANBOARDS
5	LOCUS_COMSTACKS
7	LOCUS_TTS
8	LOCUS_BINDERY
9	LOCUS_STATION
10	LOCUS_ROUTER
11	LOCUS_LOCKS
12	LOCUS_KERNEL
13	LOCUS_UPS
14	LOCUS_SERVICE_PROTOCOL
15	LOCUS_SFT_III
16	LOCUS_RESOURCE_TRACKING
17	LOCUS_NLM
18	LOCUS_OS_INFORMATION
19	LOCUS_CACHE
20	LOCUS_DOMAIN

13.5 Alert Severity Values

`alertSeverity` can have the following values:

Value	Name
0	SEVERITY_INFORMATIONAL: Counters or Gauges reached the thresholds.
1	SEVERITY_WARNING: Configuration errors, etc. exist. No damage was done.
2	SEVERITY_RECOVERABLE: Hot Fix disk, etc. exist. A workaround was made.
3	SEVERITY_CRITICAL: Disk Mirror failed, etc. A fix up was attempted.
4	SEVERITY_FATAL: Resource was fatally affected. Shut down your process.
5	SEVERITY_OPERATION_ABORTED: The operation cannot complete. The effects are unknown.
6	SEVERITY_NONOS_UNRECOVERABLE: The oepration cannot complete. This will not affect the OS.

13.6 Target Notification Bit Values

`targetNotificationBits` can have the following values:

Value	Name
0x00000001	NOTIFY_CONNECTION_BIT
0x00000002	NOTIFY_EVERYONE_BIT
0x00000004	NOTIFY_ERROR_LOG_BIT
0x00000008	NOTIFY_CONSOLE_BIT
0x10000000	NOTIFY_QUEUE_MESSAGE
0x80000000	DONT_NOTIFY_NMAGENT

Debug Functions

14

This documentation alphabetically lists the Debug functions and describes their purpose, syntax, parameters, and return values.

assert

Identifies program logic errors

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Platform: NLM

Service: Debug

Syntax

```
#include <assert.h>

void assert  (
   int   expression);
```

Parameters

`expression`

(IN) Specifies an expression to test assertion.

Return Values

assert returns no values. Because assert uses the printf function to display errors, `errno` can be set when an output occurs.

Remarks

assert prints a diagnostic message upon the `stderr` stream and terminates the program if `expression` is FALSE (0). The diagnostic message has the following form:

```
Assertion failed: expression, file filename, line
linenumber
```

The filename and linenumber variables are defined as follows:

filename	Specifies the name of the source file.
linenumber	Specifies the line number of the assertion that failed in the source file.

The filename and linenumber values are the values of the preprocessing macros __FILE__ and __LINE__, respectively. No action is taken if `expression` is TRUE (nonzero).

The given expression should be chosen so that it is true when the program is functioning as intended. After the program has been debugged, the special "no debug" identifier NDEBUG can be used to remove assert functions from the program when it is recompiled. If NDEBUG is defined (with any value) with a -d command line option or with a #define directive, the C preprocessor ignores all assert functions in the program source.

EnterDebugger

Enters system debugger

Local Servers: N/A

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Debug

Syntax

```
#include <nwdebug.h>

void EnterDebugger  (void);
```

Remarks

For information concering the NetWare internal debugger, see “NetWare Internal Debugger” on page 83.

See Also

Breakpoint

NWClearBreakpoint

Dynamically clears the breakpoint set with NWSetBreakpoint

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Debug

Syntax

```
#include <nwdebug.h>

void NWClearBreakpoint  (
   int   breakpoint);
```

Parameters

`breakpoint`

(IN) Specifies the breakpoint to clear.

Remarks

NWClearBreakpoint clears the breakpoint set with NWSetBreakpoint. The value for the `breakpoint` parameter is the return value from successful completion of NWSetBreakpoint.

See Also

NWSetBreakpoint (page 187)

NWSetBreakpoint

Sets a breakpoint programmatically

Local Servers: nonblocking

Local Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Debug

Syntax

```
#include <nwdebug.h>

int NWSetBreakpoint  (
   LONG   address,
   int    breakType ;
```

Parameters

address

(IN) Specifies the location of the breakpoint to set.

breakType

(IN) Specifies the breakpoint type.

Return Values

0-3 indicate success.

-1 indicates failure.

Remarks

NWSetBreakpoint provides a programmatic way to set a breakpoint dynamically.

For the `address` parameter, pass in a pointer to data if you are setting a write or a read/write breakpoint. Pass in a pointer to code if you are setting an execution breakpoint.

For the `breakType` parameter, pass in one of the following three constants, according to the type of breakpoint you are setting:

Constant Name	Defined Value
EXECUTION_BREAKPOINT	0
WRITE_BREAKPOINT	1
READ_WRITE_BREAKPOINT	3

If fewer than four breakpoints are set, NWSetBreakpoint returns the next higher zero-based count of breakpoints and sets the requested breakpoint. If all four breakpoints are already set when you call NWSetBreakpoint, the function fails and returns -1.

See Also

NWClearBreakpoint (page 186)

perror

Prints an error message

Local Servers: blocking

Remote Servers: N/A

Classification: ANSI

Platform: NLM

Service: Debug

Syntax

```
#include <stdio.h>

void perror  (
   const char   *prefix);
```

Parameters

`prefix`

(IN) Points to the error message to print.

Return Values

perror returns no values. Because perror uses the fprintf function, `errno` can be set when an error is detected during the execution of the function.

Remarks

The perror function prints, on the file designated by `stderr`, the error message corresponding to the error number contained in `errno`.

See Also

strerror (*NDK: Program Management*)

Device I/O Functions

15

This documentation alphabetically lists the device I/O functions and describes their purpose, syntax, parameters, and return values.

cgets

Gets a string of characters directly from the current screen and stores the string and its length in an array

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

char *cgets  (
   char   *buf);
```

Parameters

buf

(IN) Points to the array.

Return Values

cgets returns a pointer to the start of the string, which is at `buf` [2].

Remarks

The first element of the array `buf` [0] must contain the maximum length in characters of the string to be read. The array must be big enough to hold the string, a terminating null character, and two additional bytes.

The cgets function reads characters until a carriage-return/line-feed combination is read, or until the specified number of characters is read. The string is stored in the array starting at `buf` [2]. The carriage-return/line-feed combination, if read, is replaced by a null character. The actual length of the string read is placed in `buf` [1].

See Also

getch (page 201), getche (page 202), gets (Single and Intra-File Services)

Example

```
#include <nwconio.h>
#include <stdio.h>

main ()
{
```

```
   char buffer[82];
   buffer[0]=80;
   cgets (buffer );
   cprintf ("%s\r\n", &buffer[2] );
}
```

cprintf

Writes output directly to the current application screen under format control

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int cprintf  (
   const char   *format,
   ...);
```

Parameters

format

(IN) Points to the format specification string.

Return Values

Returns the number of characters written.

Remarks

The cprintf function outputs the formatted data directly to the console screen.

See Also

NWcprintf (page 207) and printf and vfprintf in Single and Intra-File Services

Example

```
#include <nwconio.h>
#include <stdio.h>

main ()
{
   char *weekday, *month;
   int day, year;
   weekday="Saturday";
   month="April";
   day=18;
   year=1991;
```

```
   cprintf ("%s, %s %d, %d\n", weekday, month, day, year);
}
```

produces the following:

```
Saturday, April 18, 1991
```

cputs

Writes a specified character string directly to the current screen

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int cputs  (
   const char   *buf);
```

Parameters

buf

(IN) Points to a character string.

Return Values

cputs returns a nonzero value if an error occurs. Otherwise, it returns a value of 0. When an error has occurred, `errno` is set.

Remarks

The carriage-return and line-feed characters are not appended to the string. The terminating NULL character is not written.

See Also

fputs (Single and Intra-File Services), putch (page 214)

Example

```
#include <nwconio.h>

main ()
{
   char   buffer[82];
   buffer[0]=80;
   cgets (buffer);
   cputs (&buffer[2] );
   putch ('\r');
   putch ('\n');
}
```

cscanf

Scans input from the current screen under format control

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int cscanf  (
   const char   *format,
   ...);
```

Parameters

`format`

(IN) Points to the format specification string.

Return Values

cscanf returns EOF when the scanning is terminated by reaching the end of the input screen. Otherwise, the number of input arguments for which values have been successfully scanned and stored is returned. When a file input error occurs, `errno` is set.

Remarks

Following the format string is a list of addresses to receive values. The scanf function uses the function getche to read characters from the console.

See Also

fscanf and scanf (Single and Intra-File Services)

Example

To scan a date in the form "Saturday, April 18 1990":

```
#include <nwconio.h>

main ()
{
   int    day, year;
   char   weekday[10], month[12];
```

```
    cscanf ("%s %s %d %d", weekday, month, &day, &year);
}
```

_disable (obsolete)

Removed from the documentation because, in order for the NetWare® API to be SFTIII™ compliant, this function will not be supported in the future

_enable (obsolete)

Removed from the documentation because, in order for the NetWare API to be SFTIII compliant, this function will not be supported in the future

getch

Obtains the next available keystroke from the current screen

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int getch  (void);
```

Return Values

This function returns a value of EOF when an error is detected. Otherwise, the getch function returns the value of the keystroke (or character).

When the keystroke represents an extended key (for example, a function key, a cursor-movement key, or the Alt key with a letter or a digit), a value of 0 is returned, and the next call to getch returns a value for the extended function. When an error occurs, `errno` is set.

Remarks

The getch function reads from the current screen. Nothing is echoed on the screen (getche echoes the keystroke, if possible). When no keystroke is available, the function waits until a key is depressed.

Use the kbhit function to determine if a keystroke is available.

See Also

getche (page 202), kbhit (page 206)

Example

```
#include <nwconio.h>
main ()
{
   int   keyStroke;
   keyStroke = getch ();
}
```

getche

Obtains the next available keystroke from the current screen and echoes the keystroke on the screen

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int getche  (void);
```

Return Values

getche returns a value of EOF when an error is detected. Otherwise, the getche function returns the value of the keystroke (or character).

When the keystroke represents an extended key (for example, a function key, a cursor-movement key, or the Alt key with a letter or a digit), a value of 0 is returned, and the next call to getche returns a value for the extended function. When an error occurs, `errno` is set.

Remarks

The getche function reads from the current screen. The function waits until a keystroke is available. That character is echoed on the screen at the position of the cursor. Use the getch function when it is not desired to echo the keystroke.

Use the kbhit function to determine if a keystroke is available.

See Also

getch (page 201), kbhit (page 206), ungetch (page 215)

Example

```
#include <stdlib.h>
#include <nwconio.h>

main ()
{
   int   keyStroke;
   while((keyStroke = getche()) != '0')
      printf ("%d\r\n",keyStroke);
}
```

inp

Reads 1 byte from the specified hardware port

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int inp  (
   int   port);
```

Parameters

port

(IN) Specifies the hardware port.

Return Values

The value returned is the byte that was read.

Remarks

The inp function reads 1 byte from the hardware port whose number is given by `port`.

A hardware port is used to communicate with a device. One byte can be read and/ or written from each port, depending on the hardware. Consult the technical documentation for your computer in order to determine the port numbers for a device and the expected usage of each port for a device.

See Also

inpd (page 204), inpw (page 205), outp (page 209), outpd (page 210), outpw (page 212)

Example

```
#include <nwconio.h>

main ()
{
   /* turn off speaker */
   outp (0x61,inp (0x61) & 0xFC);
}
```

inpd

Reads a double word (4 bytes) from the specified hardware port

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int inpd  (
   int   port);
```

Parameters

port

(IN) Specifies the hardware port.

Return Values

The value returned is the double word that was read.

Remarks

The inpd function reads a double word (4 bytes) from the hardware port whose number is given by `port`.

A hardware port is used to communicate with a device. One to 4 bytes can be read and/or written from each port, depending on the hardware. Consult the technical documentation for your computer in order to determine the port numbers for a device and the expected usage of each port for a device.

See Also

inp (page 203), outp (page 209), outpd (page 210), outpw (page 212)

Example

```
#include <nwconio.h>
#define DEVICE 34

main ()
{
   unsigned int transmitted;
   transmitted=inpd (DEVICE);
}
```

inpw

Reads a word (2 bytes) from the specified hardware port

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int inpw  (
   int   port);
```

Parameters

port

(IN) Specifies the hardware port.

Return Values

The value returned is the word that was read.

Remarks

The inpw function reads a word (2 bytes) from the hardware port whose number is given by `port`.

A hardware port is used to communicate with a device. One or 2 bytes can be read and/or written from each port, depending on the hardware. Consult the technical documentation for your computer in order to determine the port numbers for a device and the expected usage of each port for a device.

See Also

inp (page 203), inpd (page 204), outp (page 209), outpd (page 210), outpw (page 212)

Example

```
#include <nwconio.h>
#define DEVICE 34

main ()
{
   unsigned int transmitted;
   transmitted=inpw (DEVICE);
}
```

kbhit

Tests whether a keystroke is currently available

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int kbhit  (void);
```

Return Values

kbhit returns TRUE or FALSE, depending on availability of keystrokes. When a keystroke is available, TRUE is returned. If an error is detected or if no keystrokes are available, FALSE (0) is returned. When an error occurs, `errno` is set.

Remarks

When a keystroke is available, you can call getch or getche to obtain the keystroke. With a stand-alone program, you can call kbhit continuously until a keystroke is available.

See Also

getch (page 201), getche (page 202), putch (page 214), ungetch (page 215)

Example

```
#include <stdlib.h>
#include <nwconio.h>
#include <stdio.h>

main ()
{
   while(!kbhit());
      printf ("the character is ");
      getche ();
      getch ();
}
```

NWcprintf

Writes output directly to the current application screen under format control; enabled for internationalization

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int NWcprintf  (
   const char   *format,
   ...);
```

Parameters

format

(IN) Points to the format specification string.

Return Values

NWcprintf returns the number of characters written.

Remarks

The NWcprintf function is identical to the cprintf function, except that NWcprintf is enabled for internationalization.

See Also

printf and vprintf in Single and Intra-File Services, cprintf (page 194)

Example

```
#include <nwconio.h>
#include <stdio.h>

main ()
{
   char *weekday, *month;
   int day, year;
   weekday="Saturday";
   month="April";
   day=18;
```

```
   year=1991;
   cprintf ("%s, %s %d, %d\n", weekday, month, day, year);
}
```

produces the following:

```
Saturday, April 18, 1991
```

outp

Writes 1 byte, determined by `value` , to the hardware port whose number is given by `port`

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned char outp  (
   int              port,
   unsighned char   value);
```

Parameters

port

(IN) Specifies the hardware port.

value

(IN) Specifies the character to write.

Return Values

The value transmitted is returned.

Remarks

A hardware port is used to communicate with a device. One byte can be read and/ or written from each port, depending upon the hardware. Consult the technical documentation for your computer in order to determine the port numbers for a device and the expected usage of each port for a device.

See Also

inp (page 203), inpd (page 204), inpw (page 205), outpd (page 210), outpw (page 212)

Example

```
#include <nwconio.h

main ()
{
    /* turn off speaker */
   outp (0x61,inp (0x61) & 0xFC);
}
```

outpd

Writes a double word (4 bytes), determined by `value` , to the hardware port whose number is given by `port`

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int outpd  (
   int            port,
   unsigned int   value);
```

Parameters

`port`

(IN) Specifies the hardware port.

value

(IN) Specifies the double word to write.

Return Values

The value transmitted is returned.

Remarks

A hardware port is used to communicate with a device. One to 4 bytes can be read and/or written from each port, depending upon the hardware. Consult the technical documentation for your computer in order to determine the port numbers for a device and the expected usage of each port for a device.

See Also

inp (page 203), inpw (page 205), outp (page 209)

Example

```
#include <nwconio.h>
#define DEVICE 34

main ()
```

```
{
   outpd (DEVICE, 0x1234);
}
```

outpw

Writes a word (2 bytes), determined by `value`, to the hardware port whose number is given by `port`

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned short outpw  (
   int              port,
   unsigned short   value);
```

Parameters

`port`

(IN) Specifies the hardware port.

value

(IN) Specifies the word to write.

Return Values

The value transmitted is returned.

Remarks

A hardware port is used to communicate with a device. One or 2 bytes can be read and/or written from each port, depending upon the hardware. Consult the technical documentation for your computer in order to determine the port numbers for a device and the expected usage of each port for a device.

See Also

inp (page 203), inpd (page 204), inpw (page 205), outp (page 209), outpd (page 210)

Example

```
#include <nwconio.h>
#define DEVICE 34

main ()
```

```
{
   outpw (DEVICE, 0x1234);
}
```

putch

Writes a specified character to the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int putch  (
   int   charToOutput);
```

Parameters

`charToOutput`

(IN) Specifies the character to be written.

Return Values

If successful, putch returns the character written. If a write error occurs, the error indicator is set and putch returns EOF.

Remarks

The putch function writes the character specified by the `charToOutput` parameter to the current screen.

putch becomes a blocking function if the character to be written out is the newline character

See Also

getch (page 201), getche (page 202), ungetch (page 215)

Example

```
#include <stdlib.h>
#include <nwconio.h>

main ()
{
   putch ('a');
   putchar ('b');
   getch ();
}
```

ungetch

Pushes a specified character back onto the input stream for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <nwconio.h>

int ungetch  (
   int   charToPushBack);
```

Parameters

charToPushBack

(IN) Specifies the character to be pushed back to the console.

Return Values

ungetch returns the character pushed back to the console if successful.

Remarks

ungetch pushes the character specified by the `charToPushBack` parameter onto the input stream for the current screen. This character is returned by the next read from the console (by the getch or getche functions) and is detected by the kbhit function. Only the last character returned in this way is remembered.

ungetch clears the end-of-file indicator, unless the value of the `charToPushBack` parameter is EOF.

ungetch also pushes extended keystrokes. The following table lists extended keys and their " ungetch " values:

Key	Value
F1	0x3B00
F2	0x3C00
F3	0x3D00
F4	0x3E00
F5	0x3F00
F6	0x4000

Key	Value
F7	0x4100
F8	0x4200
F9	0x4300
F10	0x4400
HOME	0x4700
UP	0x4800
PGUP	0x4900
LEFT	0x4B00
RIGHT	0x4D00
END	0x4F00
DOWN	0x5000
PGDOWN	0x5100
INSERT	0x5200
DELETE	0x5300

See scrhand.c (../../../samplecode/clib_sample/nlm/screen/scrhand.c.html).

See Also

getch (page 201), getche (page 202)

vcprintf

Writes output to the console under format control

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <stdarg.h>
#include <stdio.h>

int  vcprintf  (
   const char  *format,
   va_list      arg);
```

Parameters

`format`

(IN) Points to the format control string.

`arg`

(IN) Specifies a variable argument.

Return Values

The vcprintf function returns the number of characters written, or a negative value if an output error occurred. If an error occurs, `errno` is set.

Remarks

The vcprintf function writes output to the console under control of the argument `format`. The format string is described under the description for printf. The vcprintf function is similar to printf, with the variable argument list replaced with `arg`, which has been initialized by the va_start macro.

See Also

fprintf and printf (Single and Intra-File Services), sprintf, va_arg, va_end, va_start (NLM and NetWare: Program Management), vprintf (Single and Intra-File Services)

Example

The following example shows the use of vcprintf in a general error message routine.

```
#include   <stdarg.h>
#include   <stdio.h>
```

```
void errmsg (char *format, ... )
{
   va_list arglist;
   ConsolePrintf ("Error: ");
   va_start (arglist, format);
   vcprintf (format, arglist);
   va_end (arglist);
}
```

vcscanf

Scans input from the console under format control

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Device I/O

Syntax

```
#include <stdarg.h>
#include <stdio.h>

int  vcscanf  (
   const char   *format,
   va_list       arg);
```

Parameters

`format`

(IN) Points to the format control string.

`arg`

(IN) Specifies the variable argument.

Return Values

The vsscanf function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

Remarks

The vsscanf function scans input from the console under control of the argument `format`. The format list is described with the scanf function.

The vcscanf function is similar to the scanf function, with a variable argument list replaced with `arg`, which has been initialized using the va_start macro.

See Also

fscanf, scanf (Single and Intra-File Services), va_arg, va_end, va_start (NLM and NetWare: Program Management), vscanf (Single and Intra-File Services)

Example

```
#include <stdio.h>
#include <stdarg.h>

void find (char *format, char *arg, ... )
{
   va_list arglist;
   va_start (arglist, arg);
   vcscanf (format, arglist);
   va_end (arglist );
}
```

Screen Handling Concepts 16

This documentation describes Screen Handling, its functions, and features.

16.1 Screen Types

Multiple screens can exist on a server running NetWare®. The screen types are described below:

System Console Screen

Server console commands are entered at the command line of the System Console Screen. This screen is always present. On NetWare 5.x servers, NLMs can write to this screen and receive input from its keyboard. On NetWare 6.x servers, NLMs cannot write to this screen or receive input for it. They can write only to the System Logger Screen or to their application-owned screen.

System Logger Screen

This screen is only present on NetWare 6.x servers. This screen logs all system messages as well as the output from NLMs that write to the system console. NLMs cannot get characters from this screen's keyboard because the screen accepts only a few commands related to scrolling and other such activities.

Debug Screen

The Debug Screen is accessed from within an assembly or C program or through a special key sequence. This screen is hidden unless the server is at a breakpoint.

Router Screen

This screen displays whenever the TRACK ON console command is executed.

NLM Screens

An NLM can have zero or more regular or popup screens. Popup screens, used to present instructional or error messages, are overlaid on regular screens. In some cases, an NLM may not require a screen (a library NLM, for example). An NLM may also write to the System Console Screen or to the screen of another NLM (if the other NLM cooperates).

Switch between these screens in the following ways:

- Use Alt+Esc to switch from one screen to another.
- Use Ctrl+Esc to display a menu of screens from which a screen can be selected.

16.2 Creating Screens

CreateScreen (page 237) creates a screen. In addition to the screen’s actual contents (display), a screen is composed of a screen name, a set of screen attributes (characteristics), a command history buffer, and a type-ahead buffer.

16.2.1 Screen Names

Screen names can be specified with the linker directive SCREENNAME or by CreateScreen. The SCREENNAME directive can be used to specify the "initial" screen name, the name of the first screen that is automatically created when the NLM is loaded. If no screen name is specified, then the NLM description specified by the linker directive FORMAT is used. If the NLM creates other screens, the names of these screens are specified in the CreateScreen function using the `screenName` parameter.

A set of screen names that have special meanings can be used with the SCREENNAME directive and in the CreateScreen function.

The following screen names are special only if used with the SCREENNAME directive:

None - (Case insensitive)

If the SCREENNAME directive specifies "None," the NLM has no screens when it is started.

Default - (Case insensitive)

If the SCREENNAME directive specifies "Default," the *initial* screen of the NLM is the one that was current when the NLM was started. If the NLM is started from the system console, then the System Console Screen is considered current. If the NLM is spawned from another NLM, the current screen of the spawning NLM is used.

Other screen names are special when used either with the SCREENNAME directive or with the CreateScreen function. These include:

System Console - (Case sensitive)

If "System Console" is specified, the System Console Screen is used.

Screen names that start with two underscores (_ _)

These screen names are reserved.

16.2.2 Screen Attributes

Each screen has a set of attributes that specify the screen's behavior. The supported screen attributes are as follows:

AUTO_DESTROY_SCREEN (see "Automatic Screen Destruction" on page 222)
DONT_CHECK_CTRL_CHARS (see "Control-Character Checking" on page 222)
POP_UP_SCREEN (see "Popup Screens" on page 223)
UNCOUPLED_CURSORS (see "Cursor Coupling" on page 223)

Automatic Screen Destruction

If AUTO_DESTROY_SCREEN is set, the screen is destroyed when the NLM terminates. If this attribute is not set, the screen is not destroyed when the NLM terminates until the "Press any key to close screen" message is responded to.

Control-Character Checking

If DONT_CHECK_CTRL_CHARS is set, control characters <Ctrl><C> and <Ctrl><S> are not checked for. <Ctrl><C> terminates an NLM abnormally (using the abort function), and <Ctrl><S>

pauses output (output can be resumed by pressing any key). The following control characters are recognized whether or not control-character checking is enabled:

Tab
Carriage return
Linefeed
Backspace
Bell

Popup Screens

If POP_UP_SCREEN is set, the screen is a popup screen. (A popup screen automatically overlays the current screen.) If the popup screen is still displayed when DestroyScreen or DropPopUpScreen is called, then the screen that was overlayed is redisplayed.

Cursor Coupling

If UNCOUPLED_CURSORS is set, cursor coupling is disabled. The input and output cursors for the specified screen occupy separate positions on the screen. The position of the input cursor indicates the starting column and row position on the screen where the blinking cursor is located when a function that takes input from the keyboard is called. The output cursor indicates the starting column and row position on the screen where the output goes when a function that writes to the screen is called. When the cursors are uncoupled, the position of one cursor can be changed without affecting the other cursor's position.

When cursor coupling is enabled, the input and output cursors for the specified screen always occupy the same position. In effect, there is only one cursor for the screen.

16.2.3 Initial Screen Attribute Settings

By default, an NLM has one screen for its exclusive use when it begins. This screen, if it exists, is called the *initial* screen. The initial attribute settings for the initial screen are as follows:

- The screen is not destroyed when the NLM terminates until a key is pressed.
- Control-character checking is enabled.
- The screen is not a popup screen.
- Cursor coupling is enabled.

If the SCREENNAME directive specifies "System Console" or "Default," the initial screen can be the System Console Screen. The initial attribute settings for the System Console Screen are as follows:

- The screen is not destroyed when the NLM terminates.
- Control-character checking is disabled.
- The screen is not a popup screen.
- Cursor coupling is enabled.

Any input attempted from the System Console Screen causes an error. If the NLM calls any screen input function while the System Console Screen is the current screen, the function returns an error (-1).

An NLM can call the CreateScreen function to create other screens. If a screen is created with CreateScreen, then the attributes are specified by the `attributes` parameter.

16.2.4 Changing Screen Attributes

A screen's attributes can be changed with the following functions:

SetAutoScreenDestructionMode
SetCtrlCharCheckMode
SetCursorCouplingMode

The POP_UP_SCREEN attribute cannot be changed.

16.2.5 Type-Ahead and Command History Buffers

Each screen has its own type-ahead buffer and command history buffer.

- The type-ahead buffer holds input from the keyboard before it is processed by the NLM.
- The command history buffer saves strings entered from the keyboard. (The string-oriented input functions support this feature.) Previously entered strings can be retrieved using the Up- and Down-arrow keys and then can be edited.

16.3 Performing Screen I/O

In the NetWare environment, most functions that deal with a screen implicitly specify a target screen, although a few functions explicitly specify the screen.

- When a screen is implicitly specified, the *current* screen is the target screen. Functions involving screen operations process I/O to and from the current screen.

  All threads in a thread group have the same screen context. That is, all screens within a thread group access the current screen.
- A screen handle is used when explicitly specifying a target screen.

Any I/O done by a thread causes an implicit thread switch. The thread switch occurs before the actual I/O is processed. To ensure that the I/O from a thread that is part of a thread group goes to the correct screen, all I/O should be performed in critical sections of code. Critical code (bracketed between the EnterCritSec and ExitCritSec functions) prevents implicit thread switching from taking place.

16.3.1 Keyboard Input

For each screen, only one thread can wait on keyboard input from a given screen at a time. Any other thread that attempts input is blocked until the keyboard is free.

16.3.2 Screen Output

Any number of threads can do output to a single screen at a time. All output functions in the NetWare API usually complete their output before an output function called from another thread is allowed to write to the screen. The exception to this is a single call to a Stream I/O function (such as printf) that causes more data to be output to the screen than can fit in a Streams buffer: (default

buffer size: 512 bytes). This means that, in general, output from multiple threads is not scrambled together.

16.4 Destroying Screens

All of the NLM screens (except the System Console Screen or a screen inherited from another NLM) are destroyed when the NLM terminates. An NLM can call DestroyScreen to dispose of a screen at any time.

16.5 Screen Handling Function List

Screen Handling Functions 17

This documentation alphabetically lists the screen handling functions and describes their purpose, syntax, parameters, and return values.

CheckIfScreenDisplayed

Checks whether a screen is active

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int CheckIfScreenDisplayed  (
   int    screenHandle,
   long   waitFlag);
```

Parameters

`screenHandle`

(IN) Specifies the screen handle of the screen to check if active.

`waitFlag`

(IN) Specifies whether to wait for the screen to become active (displayed).

Return Values

When `waitFlag` = TRUE (1):

0 = Thread did not sleep.
1 = Thread did sleep.

When `waitFlag` = FALSE (0):

0 = Screen is not active.
1 = Screen is active.

If an error occurs, this function returns a value of -1 and `errno` is set to:

Value	Hex	Name	Description
22	(0x16)	EBADHNDL	Bad screen handle was passed in.

Remarks

The active screen is the screen currently being displayed on the server monitor.

The CheckIfScreenDisplayed function serves one of the following two purposes based on the value of the `waitFlag` parameter:

- When `waitFlag` is TRUE, this function suspends the calling thread until the screen specified by `screenHandle` is active (displayed). In this case, it returns TRUE if the calling thread slept and FALSE if the calling thread did not sleep (the screen was already active).
- When `waitFlag` is FALSE, this function merely checks to see if the screen specified by `screenHandle` is active.

Blocking Information This function is nonblocking unless `waitFlag` is set to TRUE.

See Also

DisplayScreen (page 242), GetCurrentScreen (page 245)

clrscr

Disables the cursor and clears the current screen (implemented for NetWare ® 3.0 and above)

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void clrscr  (void);
```

Return Values

None

If an error occurs, `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

The clrscr function clears the current screen and places the cursor (invisibly) in the upper left-hand corner (at position 0,0).

See Also

DisplayInputCursor (page 241)

Example

```
#include <stdlib.h>
#include <nwconio.h>
#include <stdio.h>
main()
{
   printf("type any character...");
   getch();
   clrscr();
   printf("this should be on a clear screen\r\n");
   getch();                      /* getch will reenable cursor */
}
```

ConsolePrintf

Writes a message to a console screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void ConsolePrintf  (
   const char   *format,
   ...);
```

Parameters

`format`

(IN) Points to the format control string.

Remarks

The ConsolePrintf function writes output under control of the argument `format`. The format string is described under the description of the printf function.

However, the format string has several limitations from that described under printf. The limitations are:

- Asterisk (*) is not allowed for the width or precision specification.
- No type length specifiers are allowed.
- The only format control flag allowed is "-", and the following conversions are not allowed:

e	E	f	F	g	G	i	n	p	X

- The \n character only performs a line-feed (with printf, \n performs carriage-return/line-feed.

On NetWare 5.x and earlier, the ConsolePrintf function writes output to the System Console Screen. On NetWare 6.0 and later, it writes output to the Logger Screen.

See Also

printf (Single and Intra-File Services)

CopyFromScreenMemory

Copies a rectangular region from screen memory

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void CopyFromScreenMemory  (
   WORD    height,
   WORD    width,
   BYTE   *Rect,
   WORD    beg_x,
   WORD    beg_y);
```

Parameters

height

(IN) Specifies the number of rows in the rectangular region.

width

(IN) Specifies the number of columns in the rectangular region.

Rect

(OUT) Points to the screen memory data.

beg_x

(IN) Specifies the starting column in the rectangular region.

beg_y

(IN) Specifies the starting row in the rectangular region.

Remarks

The CopyFromScreenMemory function copies a rectangular region, whose size is specified by `width` and `height`, from screen memory, starting from column `beg_x` and row `beg_y`.

Ensure that:

- `beg_x` + `width` is less than or equal to the number of columns on the screen (currently always 80).
- `beg_y` + `height` is less than or equal to the number of rows on the screen (currently always 25).

The rectangle is clipped to the screen's borders if it is too big.

If either `beg_x` or `beg_y` is greater than either SCREEN_COLUMNS or SCREEN_ROWS, the function returns without writing anything to the array `Rect`.

The size of the array `Rect` must be:

```
(2 * width * height)
```

The array `Rect` is an array of char attribute pairs:

```
struct cell
{
   char   charValue;
   char   attribute;
};
```

See Also

CopyToScreenMemory (page 235)

CopyToScreenMemory

Copies a rectangular region into screen memory

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void CopyToScreenMemory  (
   WORD    height,
   WORD    width,
   BYTE   *Rect,
   WORD    beg_x,
   WORD    beg_y);
```

Parameters

height

(IN) Specifies the number of rows in the rectangular region.

width

(IN) Specifies the number of columns in the rectangular region.

Rect

(IN) Points to the data to be copied into screen memory.

beg_x

(IN) Specifies the starting column in the rectangular region.

beg_y

(IN) Specifies the starting row in the rectangular region.

Remarks

The CopyToScreenMemory function copies a rectangular region, whose size is specified by `width` and `height`, into screen memory, starting from column `beg_x` and row `beg_y`.

Ensure that:

- `beg_x` + `width` is less than or equal to the number of columns on the screen (currently always 80).
- `beg_y` + `height` is less than or equal to the number of rows on the screen (currently always 25).

The rectangle is clipped to the screen's borders if it is too big.

If either `beg_x` or `beg_y` is greater than either SCREEN_COLUMNS or SCREEN_ROWS, the function returns without doing anything to the screen.

The size of the array `Rect` must be:

```
(2 * width * height)
```

The array `Rect` is an array of char attribute pairs:

```
struct cell {
   char   charValue;
   char   attribute;
};
```

See Also

CopyFromScreenMemory (page 233)

CreateScreen

Creates a new screen

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int CreateScreen  (
   char  *screenName,
   BYTE   attributes);
```

Parameters

screenName

(IN) Points to the name of the new screen.

attributes

(IN) Specifies the screen attributes.

Return Values

Returns the screen handle if successful or EFAILURE if an error occurs. If a NULL value is returned, the screen handle cannot be returned and `errno` is set to ENOMEM.

Remarks

A new screen is created for use by the NLM™ application. The new screen is displayed and made the current screen only if no other screens exist for the NLM when this call is made; otherwise, the current and displayed screens remain unchanged.

If a screen has the DONT_AUTO_ACTIVATE attribute set, it is not automatically displayed when it is created, even if it is the only screen for the NLM.

The supported screen attributes are:

Attribute	Description
DONT_AUTO_ACTIVATE	Prevents auto activation when screens are created and no other screens exist.
DONT_CHECK_CTRL_CHARS	Turns off Ctrl-C and Ctrl-S processing.
AUTO_DESTROY_SCREEN	Prevents the "Press any key to close" message.

Attribute	Description
POP_UP_SCREEN	Makes the screen a pop up screen.
UNCOUPLED_CURSORS	Sets distinct input and output cursors.

A popup screen automatically overlays the currently displayed screen. If the popup screen is still displayed when the DestroyScreen function or DropPopUpScreen (for the popup screen) function is called, the screen that was overlayed is redisplayed.

If `screenName` is "System Console" (case sensitive), a new screen is not created, rather the returned screen handle refers to the System Console Screen. In this case, the attributes should be set to zero. Input is not allowed from the System Console Screen. (All the input functions return EFAILURE with `errno` set to EWRNGKND.)

NOTE: If you pass a valid OS screen ID (usually obtained by other functions in this module) in the `screenName` parameter, CreateScreen creates a C Library screen handle from the given screen ID.

See scrhand.c (../../../samplecode/clib_sample/nlm/screen/scrhand.c.html).

See Also

DestroyScreen (page 239), DisplayScreen (page 242), GetCurrentScreen (page 245), __GetScreenID (page 250), GetScreenInfo (page 251), ScanScreens (page 261), SetCurrentScreen (page 267)

Example

```
#include <nwconio.h>
char   screenName[ ] = "NLM x New Screen";
int    screenHandle;
BYTE   attributes = 0;
screenHandle = CreateScreen(screenName, attributes);
```

DestroyScreen

Closes a screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int DestroyScreen  (
   int   screenHandle);
```

Parameters

`screenHandle`

(IN) Specifies the screen handle of the screen being closed.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
22	(0x16)	EBADHNDL	Bad screen handle was passed in.

Remarks

DestroyScreen closes the screen specified by the `screenHandle` parameter. The following conditions determine which screen is displayed next:

- If the `screenHandle` parameter specifies the current screen, then a new current screen is set if the NLM has any screens left.
- If the `screenHandle` parameter specifies the screen that is displayed, then another one of the screens of the NLM is displayed if any are left. Otherwise, the System Console Screen is displayed.
- If the `screenHandle` parameter specifies a popup screen that is displayed, the screen that was covered by the popup screen is redisplayed if it still exists. Otherwise, the System Console Screen is displayed.

See Also

CreateScreen (page 237)

DisplayInputCursor

Enables the input cursor for the current screen

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int DisplayInputCursor  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
19	(0x13)	EWRKGKND	Current screen is the System Console Screen. Input is not allowed.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

This function makes the input cursor of the current screen visible when the screen is next displayed. If another thread is waiting on the keyboard, this function waits until the keyboard is free.

See Also

HideInputCursor (page 256)

DisplayScreen

Displays the specified screen

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int DisplayScreen  (
   int   screenHandle);
```

Parameters

`screenHandle`

(IN) Specifies the screen handle of the screen to display; if NULL is specified and the current screen is a popup screen, then a DropPopUpScreen is done on the current screen.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
22	(0x16)	EBADHNDL	Bad screen handle was passed in.

WARNING: An invalid screen handle is not guaranteed to return EBADHNDL; it can also cause the server to abend. Always pass a handle returned from CreateScreen or GetScreenInfo.

Remarks

In addition to displaying the specified screen, this function also makes the specified screen the current screen.

If the `screenHandle` parameter specifies a popup screen:

- The specified popup screen is displayed over the currently displayed screen (original screen).
- When the DropPopUpScreen or DestroyScreen function is called for the popup screen, and the popup screen is displayed, the original screen, if it still exists, is redisplayed. Otherwise, the System Console Screen is displayed.

See Also

CheckIfScreenDisplayed (page 229), DestroyScreen (page 239)

DropPopUpScreen

Redisplays the screen that the popup screen covered

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int DropPopUpScreen  (
   int   screenHandle);
```

Parameters

screenHandle

(IN) Specifies the screen handle of the popup screen.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
22	(0x16)	EBADHNDL	Bad screen handle was passed in.
105	(0x69)	ERR_NOT_A_POPUP_SCREEN	

WARNING: An invalid screen handle is not guaranteed to return EBADHANDLE; it can also cause the server to abend. Always pass a handle returned from CreateScreen or GetScreenInfo.

Remarks

This function redisplays the screen the popup screen covered if the popup screen is the displayed screen when this function is called and the covered screen still exists. In addition, the screen that was covered is made current if it is a screen owned by the calling NLM.

See Also

DestroyScreen (page 239), DisplayScreen (page 242)

GetCurrentScreen

Returns the screen handle of the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int GetCurrentScreen  (void);
```

Return Values

This function returns the screen handle of the current screen if successful.

If an error occurs this function returns NULL, and `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

GetCurrentScreen returns the handle of the current screen.

NOTE: The handle returned pertains only to the current screen of the current NLM. It is not necessarily the handle of the screen displayed on the console.

See Also

SetCurrentScreen (page 267)

GetCursorCouplingMode

Returns whether cursor coupling is enabled or disabled for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE GetCursorCouplingMode  (void);
```

Return Values

This function returns the cursor coupling mode if successful; otherwise, it returns EFAILURE.

If an error occurs, `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

This function returns TRUE if cursor coupling is enabled, and FALSE if cursor coupling is disabled for the current screen.

See Also

SetCursorCouplingMode (page 268)

Example

```
#include <nwconio.h>
BYTE   newMode;
newMode = GetCursorCouplingMode();
```

GetCursorShape

Returns the start and end scan line for the cursor

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

WORD GetCursorShape  (
   BYTE   *startline,
   BYTE   *endline);
```

Parameters

`startline`

(OUT) Points to the first cursor scan line.

`endline`

(OUT) Points to the last cursor scan line.

Return Values

This function returns the scan line for the cursor.

Remarks

The GetCursorSize function returns the cursor shape as specified by the `startline` and `endline` parameters.

See Also

GetCursorSize (page 248), SetCursorShape (page 269)

Example

```
#include <nwconio.h>
WORD   scanline;
BYTE   startline;
BYTE   endline;
scanline = GetCursorShape (&startline, & endline);
```

GetCursorSize

Returns the cursor size

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

WORD GetCursorSize  (
   BYTE  *firstline,
   BYTE  *lastline);
```

Parameters

`firstline`

(OUT) Receives the first cursor scan line.

`lastline`

(OUT) Receives the last cursor scan line.

Return Values

This function returns the cursor size.

Remarks

The GetCursorSize function returns the maximum (`lastline`) and minimum (`firstline`) values that the cursor scan lines can be set to.

See Also

SetCursorShape (page 269)

Example

```
#include <nwconio.h>
WORD   scanline;
BYTE   firstline;
BYTE   lastline;
scanline = GetCursorSize (&firstline, & lastline);
```

GetPositionOfOutputCursor

Returns the output cursor's current row and column position for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int GetPositionOfOutputCursor  (
   WORD   *row,
   WORD   *column);
```

Parameters

row

(OUT) Points to the row on which the cursor is positioned (first row is 0).

column

(OUT) Points to the column on which the cursor is positioned (first column is 0).

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
23	(0x17)	ENO_SCRNS	No screens were open.

Remarks

This function returns the output cursor's position on the current screen; it also returns the input cursor's position if cursor coupling is enabled for the current screen.

See Also

gotoxy (page 254)

__GetScreenID

Returns the screen ID for a screen handle

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int _ _GetScreenID  (
   int   screenHandle);
```

Parameters

screenHandle

(IN) Specifies a handle of a C Library Open Screen Structure.

Return Values

The function returns the OS screen ID related to the C Library screen.

Remarks

The value returned by this function can be passed to functions that take a screen ID.

See Also

CreateScreen (page 237), GetScreenInfo (page 251), ScanScreens (page 261)

GetScreenInfo

Returns the screen handle associated with the specified screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int GetScreenInfo  (
   int     screenID,
   char   *name,
   LONG   *attrib);
```

Parameters

screenID

(IN) Specifies a screen ID (an OS structure).

name

(OUT) Points to the name of the screen. Names of nonC Library screens are also returned (for example, MONITOR.NLM's screen).

attrib

(OUT) Points to the attributes of a given screen ID. If there is a valid C Library screen handle associated with this screen ID, then the screen handle's attributes are returned as well.

Return Values

This function returns the screen handle associated with the specified screen. If the screen handle is nonzero, then it can be passed to functions that take a C Library screen handle. If the function returns a NULL value, there is no C Library equivalent of the specified screen. That is, the screen was not opened by CreateScreen.

A return value of -1 indicates the screen ID was not a valid OS screen ID, and `errno` is set to EBADHNDL.

WARNING: An invalid screen ID is not guaranteed to return EBADHANDLE; it can also cause the server to abend.

Remarks

You can pass NULL values in any parameter.

The following are C Library settable attribute bits. These can be returned for C Library screens.

Attribute	Description
DONT_CHECK_CTRL_CHARS	Overrides the control characters (Ctrl+C, Ctrl +S) and tab processing.
AUTO_DESTROY_SCREEN	Causes the <Press any key to close> message.
POP_UP_SCREEN	Sets screen to be a popup screen.
UNCOUPLED_CURSORS	Sets distinct and separate input and output cursors.

The following attribute can be set bit if there is a related C Library screen:

HAS_A_CLIB_HANDLE

The following are OS attribute bits. These cannot be set using C Library APIs.

_KEYBOARD_INPUT_ACTIVE
_PROCESS_BLOCKED_ON_KEYBOARD
_PROCESS_BLOCKED_ON_SCREEN
_INPUT_CURSOR_DISABLED
_SCREEN_HAS_TITLE_BAR
_NON_SWITCHABLE_SCREEN

See Also

CreateScreen (page 237), __GetScreenID (page 250), ScanScreens (page 261)

GetSizeOfScreen

Returns the number of rows and columns of the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int GetSizeOfScreen  (
   WORD   *height,
   WORD   *width);
```

Parameters

height

(OUT) Points to the number of rows in the screen (first column is 0)

width

(OUT) Points to the number of columns in the screen (first column is 0)

Return Values

Value	Hex	Name
0	(0x00)	ESUCCESS

Remarks

This function returns the size of the current screen. Currently all screens are 25x80.

See Also

DisplayScreen (page 242)

gotoxy

Positions the output cursor on the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void gotoxy  (
   WORD   column,
   WORD   row);
```

Parameters

column

(IN) Specifies the column on which to position the cursor.

row

(IN) Specifies the row on which to position the cursor.

Return Values

This function returns no value.

If an error occurs, `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	No screens were open.

Remarks

The output cursor is positioned on the current screen. If cursor coupling is enabled for the current screen, the input cursor is also positioned.

NOTE: The order of the row and column parameters is different from all the other functions that take row and column arguments.

See Also

SetOutputAtInputCursorPosition (page 271), SetPositionOfInputCursor (page 272)

Example

```
#include   <stdlib.h>
#include   <nwconio.h>
main()
{
   gotoxy(10,10);
   printf("A");
   gotoxy(50,10);
   printf("B");
   getch();
}
```

HideInputCursor

Disables the input cursor for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int HideInputCursor  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
19	(0x13)	EWRNGKND	Input to System Console was attempted.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

This function causes the input cursor to be invisible when the current screen is displayed.

See Also

DisplayInputCursor (page 241), GetPositionOfOutputCursor (page 249), SetPositionOfInputCursor (page 272)

IsColorMonitor

Determines whether a color monitor is being used

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void IsColorMonitor  (void);
```

Return Values

This function returns a value of 1 if the machine is using a color monitor; otherwise, it returns a value of 0.

PressAnyKeyToContinue

Writes the message <Press any key to continue> to the current screen

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int PressAnyKeyToContinue  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
19	(0x13)	EWRNGKND	Input to System Console was attempted.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

When a key is pressed, the <Press any key to continue> message is cleared and normal screen activity resumes. The thread is blocked until a key is pressed.

NOTE: If another thread causes the screen to scroll before a key is pressed, the <Press any key to continue> message might not be properly erased.

See Also

getch (page 201), PressEscapeToQuit (page 259)

PressEscapeToQuit

Writes the message <Press Escape to quit or any key to continue> to the current screen

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int PressEscapeToQuit  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Any key other than Escape was pressed.
1			Escape was pressed.
19	(0x13)	EWRNGKND	Input to System Console was attempted.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

Pressing the Escape key clears the <Press Escape to quit or any key to continue> message and the user can terminate the NLM depending on the return value.

See Also

getch (page 201), PressAnyKeyToContinue (page 258)

RingTheBell

Causes the console speaker to beep

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12 and above, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void RingTheBell  (void);
```

Return Values

This function does not return a value.

Remarks

This function can be repeated several times in a row, to increase the duration of the beep.

Example

```
#include <stdio.h>
#include <nwconio.h>
main()
{
   printf("\nError\n");
   RingTheBell();
}
```

ScanScreens

Returns a screen ID (a pointer to an OS screen structure) associated with the specified screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int ScanScreens  (
   int     LastScreenID,
   char   *name,
   LONG   *attrib);
```

Parameters

`LastScreenID`

(IN) Specifies a screen ID obtained by a previous ScanScreens call (or NULL to get the first screen ID).

`name`

(OUT) Points to the name of the screen.

`attrib`

(OUT) Points to the attributes of the given screen ID.

Return Values

This function returns the screen ID of the next screen on the list. If it returns a NULL value, there are no more screen IDs, or an invalid screen ID has been passed to the function, and `errno` is set to EBADHNDL.

Remarks

This function is used to get the next member on the list of the OS screen IDs.

When calling this function, pass a NULL value to obtain the first screen ID on the list. (Currently, it is always the system console's ID. However, this might change in the future.)

You can also pass NULL values in the `name` and `attrib` parameters.

See Also

CreateScreen (page 237)

Example

```
/*
   This example demonstrates using the Screen ID/Screen Handle
   Conversion APIs. This program looks for all the screens
   in the system and then prints on those screens their
   names and attributes.
*/

#include <errno.h>
#include <nwtypes.h>
#include <nwconio.h>
#include <stdio.h>
#include <nwthread.h>
#define property (x) if (att & x) printf ("%-40s\n", #x)
main ()
{
   int  sID;
   int  sh;
   char buf[80];
   LONG attr;
   for (sID = NULL; sID = ScanScreens (sID, buf, &attr);)
   {
      sh = GetScreenInfo (sID, NULL, NULL);
      /* there is no CLIB equivalent? */
      if (!sh)
      sh = CreateScreen ((char*) sID, 0);
      /* let's create one */
      if (!sh)
      {
         ConsolePrintf ("errno: %d\n", errno);
         abort();
      }
      SetCurrentScreen (sh);
      gotoxy (1,1);
      printf ("This screen is %s with these attributes:\n\r", buf);
      property(HAS_A_CLIB_HANDLE);
      property(_KEYBOARD_INPUT_ACTIVE);
      property(_PROCESS_BLOCKED_ON_KEYBOARD);
      property(_PROCESS_BLOCKED_ON_SCREEN);
      property(_INPUT_CURSOR_DISABLED);
      property(_SCREEN_HAS_TITLE_BAR);
      property(_NON_SWITCHABLE_SCREEN);
      property(DONT_CHECK_CTRL_CHARS);
      property(AUTO_DESTROY_SCREEN);
      property(POP_UP_SCREEN);
      property(UNCOUPLED_CURSORS);
      DestroyScreen (sh);
   }
/*
getchar(); It abends if another process is doing input on this screen.
*/
}
```

ScrollScreenRegionDown

Scrolls down a portion of the current screen (a set of contiguous rows)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int ScrollScreenRegionDown  (
   int   firstLine,
   int   numberOfLines);
```

Parameters

`firstLine`

(IN) Specifies the row number of the first row in the set. The top row of the screen is 0 (zero).

`numberOfLines`

(IN) Specifies the number of rows in the region (in set).

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

This function scrolls a portion of the screen down. (The bottom line of the region is replaced by the next-to-the-bottom line. The next-to-the-bottom line is replaced by the line above it, and so on. Finally, the first line of the region is cleared.) All of the lines on the screen that are not in the defined region are not affected.

See Also

CopyToScreenMemory (page 235), SetScreenRegionAttribute (page 279)

ScrollScreenRegionUp

Scrolls up a portion of the current screen (a set of contiguous rows)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int ScrollScreenRegionUp  (
   int   firstLine,
   int   numberOfLines);
```

Parameters

`firstLine`

(IN) Specifies the row number of the first row in the set. The top row of the screen is 0.

`numberOfLines`

(IN) Specifies the number of rows in region (in set).

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

This function scrolls a portion of the screen up. (The top line of the region is replaced by the next-to-the-top line. The next-to-the-top line is replaced by the line below it, and so on. Finally, the bottom line of the region is cleared.) All of the lines on the screen that are not in the defined region are not affected.

See Also

CopyToScreenMemory (page 235), SetScreenRegionAttribute (page 279)

SetAutoScreenDestructionMode

Enables or disables auto-screen destruction for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE SetAutoScreenDestructionMode  (
   BYTE   newMode);
```

Parameters

newMode

(IN) Specifies TRUE = Enable auto-screen destruction. FALSE = Disable auto-screen destruction.

Return Values

This function returns the value of the old auto-screen destruction mode setting.

If an error occurs, the function returns a value of -1, and `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

If auto-screen destruction is disabled for a particular screen while the NLM is terminating, that screen remains open with the message <Press any key to close screen >. The screen does not close and the NLM does not continue with its termination until a key is pressed on that screen.

SetCtrlCharCheckMode

Enables or disables control-character checking for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE SetCtrlCharCheckMode  (
   BYTE   newMode);
```

Parameters

newMode

(IN) Specifies TRUE = Enable control-character checking. FALSE = Disable control-character checking.

Return Values

This function returns the value of the old control-character check mode setting.

If an error occurs, the function returns -1 and `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

Set the `newMode` parameter to TRUE if control characters are to be checked for, and FALSE otherwise. The control characters to check for are Ctrl+C and Ctrl+S.

- Ctrl+C terminates the NLM abnormally (via the abort function).
- Ctrl+S pauses output (pressing any key resumes output).

SetCurrentScreen

Sets the current screen of the thread group belonging to the running thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetCurrentScreen  (
   int   screenHandle);
```

Parameters

`screenHandle`

(IN) Specifies the screen handle of the screen to make current.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
22	(0x16)	EBANHNDL	Bad screen handle was passed in.

Remarks

This function sets the screen specified by the `screenHandle` parameter as the target of screen I/O functions. It does not change the displayed screen.

See scrhand.c (../../../samplecode/clib_sample/nlm/screen/scrhand.c.html).

See Also

CreateScreen (page 237), DestroyScreen (page 239), DisplayScreen (page 242)

SetCursorCouplingMode

Enables or disables input and output cursor coupling for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE SetCursorCouplingMode  (
   BYTE   newMode);
```

Parameters

newMode

(IN) Specifies TRUE = Enable cursor coupling. FALSE = Disable cursor coupling.

Return Values

This function returns the value of the old cursor-coupling mode setting.

If an error occurs, the function returns a value of -1, and `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

When cursor coupling is disabled, the input and output cursors for the specified screen occupy separate positions on the screen. The position of the input cursor indicates the starting column/row position on the screen where the blinking cursor is located when a function that takes input from the keyboard is called. The output cursor indicates the starting column/row position on the screen where the output goes when a function that writes to the screen is called. The position of one cursor can be changed without affecting the other cursor's position.

When cursor coupling is enabled, the input and output cursors for the specified screen always occupy the same position. In effect, there is only one cursor for the screen.

See Also

GetCursorCouplingMode (page 246)

SetCursorShape

Sets the cursor shape

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

WORD SetCursorShape  (
   BYTE   startline,
   BYTE   endline);
```

Parameters

`startline`

(IN) Specifies the first cursor scan line.

`endline`

(IN) Specifies the last cursor scan line.

Return Values

This function returns the old cursor shape.

Remarks

The SetCursorShape function sets the cursor shape as specified by the `startline` and `endline` parameters.

See Also

GetCursorShape (page 247)

SetInputAtOutputCursorPosition

Sets the input cursor position to the output cursor position

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetInputAtOutputCursorPosition  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
19	(0x13)	EWRNGKND	Input from the System Console was attempted.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

The input cursor position is set to the output cursor position on the current screen. If another thread is waiting on the keyboard, the current thread waits until the keyboard is free.

See Also

DisplayInputCursor (page 241), GetPositionOfOutputCursor (page 249), gotoxy (page 254), HideInputCursor (page 256), SetOutputAtInputCursorPosition (page 271), wherex (page 281), wherey (page 282)

SetOutputAtInputCursorPosition

Sets the output cursor position to the input cursor position

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetOutputAtInputCursorPosition  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

The output cursor position is set to the input cursor position on the current screen.

See Also

GetPositionOfOutputCursor (page 249), gotoxy (page 254), SetInputAtOutputCursorPosition (page 270), wherex (page 281), wherey (page 282)

SetPositionOfInputCursor

Sets the position of the input cursor on the current screen

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetPositionOfInputCursor  (
   WORD   row,
   WORD   column);
```

Parameters

row

(IN) Specifies the row number on which to position the cursor (top row is 0).

column

(IN) Specifies the column number on which to position the cursor (leftmost column is 0).

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
19	(0x13)	EWRNGKND	Input to System Console was attempted.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

The application must check that the row and column positions are within the current screen size.

The SetPositionOfInputCursor function positions the input cursor on the current screen. It also positions the output cursor if cursor coupling for the current screen is enabled. If another thread is waiting on the keyboard, the calling thread is blocked until the keyboard is free.

See Also

DisplayInputCursor (page 241), HideInputCursor (page 256)

SetScreenAttributes

Sets the display attribute bytes for the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetScreenAttributes (
   LONG   mask,
   LONG   attr);
```

Parameters

mask

(IN) Specifies the attributes you want to set or clear:

Value	Bit	Name	Description
0x01	Bit 1	DONT_AUTO_ACTIVATE	Avoids autoactivation when screens are created, but no other screens exist.
0x02	Bit 2	DONT_SWITCH_SCREEN	Avoids screen being switched. Converted to _NON_SWITCHABLE_SCREEN.
0x10	Bit 4	DONT_CHECK_CTRL_CHARS	Turns off ^C and ^S processing.
0x20	Bit 5	AUTO_DESTROY_SCREEN	Avoids "Press any key to close screen."
0x80	Bit 7	UNCOUPLED_CURSORS	Displays distinct input and output cursors.

attr

(IN) Specifies the attributes to set, using the same values as the `mask` parameter. To set an attribute, it must be specified in both the `mask` and `attr` parameters. To clear an attribute, it must be specified in only the `mask` paramter.

Return Values

Returns a value of ESUCCESS (0) if successful. Otherwise, it returns a nonzero value.

Remarks

SetScreenAttributes sets and clears various bit flags. For example:

```
SetScreenAttributes(DONT_SWITCH_SCREEN|AUTO_DESTROY_SCREEN,
DONT_SWITCH_SCREEN);
```

This sets the DONT_SWITCH_SCREEN attribute and clears the AUTO_DESTROY_SCREEN attribute. Bit attributes that are not specified in `mask` are not affected.

Before setting the DONT_SWITCH_SCREEN bit, you need to know if the screen is a popup screen (use GetScreenInfo). If the screen is a popup screen, you need to call DropPopUpScreen one or more times. Each call to DropPopUpScreen decrements the popup screen in-use count. When the in-use count reaches 0, the function sets the DONT_SWITCH_SCREEN bit on the popup screen.

See Also

SetScreenAreaAttribute (page 275), SetScreenCharacterAttribute (page 277), SetScreenRegionAttribute (page 279)

SetScreenAreaAttribute

Sets the display adapter attribute bytes for an area of the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

Long SetScreenAreaAttribute  (
   LONG   line,
   LONG   column,
   LONG   numLines,
   LONG   numColumns,
   LONG   attribute);
```

Parameters

line

(IN) Specifies the row number of the top line in the screen area.

column

(IN) Specifies the column number of the left column in the screen area.

numLines

(IN) Specifies the number of rows to be included in the screen area.

numColumns

(IN) Specifies the number of columns to be included in the screen area.

attribute

(IN) Specifies the value of the attribute to be set. This value depends upon the type of monitor present.

Return Values

This function returns a value of ESUCCESS (0) if successful. Otherwise, it returns a nonzero value.

Remarks

This function changes the attribute for characters that have been sent to the specified area of the screen. Whenever you send a character to a screen, the output is written with a white-on-black attribute (0x07). If you want to change the attribute for the characters in that area, you must call this function after you write the characters to the screen.

See Also

SetScreenCharacterAttribute (page 277), SetScreenRegionAttribute (page 279)

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <nwconio.h>
#include <nwthread.h>
main()
{
   int   i;
   for (i = 0; i < 14;i++)
   {
      gotoxy(i, i);
      printf("COLOR TEST");
      SetScreenCharacterAttribute(i, i, i);
      SetScreenAreaAttribute(0, 64, i, 14-i, i*16);
      getch() /* to create a pause between colors */
   }
}
```

SetScreenCharacterAttribute

Sets the display adapter attribute bytes for a character on the current screen

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

Long SetScreenCharacterAttribute  (
   LONG   line,
   LONG   column,
   LONG   attribute);
```

Parameters

line

(IN) Specifies the row number of the character's position.

column

(IN) Specifies the column number of the character's position.

attribute

(IN) Specifies the value of the attribute to be set. This value depends upon the type of monitor present.

Return Values

This function returns a value of ESUCCESS (0) if successful. Otherwise, it returns a nonzero value.

Remarks

This function changes the attribute for a character that has been sent to the screen. Whenever you send a character to a screen, the output is written with a white-on-black attribute (0x07). If you want to change the attribute for the character, you must call this function after you write the character to the screen.

See Also

SetScreenAreaAttribute (page 275), SetScreenRegionAttribute (page 279)

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <nwconio.h>
#include <nwthread.h>
main()
{
   int   i;
   for (i = 0; i < 14;i++)
   {
      gotoxy(i, i);
      printf("COLOR TEST");
      getch();    /* to create a pause between colors */
      SetScreenCharacterAttribute(i, i, i);
   }
}
```

SetScreenRegionAttribute

Sets the display adapter attribute bytes for a region of the current screen (contiguous set of rows)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetScreenRegionAttribute  (
   int    firstLine,
   int    numberOfLines,
   BYTE   attribute);
```

Parameters

`firstLine`

(IN) Specifies the row number of the first row in the set. The top row of the screen is 0 (zero).

`numberOfLines`

(IN) Specifies the number of rows in region (in set).

`attribute`

(IN) Specifies the value of attribute to set; value depends on the type of monitor that is present (see the IBM Technical Reference for the Personal Computer XT).

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Successful.
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

Whenever output to a screen is performed, the output is written with white-on-black attribute (0x07). This nullifies the effect of this function. Therefore, this function should be called after the screen is written to.

See Also

ScrollScreenRegionDown (page 263)

wherex

Returns the horizontal position of the input cursor

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

WORD wherex  (void);
```

Return Values

This function returns the current column of the input cursor if successful. If an error occurs, it returns EFAILURE.

If an error occurs, `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

The wherex function returns the x coordinate of the current input cursor position (within the current screen). It also returns the output cursor's position if cursor coupling for the current screen is enabled.

See Also

SetPositionOfInputCursor (page 272), wherey (page 282)

Example

```
#include <stdlib.h>
#include <nwconio.h>
#include <stdio.h>

main()
{
   printf("%d,%d\r\n",wherex(),wherey());
   getch();
}
```

wherey

Returns the vertical position of the input cursor

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Screen Handling

Syntax

```
#include <nwconio.h>

WORD wherey  (void);
```

Return Values

This function returns the current row of the input cursor if successful. If an error occurs, it returns EFAILURE.

If an error occurs, `errno` is set to:

Value	Hex	Name	Description
23	(0x17)	ENO_SCRNS	Screen I/O was attempted when no screens were open.

Remarks

The wherey function returns the `y` coordinate of the current input cursor position (within the current screen). It also returns the output cursor's position if cursor coupling for the current screen is enabled.

See Also

SetPositionOfInputCursor (page 272), wherex (page 281)

Example

```
#include <stdlib.h>
#include <nwconio.h>

main()
{
   printf("%d,%d\r\n",wherex(),wherey());
   getch();
}
```

Revision History

A

The following table outlines all the changes that have been made to the NLM Development Concepts, Tools, and Functions documentation (in reverse chronological order):

Release Date	Revision Description
February 2008	Removed the section for registering an NLM prefix.
October 11, 2006	Removed link to Software Test Tools.
March 1, 2006	Updated format.
October 5, 2005	Transitioned to revised Novell documentation standards.
March 2, 2005	Modified the documentation for the SetScreenAttributes (page 273) function.
October 6, 2004	Removed the documentation for the assert_action function, because CLib does not support it.
	Removed documentation for the server -o option, because this parameter is no longer used by the NetWare operating system.
	Added information to the ScanSetableParameters (page 154) function.
June 9, 2004	Added a description for the 0x01000000 linker flag. See "FLAG ON and FLAG OFF Parameters" on page 30.
	Modified the installation instructions to remove references to LibC (see Section 1.7, "Installing the CLib Files on a NetWare Server," on page 32). CLib and LibC releases are now independent of each other.
October 8, 2003	Made technical corrections to "Prelude Object Files" on page 43.
July 30, 2003	Added information to ConsolePrintf (page 232), indicating that on NetWare 6 and later, this function prints to the System Logger Screen.
	Restructured the compiler, linker, and testing information to create a Getting Started section.
	Added a description of the CLIB manuals to Section 2.6, "Introduction to CLIB," on page 42, indicating which manuals contain documentation for cross-platform functions and NLM-only functions.
June 2003	Removed references to the NetWare Remote Debugger because it is no longer supported and is now located in the NDK Graveyard. Added information about CodeWarrior (see "Setting Up CodeWarrior for NLM Development" on page 15).
March 2003	Added information to the GetFileHoleMap (page 130) function and the SetScreenAttributes (page 273) function.
September 2002	Added information to Section 13.3, "Alert ID Values," on page 180.
May 2002	Updated the examples in Step 14 of "Using the WATCOM IDE" on page 18.
	Added links to all subsections.

Release Date	Revision Description
February 2002	Removed the "NLM Message Tools for Internationalization" section from “Basic NLM Concepts” on page 35.
	Updated links.
October 2001	Removed references to nwsmp.h—an obsolete header file.
September 2001	Added text alternatives to figures.
	Added NetWare 6.x support in documentation.
	Added “Setting Environment Variables” on page 41 section.
	Changed the description of EVENT_LOGOUT_CONNECTION (#38) in RegisterForEvent (page 148).
	Removed Marshaling Functions since they were never fully implemented.
June 2001	Added explanation of the mask and attribute parameters to SetScreenAttributes (page 273).
	Changed the description of EVENT_LOGOUT_CONNECTION (#38) in RegisterForEvent (page 148).
	Added introductory text to Chapter 2, “Basic NLM Concepts,” on page 35.
	Made changes to improve document accessibility.
February 2001	Added documentation for assert_action and Chapter 17, “Screen Handling Functions,” on page 227.
	Deleted obsolete information.
	Added descriptions for two fields of T_DYNARRAY_BLOCK (page 177).
	Corrected the syntax of outp (page 209) and outpw (page 212).
May 2000	Added revision history.