NLM Programming

NLM Programming Overview

NLM Development

NLM Development: Guides

NLM Development Tools

NLM Development Tools: Guides

Advanced

Advanced: Guides

Advanced: Functions

Advanced: Structures

Bit Array

Bit Array: Functions

Character Manipulation

Character Manipulation: Functions

Device I/O

Device I/O: Functions

Library

Library: Guides

Library: Functions

Mathematical Computation

Mathematical Computation: Functions

Mathematical Computation: Structures

Memory Allocation

Memory Allocation: Functions

Memory Manipulation

Memory Manipulation: Functions

NLM Debug

NLM Debug: Functions

NWSNUT

NWSNUT: Guides

NLM Development

NLM Development: Guides

NLM Programming Overview Guide

The NetWare® 3.x and 4.x OS's are a platform, consisting of core services such as process management, memory management, and screen management. To this platform of core services, you can add building blocks of functionality---such as communication/service protocols, file and directory systems, naming conventions, LAN and disk drivers, and other value-added services. This way, you can create a custom network OS, suited to a particular environment or situation.

- NLM Applications
- Types of NLM Applications
  - Utility and Service Modules
  - LAN and Device Driver Modules
  - Name Space Modules
- Loading NLM Applications
  - How NLM Applications Are Loaded
  - Using the LOAD Command
  - Unloading NLM Applications

NLM Development

- NLM Programming Overview
- Client-Server Applications
- NLM Code Development
- SFT III
- NLM Assembly Interface on the Intel Platform
- NLM Optimization for the Intel Platform
- Controlling the Server Clock under NetWare 4.x

Modular CLIB Header Files

Renamed NLM Functions

Remote Server Support

VAP to NLM Conversion

Client-Server Applications

Introduction to Client-Server Applications

Designing Client-Server NLM Applications

Locating Services

Client-Server Communication

NLM Code Development

Includes compiler issues, OS design features that should be considered by NLM™ developers, and features of the NetWare® OS.

Once you understand the model of NLM applications, you can begin writing modules that take advantage of the multithreaded environment that the NetWare OS and the NetWare API provide for you. To locate the information that you need, see the SDK Roadmap.

Programming Languages

NetWare API

- Modular CLIB
- Cross-Platform Functions in NLM Applications

Development Steps

OS-Related Issues

- Nonpreemptive Environment
- Protected or Nonprotected Environment
- Current Working Directory
- Connection and Task Numbers
- Screens and the NetWare OS

SFT III

Many applications work, as is, with SFT III™ software. This information will help you understand the issues involved with developing NLM™ applications that run on NetWare® SFT III versions 3.11 and 4.x.

NLM Assembly Interface on the Intel Platform

NLM Optimization for the Intel Platform

Part of optimizing the performance of your NLM™ application is eliminating wasted instructions, or clock cycles. These topics discuss how to design data structures so that the operations performed on them take the fewest clock cycles.

Controlling the Server Clock under NetWare 4.x

NetWare® 4.0 introduced a mechanism for controlling the apparent tick rate of the clock to simplify time synchronization.

Time synchronization is complicated by daylight savings time and time zones. Very little detail about these subjects is presented here because of the

large number of special cases.

You can discover useful information about the time environment on a server and control the tick rate of the clock by using the interface to the synchronized clock structure of a NetWare 4.x server. It is possible to duplicate or augment the function of the TIMESYNC NLM through this interface.

- The Synchronized Clock Structure
- Operation of the Clock
- The Synchronized Clock Interface
  - Synchronized Clock Status Flags
  - Clock Control Fields
  - The eventOffset and eventTime Fields
  - Daylight Savings Time Information Fields
- Interactions between Local Time, UTC, and Other Variables
- The Effect of Setting the Hardware Clock Bit
- More Detail for High Accuracy Time Synchronization Users
- Interactions with TIMESYNC
- SFT III Considerations for Time Synchronization

Modular CLIB Header Files

- Former and Present Header Names
- New NetWare SDK Headers
- Obsolete CLIB Headers

Remote Server Support

- Introduction to Remote Server Support
- Accessing Remote Servers
- Changing the Current Server
- Logging Out from Remote Servers
- Remote and Local Server Operations

Renamed NLM Functions

Some functions have either been removed from the NLM™ Library, or have had their names changed. See the following for more information.

Renamed Functions in Connection Number and Task Management Services

Renamed Functions in Connection Services

Renamed Functions in Screen Handling Services

Changes to SMP Services

NLM Limited Support Functions

VAP to NLM Conversion

Altering the Source Code

VAP Processes and NLM Threads

VAP to NLM Conversion and Linking

VAP to NLM Conversion and Loading

NLM Equivalents for VAP Functions

NLM Development: Tasks

Designing Client-Server NLM Applications

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

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. Other I/O operations should reside on the server because the data or resource resides on the server.

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

   Scalability is an important factor in designing client-server applications. For example, if you overload the server with CPU-intensive operations, your application might work well for 10 users, but not for 1,000.

   In addition, try to achieve load balancing in the application. Processing loads should be distributed somewhat evenly between the client and server portions of the application, so that neither portion 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.**

   As you decide, consider the performance characteristics of the interfaces and protocols. A good rule of thumb is to choose the lowest-level interface possible without significantly compromising the level of effort required to implement it.

   You should also consider portability issues. For example, using TLI eases supporting multiple client platforms, whereas IPX™ and SPX™ limit those choices.

3. **If you want to limit the number of clients your NLM™ will serve, determine the maximum number of clients to accept.**

   Consider the following factors:

Memory considerations---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 considerations---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.

Parent Topic: Introduction to Client-Server Applications

Development Steps

In general, follow these steps when developing an NLM™ application:

1. **Set up the development environment you will use to build, run and debug the NLM.**

   Three methods for setting up a development environment are discussed in NLM Client Development Setup.

2. **Plan the threads of execution your NLM will run.**

   All NLM applications 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 NLM Coding Issues.

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 NLM applications in the nonpreemptive environment (see Nonpreemptive Environment).**

   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 Tools Overview.**

Parent Topic: NLM Code Development

NLM Development: Concepts

16-bit Variables

To increase the execution speed of your NLM™ application, you should avoid 16-bit variables and use 32-bit variables instead. Accessing a 16-bit value takes more time than accessing a 32-bit value. For example,

```
MOV AX, WordVar ;16-bit value
```

on a 486 takes two clock ticks, but

```
MOV EAX, LongVar ;32-bit value
```

only takes one.

So, using the structure in Data Alignment, a better choice would be to change C to be a LONG as follows:

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

This structure is designed for optimal speed.

16-bit Parameters: If you are using 16-bit values to save space, remember that they are converted to 32-bit values when they are passed as parameters.

32-bit Registers

Because you are adapting code from a 16-bit environment (NetWare® 2.x) to a 32-bit environment (NetWare 3.x and 4.x), avoid loading pointers into 16-bit pointers. In the 80386 environment, the eight general-purpose, 32-bit registers are named EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. You should rewrite your program to use these registers.

For example, the following VAP assembly code uses the AX register in a 16-bit environment:

```
mov ax,offset mydata
```

To convert this code to an NLM, change it to use the EAX register available in the 32-bit environment, as in the following example:

```
mov eax,offset mydata
```

The following VAP assembly code uses two register moves to identify the beginning address of the data segment, DGROUP:

```
mov ax,DGROUP
mov ds,ax
mov dx,offset mydata
mov byte ptr [dx],0
```

The process of loading the segment register (shown in the VAP code) is unnecessary in an NLM:

```
mov mydata, 0
```

Parent Topic:

Memory Considerations When Converting VAPs

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™ application 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 which 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 **open**, **close**, **chdir**) 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.

Parent Topic:

Remote Server Support

Altering the Source Code

Your C language VAP can run as an NLM™ application with only minor

adjustments. Although some alterations are optional, they are suggested because they allow you to take full advantage of the NetWare® 3.x and 4.x architectures.

Assembly Language Considerations for VAP Conversion

Transferring Functions from VAPs to NLM Applications

Memory Considerations When Converting VAPs

Communication and VAP Conversion

Parent Topic:

VAP to NLM Conversion

API Support for NLM Applications Running in the IOEngine

When the MSEngine is loaded, a NetWare® SFT III™ server (in conjunction with the NetWare API) emulates a standard NetWare environment. This emulation involves redirecting file system (and related) functions targeted to what would be (in standard NetWare) the local server from the local server to the MSEngine (which is a remote server).

The redirection of file system-related functions made by an NLM running in the IOEngine is accomplished by the NetWare API and is transparent to the NLM. To achieve this type of transparent support, additional SFT III support code was added to the NetWare API.

Parent Topic:

The NetWare API and the IOEngine

Assembly Code That Calls C Functions

When assembly code calls a C function, it expects the C function to do the following:

Preserve the EBX, ESI, EDI and EBP registers

Place the return code in the EAX register

Your assembly code does not need to save the EBX, ESI, EDI and EIP registers before calling a C function because C functions automatically restore these registers when they return. It is a waste of instructions to save these registers before calling a C function. (This is not the case if you are using register-based parameter passing.)

C places the return code of a function in the EAX register. If a C function

returns a double, the return value is in the EAX and EDX registers, with the high bytes in EDX.

The C function does not restore the values of the EAX, ECX, or EDX registers if it changes them, so if your assembly code uses these registers, it should save them before calling the C function.

Parent Topic:

NLM Assembly Interface on the Intel Platform

Assembly Language Considerations for VAP Conversion

Most VAP assembly source code, written for the 80286 processor, requires substantial rewriting for the 80386 processor. If any of the following occur in your assembly language VAP, you must rewrite the program using 32-bit assembly language:

Segment register change

Calls to far pointers

Use of 16-bit registers as index registers

Other use of far memory or 16-bit registers

AtUnload and atexit Functions

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.

> **CAUTION: The AtUnload and atexit functions register routines that will be run by OS threads that only have NLM level context. These threads cannot be given thread group context because all thread groups have been destroyed by the time these functions are run. You cannot use any of the NetWare API functions that need context (for example, printf) while in the AtUnload and atexit routines. If you do, the server abends. If you need thread group level context to do cleanup, use a SIGTERM routine (see signal and Using signal handlers: Example).**

NLM applications 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.

NLM applications 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.

Autoloading Prerequisite NLM Applications

You can specify more than one prerequisite NLM™ applicaton by including the MODULE directive in a linker directive file. You can separate the NLM names with commas or use multiple MODULE directives on separate lines.

If different NLM applications 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 NLM applications that need to be autoloaded are specified with the linker directive MODULE (for WLINK and NLMLINK) as follows:

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

module_name1 and *module_name2* are the filenames of the NLM applications to be autoloaded.

NOTE: The NetWare® 3.0 OS does not support the automatic loading of modules specified in the MODULE directive. You have to load them

manually.

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.

C Code That Calls Assembly Procedures

When C calls an assembly procedure, it expects the procedure to behave as a C function behaves. Therefore, C expects the assembly procedure to do the following:

Preserve the EBX, ESI, EDI, and EBP registers.

Place the return value in the EAX parameter.

The first step the assembly routine must take is to save the EBX, ESI, EDI, and EBP registers if it is going to modify them. Then, before the assembly routine returns, it must restore the saved registers to their original states. Saving and restoring is usually accomplished by using the PUSH and POP instructions.

C expects to find the return code of a function in the EAX register. Therefore, an assembly procedure that is called by C needs to place its return code in the EAX register. If the procedure returns a double, it must place the return code in the EDX and EAX registers, with the high bytes in EDX.

The assembly routine does not need to save the state of the EAX, ECX, or EDX registers because C allows these registers to be changed.

It is important to conform to this behavior when writing assembly procedures to be registered as callbacks with the NetWare OS because this is the behavior that the OS expects.

Parent Topic:

NLM Assembly Interface on the Intel Platform

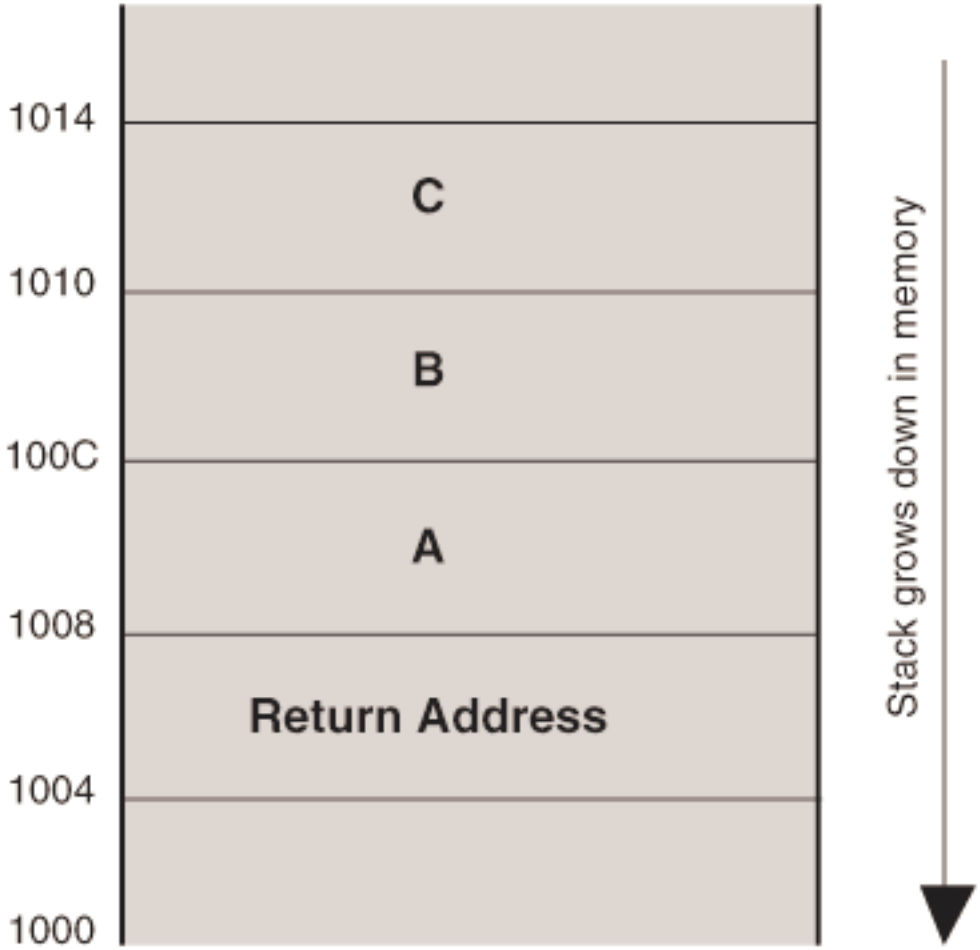
C Parameter Ordering

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 1. Example of C Parameter Passing



When C calls an assembly procedure, it pushes the parameters on the stack just as if the assembly procedure is a C function. For example, C issues the call

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

it places the parameters on the stack in the order of C, B, and A.

Related Topics::

Parameter Processing by Assembly Procedures

Passing Parameters to C Code

Parent Topic:

Parameter Passing

Changes to SMP Services

Changes to SMP Services were made in December, 1995 (Edition 2.3 of the NetWare SDK).

The following functions were removed because local semaphore functions in Thread are SMP enabled:

sema_alloc
sema_destroy
sema_examine
sema_init
sema_name
sema_post
sema_trywait
sema_wait

The following initialization functions were removed because they are not needed. Initialization is accomplished by allocation functions.

barrier_init
cond_init
mutex_init
rmutex_init
rwlock_init
spin_init

The following barrier spin-waiting functions have been removed:

_barrier_spin
_barrier_spin_destroy
_barrier_spin_init

The remaining functions have been renamed as follows:

Old Function Name	New Function Name
barrier_alloc	NWSMPBarrierAlloc
barrier_destroy	NWSMPBarrierDestroy
barrier_wait	NWSMPBarrierWait
cond_alloc	NWSMPCondAlloc
cond_broadcast	NWSMPCondBroadcast
cond_destroy	NWSMPCondDestroy
cond_signal	NWSMPCondSignal
cond_wait	NWSMPCondWait
mutex_destroy	NWSMPMutexDestroy

mutex_lock	**NWSMPMutexLock**
mutex_sleep_alloc	**NWSMPMutexSleepAlloc**
mutex_trylock	**NWSMPMutexTryLock**
mutex_unlock	**NWSMPMutexUnlock**
rmutex_alloc	**NWSMPRMutexAlloc**
rmutex_destroy	**NWSMPRMutexDestroy**
rmutex_lock	**NWSMPRMutexLock**
rmutex_owner	**NWSMPRMutexOwner**
rmutex_trylock	**NWSMPRMutexTryLock**
rmutex_unlock	**NWSMPRMutexUnlock**
rwlock_alloc	**NWSMPRWLockAlloc**
rwlock_destroy	**NWSMPRWLockDestroy**
rw_rdlock	**NWSMPRWReadLock**
rw_tryrdlock	**NWSMPRWTryReadLock**
rw_trywrlock	**NWSMPRWTryWriteLock**
rw_unlock	**NWSMPRWUnlock**
rw_wrlock	**NWSMPRWWriteLock**
spin_alloc	**NWSMPSpinAlloc**
spin_destroy	**NWSMPSpinDestroy**
spin_lock	**NWSMPSpinLock**
spin_trylock	**NWSMPSpinTryLock**
spin_unlock	**NWSMPSpinUnlock**
thr_yield_to_mp	**NWSMPThreadToMP**
thr_yield_to_NetWare	**NWSMPThreadToNetWare**

Changing the Current Server

The current server can be changed by calling **SetCurrentFileServerID** or **chdir**. The **chdir** function 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.

CHECK Function

In the first step of the unload process, the NLM™ application calls the CHECK function, but only if it has been specified. NLM applications specify

CHECK functions in the linking phase using directives. For example, the WLINK linker uses the "OPTION CHECK" directive to specify the name of the CHECK function. (The function does not need to be named CHECK.) 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
```

Your CHECK function must be defined in one of the object modules. The CHECK function should determine if the NLM is in a state in which it can unload safely. If so, the function should return a zero value, indicating that the unload process can continue normally. However, 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® OS 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, because the operator has the option of continuing with the execution of the NLM, rather than terminating it.

> **CAUTION: The CHECK function is run by an OS 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.**

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
```

:

NOTE: This method does not prevent an operator from continuing with the unload process. Instead, it 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** 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;
}
```

Starting with the NetWare 4.0 OS, 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.

Related Topics::

Termination Process

Using check functions: Example

Using check functions: Example 2

CLIBAUX.NLM

CLIBAUX.NLM provides support for symbols that have changed for various reasons, allowing NLM applications built with new headers to run with older versions of CLIB.NLM.

For example, to support new ANSI code for **stat**, the structure stat had to be enlarged. To provide support for NLM applications compiled with old headers, **stat** was retained and a new function, **stat_411**, provides the new functionality (**stat** is now redefined to **stat_411** by the preprocessor). NLM applications that use older headers continue to work with the newer CLIB and NLM applications that use new headers have increased functionality. However, if an NLM that uses new headers is loaded with an older version of CLIB.NLM, **stat_411** is missing.

CLIBAUX.NLM provides missing symbols so that the NLM can load. However, CLIBAUX.NLM usually is not able to fill in the new information in the structure, filling it in with NULLs instead. If the field information changed offsets in the structure, CLIBAUX.NLM copies the structure into the expected offsets based on the new header file.

The following figure shows what the behavior of an NLM is expected to be when loaded with older versions of CLIB.

NLM built using:

	Old Libraries v. 3.12, 4.10 to 4.11i	**New Libraries** v. 4.11j and Moab
Old SDK Headers up to Sept. 1996	loads and works	loads and works
SDK Headers Sept. 1996 to Sept. 1997	loads, but might require CLIBAUX.NLM to work	loads and works
New SDK Headers starting Sept. 1997	requires CLIBAUX.NLM to load and work	loads and works

IMPORTANT: In this graphic, "works" means that the NLM runs

without recompiling or loading CLIBAUX.NLM. The NLM might abend or produce inaccurate results for some functions using older versions of CLIB.

Functions affected by these changes include at least the following:

fstat
NWLsetlocale
opendir
readdir
ScanErasedFiles
setlocale
stat

NOTE: See information about the use of Watcom headers and statically linked libraries below.

Load Dependence

We do not recommend load-dependence on CLIBAUX.NLM, although the alternative of making the system administrator aware of this potential problem might be unsatisfactory. You can attempt to load CLIBAUX.NLM along with later libraries that provide the new symbols because CLIBAUX.NLM autoloads CLIB.NLM and unloads itself if none of the symbols that it exports are missing. The autoload, autounload process can be repeated as often as necessary for NLM applications that depend on CLIBAUX.NLM.

Compiling with Watcom Headers

Watcom provides many functions also provided by CLIB.NLM including almost all ANSI functions. Because Watcom's library is statically linked, Watcom headers can differ greatly from Novell headers with no problem because the header definitions are linked into the NLM.

However, occasionally changes cause Watcom and Novell to become incompatible. For example, the Novell and Watcom versions of locale.h were identical before Watcom v. 9.0, when Watcom changed some definitions, including that for LC_ALL. This caused no problem when calling **setlocale** because it is provided by Watcom, but calling **NWLsetlocale** caused problems with the underlying collation tables and other locale settings.

Rather than requiring the use of Novell headers, we have changed locale.h to match Watcom and supply **setlocale_411** and **NWLsetlocale_411** in newer libraries. These functions are now called when **setlocale** or **NWLsetlocale** are called. CLIBAUX.NLM provides these symbols for newer NLM applications running with older versions of CLIB.

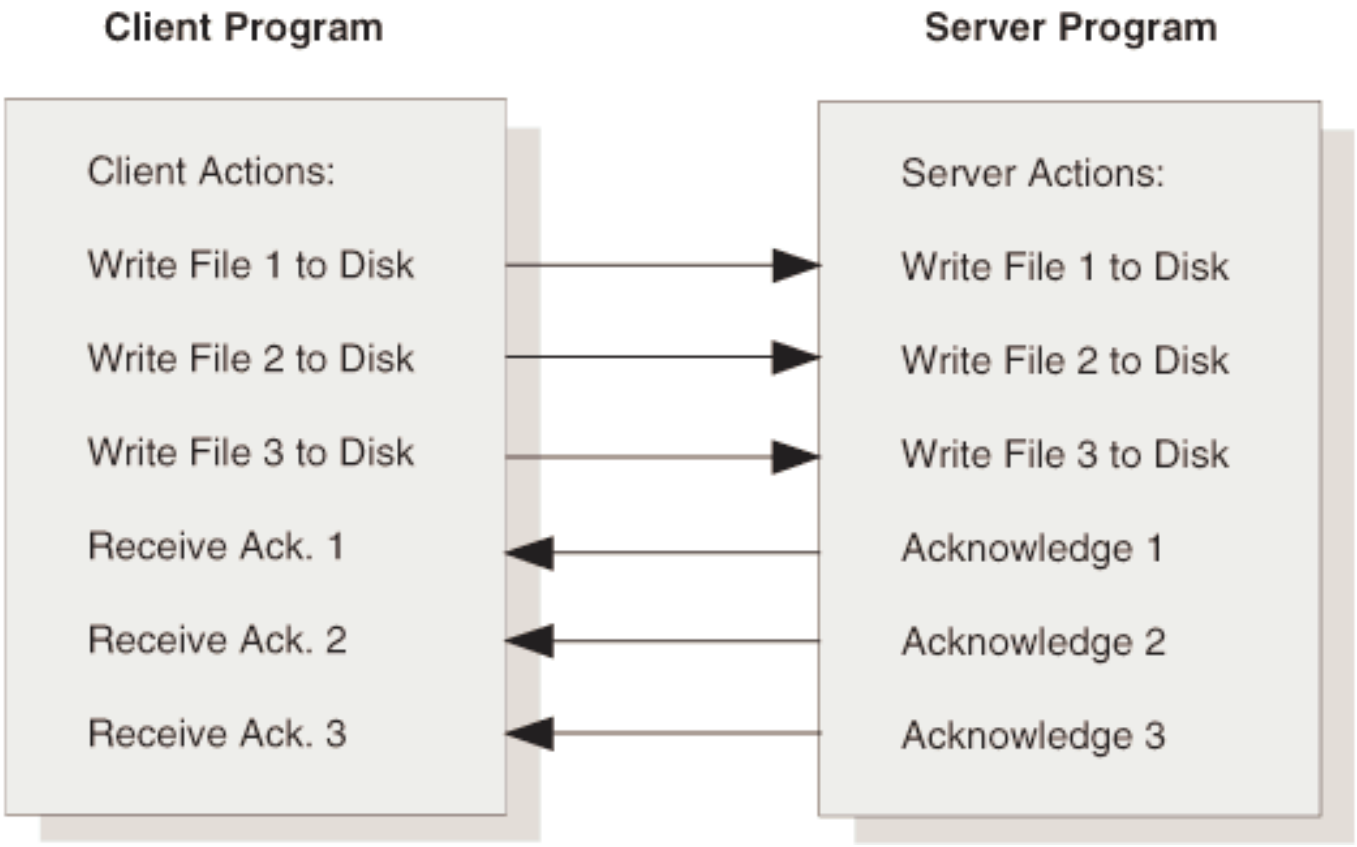
Client-Server Communication

One factor in determining the type of client-server communication you use involves how your client and server applications will manage service requests. There are two types of client-server communication:

Asynchronous communication

The client continues processing after it issues service requests, but before the server replies, as shown in the following figure.

Figure 2. Asynchronous Communication

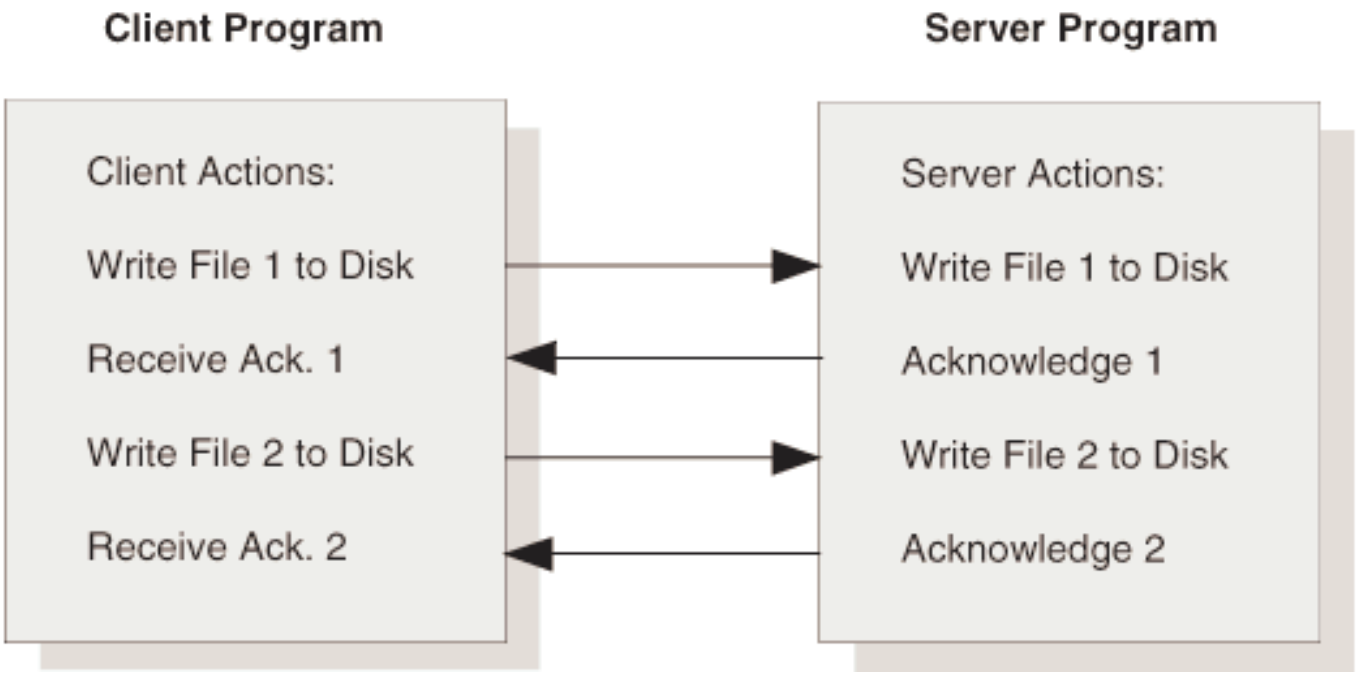


In this sample archiving application, the client sends multiple requests to the server without waiting between requests for a reply.

Synchronous communication

The calling program makes a service request and then suspends processing until the called process replies, as shown in the following figure.

Figure 3. Synchronous Communication



In this sample database application, the client sends a read file request to the server and then waits for a reply before sending the next request.

Asynchronous communication can be achieved through any of the following: IPX, SPX, SPX II, TLI, BSD Sockets, or Queue Management (see QMS for Client-Server Communication).

Synchronous communication can be achieved through any of the following: IPX, SPX, SPX II, TLI, or BSD Sockets API.

Related Topics:

QMS for Client-Server Communication

Clock Control Fields

The *clock* field can be set if necessary. See Interactions between Local Time, UTC, and Other Variables for an explanation of the relationship between local time and UTC. When the clock is set, a time change event occurs and can be detected through the event notification interface.

The *tickIncrement*, *adjustmentCount*, *adjustmentValue*, and *grossCorrection* fields can be changed as needed to control the clock, as explained inThe

Synchronized Clock Interface. If necessary, you can change the *tickIncrement* .

The *stdTickIncrement* field contains the value that is used to initialize the *tickIncrement*. This field should not be changed. It is provided as a reference value, in case *tickIncrement* must be restored.

Communicating with Other NLM Applications

A loaded NLM™ application 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 NLM Linkers.

Communication and VAP Conversion

NLM™ applications have available to them a wide variety of functions for accomplishing their tasks. NLM applications can access the many functions available in the NetWare® API. In addition, NLM applications can import functions from other NLM applications.

Once an NLM is loaded, it becomes part of the NetWare 3.x or 4.x OS. This means NLM applications have advantages that VAPs do not. Specifically, NLM applications can directly access both the server and other NLM applications. Thus, NLM applications handle communication more efficiently than VAPs.

Related Topics:

Server Connections and VAP Conversion

Communicating with Other NLM Applications

Connection and Task Numbers

Connection numbers are unique numbers assigned by a NetWare® server to identify its clients. They reflect the client's place in the server's connection table. Connection numbers provide an easy way to identify objects logged in on the network and to obtain additional information about them. When a client closes its connection, this connection goes back into the pool of available connections.

NetWare file services use a combination of connection and task numbers to identify individual programs, running at any given time at a workstation. Task numbers are only unique for a given workstation.

Connection numbers and task numbers together are used for:

Client Identification

When a client attaches to a server, each file service request has an identifying connection number and task number.

Security

When a file service is requested, the OS verifies that the client and/or NLM has the appropriate access rights by looking at the connection number.

Accounting

In a system where the NetWare accounting feature is installed, the price of servicing a request is charged to the specified connection number.

Resource Management

When a connection logs out, server resources allocated to that connection are freed.

A DOS application on a network workstation does not normally concern itself with connection numbers and task numbers; the network shell program handles those issues.

Because server-based applications often offer services on behalf of more than one client, they must be able to specify on whose behalf they are operating.

An NLM has three different types of connection numbers it can specify:

Connection number (0)

This gives the NLM supervisor rights to the server's file system. This is the default connection number for NLM applications.

The connection number of an already logged-in workstation

This gives the NLM the file access rights of the logged-in workstation. This lets the NLM do work on behalf of the workstation.

An NLM connection number

The NetWare 3.11 OS reserves 100 "hidden" connection numbers for NLM applications. The numbers begin after the last client connection number. For instance, in a 250-user system, NLM connection numbers run from 251 through 350. Likewise, in a 100-user system, the NLM connection numbers are 101 through 200.

The NetWare 4.x OS provides unlimited connection numbers for NLM applications. These connection numbers start at the first number after the last client connection number.

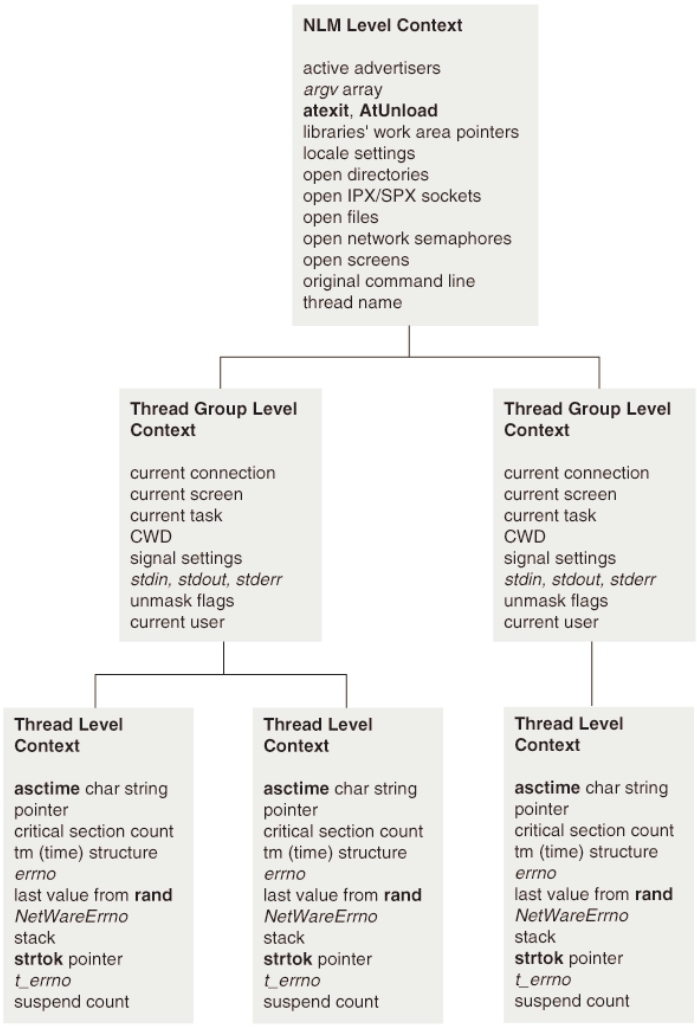
NLM applications can get an NLM connection number on the server they reside on by attaching to the server or by calling "**SetCurrentConnection**(-1)". They can then log in as a user and assume the rights of that user.

For more information about connection and task numbers, see Connection Number and Task Management.

Context

The NetWare® API maintains a context for each NLM™ application that is running. The context is divided into three levels of scope: thread level context, thread group level context, and NLM level context. (The following figure illustrates the three levels of context.) Because this context is created using the functions found in CLIB.NLM, it is commonly known as CLIB context.

Figure 4. Context Levels



> **NOTE:** An understanding of context is critical to NLM development. Many errors in NLM applications are caused by developers not understanding context and how it can change.

Threads created in one of the four ways described in Thread Groups have the CLIB thread level, group level, and NLM level context. These context levels contain different information that is changed by the NetWare API. The context information cannot be changed directly by the programmer.

> **NOTE:** A fifth way to create threads is for the OS to create threads. These threads do not have CLIB context, and must be given CLIB context. This issue is discussed after the following discussion about the context levels.

Related Topics:

Thread Level Context

Thread Group Level Context

NLM Level Context

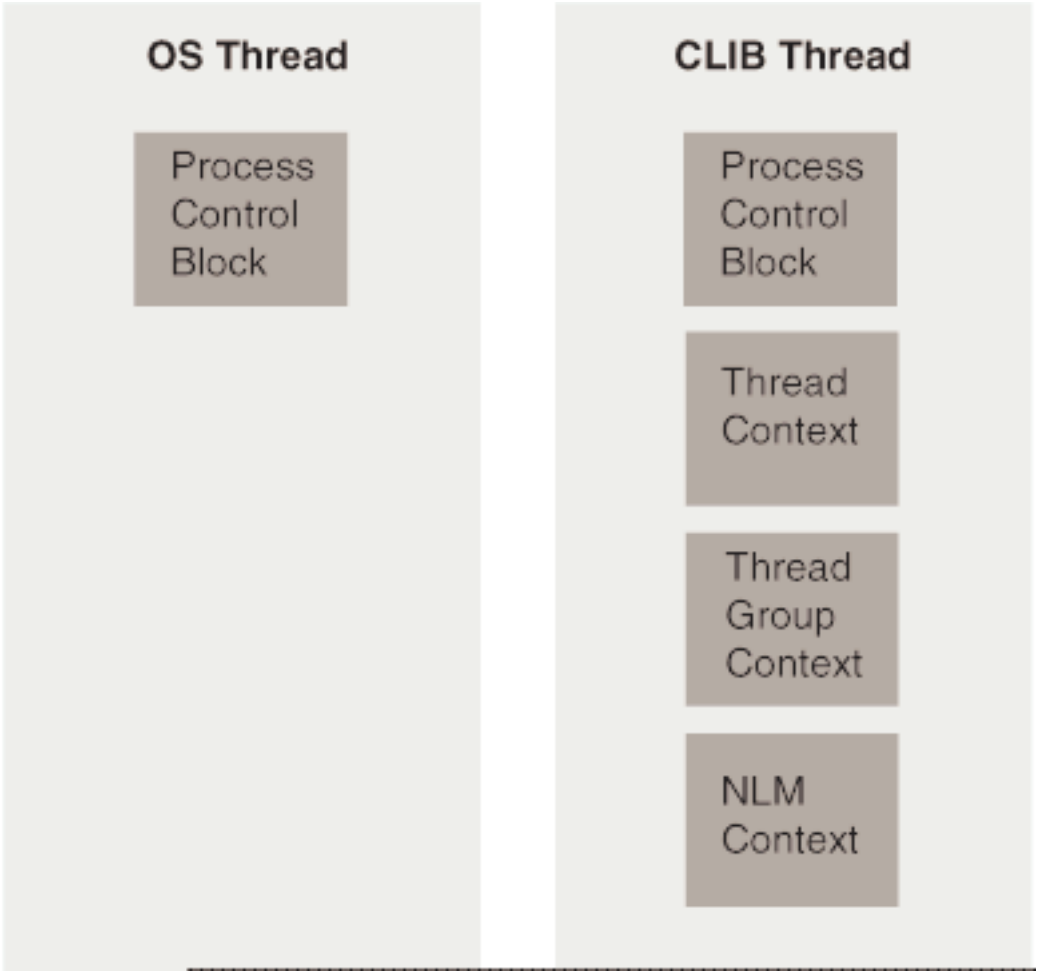
Context Problems with OS Threads

Context Solutions for OS Threads

Context Problems with OS Threads

There are two types of threads running in the NetWare® OS: OS threads (also known as callbacks) and CLIB threads (those created by the CLIB.NLM functions). The OS threads are created by the OS in instances such as when the **LOAD** and the **UNLOAD** commands are used. CLIB threads are created by calling the NetWare API functions **BeginThread**, **ScheduleWorkToDo**, and **BeginThreadGroup**, and by a default thread starting at the **main** function. The following figure shows that OS threads are missing the context that CLIB threads have.

Figure 5. CLIB and OS Threads



The problem here is that many---but not all---NetWare API functions need to have a context in order to work correctly. For example:**printf** writes to the calling thread's current screen. The current screen is kept in the thread's thread group context. OS threads do not have any CLIB context, so their calls to **printf** do not produce output anywhere. In more extreme cases, OS threads calling the NetWare API functions that need CLIB context can cause the server to abend.

> **NOTE:** The solution to this problem is to give the OS threads context, thereby turning them into CLIB threads. The method for doing this is presented in the following section.

Developers must be aware of all the situations where NLM™ applications will be running with OS threads instead of CLIB threads, and adjust their code accordingly to give the OS threads CLIB context. The following is a list of conditions where the NLM runs with OS threads:

In the optional startup function that is specified with the "OPTION START" directive, when using the WLINK linker, or with the "START" directive for NLMLINK

In the check function that is specified with the "OPTION CHECK" directive when using the WLINK linker, or with the "CHECK" directive for NLMLINK

In some of the functions registered with **signal**, such as SIGTERM

In functions registered with **RegisterForEvent**

In functions registered with **ScheduleSleepAESProcessEvent**

In functions registered with **ScheduleNoSleepAESProcessEvent**

In the function registered with **RegisterConsoleCommand**

Adding Console Commands: Example

In the function registered with **FERegisterNSPathParser** (might not have the correct context)

In the library cleanup function set by **RegisterLibrary**

Context Solutions for OS Threads

Two solutions to these context problems are as follows:

NetWare 3.11 and 4.x solution:read group ID of one of the groups, such as for the default thread group created for the **main** function. (You might also want to create a global variable for each of the thread groups that are created.) The thread group ID of the current thread group can be obtained with **GetThreadGroupID**, as shown in the following example:

```
#include <process.h>
   int globalThreadGroupID;

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

Then, when you have an OS thread running, you give the OS thread context using **SetThreadGroupID** as follows:

using **SetThreadGroupID** as follows:

```
oldTGID = SetThreadGroupID(globalThreadGroupID);
   /* do work */
   SetThreadGroupID(oldTGID);
   /* always set back the thread group ID */
```

At this point, the NetWare® API takes the OS thread and gives it context, just as if it had been a CLIB thread. This lets you use the NetWare API functions that need context.

> **CAUTION: You must be careful when using the thread group ID that other threads are using. Changes to the context affect all threads in that group.**

NetWare 4.x Specific solution:CLIB threads in the NetWare 4.x OS have been given a context specifier that gives these threads the ability to automatically give context to callbacks (OS threads that are registered to be called to run when specific conditions occur) that they register. The context that is given to the callbacks when they are registered is determined by the setting of the registering thread's context specifier. The context specifier can be set to one of the following settings:

NO_CONTEXT

Use this when you don't want callbacks to be automatically registered with a 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.

Call **SetThreadGroupID** and pass in a valid thread group ID. Use this once inside your callback to give your callback thread CLIB context.

USE_CURRENT_CONTEXT

Use this to register callbacks to have the thread group context of the registering thread.

A valid thread group ID

Use this when you want the callbacks to have a different thread group context than the thread that schedules them.

You can determine the existing setting of the registering thread's context specifier by calling **GetThreadContextSpecifier**. Call **SetThreadContextSpecifier** to set a thread's context specifier.

When a new thread is started with **BeginThread**, **BeginThreadGroup**, or **ScheduleWorkToDo**, its context specifier is set to USE_CURRENT_CONTEXT by default.

Using this solution, if you want the registered thread to have the thread group context of the registering thread, you would set the registering thread's context specifier to USE_CURRENT_CONTEXT (if it has been changed from the default) and then register the function that will run as a

callback.

> **NOTE:** The drawback to using this solution is that the context specifier is specific to the NetWare 4.x OS. If you use this solution, your NLM will not run on the NetWare 3.11 OS.

Critical Sections

Critical sections allow you to put to sleep all threads in the NLM™ application, except for the current thread, by calling **EnterCritSec**. This allows the current thread to execute code without relinquishing control to other threads in the NLM. The thread must exit the critical section, using **ExitCritSec**, before any other thread in the NLM can run.

> **NOTE:** Critical sections only affect the NLM they are called from. Calling **EnterCritSec** does not put to sleep threads in other NLM applications.

An alternative to using critical sections is to use local or network semaphores, which limit the access of threads to specific resources and put threads in semaphore queues. Semaphores are discussed in NLM Synchronization.

For more information on critical sections, see Thread.

Cross-Platform Functions in NLM Applications

All cross-platform ("client") functions have now been ported to run on the NLM™ platform through the library CALNLM32.NLM, also included in this SDK. These functions provide a much richer API set for NLM applications than has previously been offered. (NLM specific functions that provide the same functionality as the cross-platform functions are called Limited Support NLM functions. These functions are contained in the library NIT.NLM and documented in the *NLM Function Reference* of the NetWare® Limited Support SDK. See NLM Limited Support Functions.)

> **WARNING: The connection model for CALNLM32.NLM is significantly different from the connection model for NIT.NLM. An application that makes calls to both libraries will break. Thus link to either CALNLM32.NLM or NIT.NLM to prevent calling functions from both.**

When considering the use of cross-platform functions for NLM development, observe the following about load order and include order:

Load Order

If your NLM application 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 application does not include calls to the cross-platform libraries, load CLIB.NLM first. Its associated modules will also load automatically.

Include Order

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.

If an NLM application 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 application makes calls to the NIT.NLM library, specify the following include order in the make file or in a **SET** command:

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

Parent Topic:

NetWare API

Current Working Directory

One difference between programming for the NetWare® OS and DOS is that the threads in NLM™ applications can share information called **context** . An example of context is the implementation of the current working directory (CWD). Under DOS, all programs on a given DOS drive share a single CWD. In the NLM environment, each thread group has its own 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 NLM applications 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. Similarly, if the pathname does not contain a server or volume, the path is considered relative to the CWD.

Custom VAP Functions

You might have defined some custom functions for your VAP that are not available through the NetWare® API---DOS Library, such as screen and I/O management functions. However, the NetWare API provides many of these functions for you. Therefore, you might want to replace many of your custom functions with NetWare API functions.

For more information on the NetWare API functions, see the function descriptions.

Data Alignment

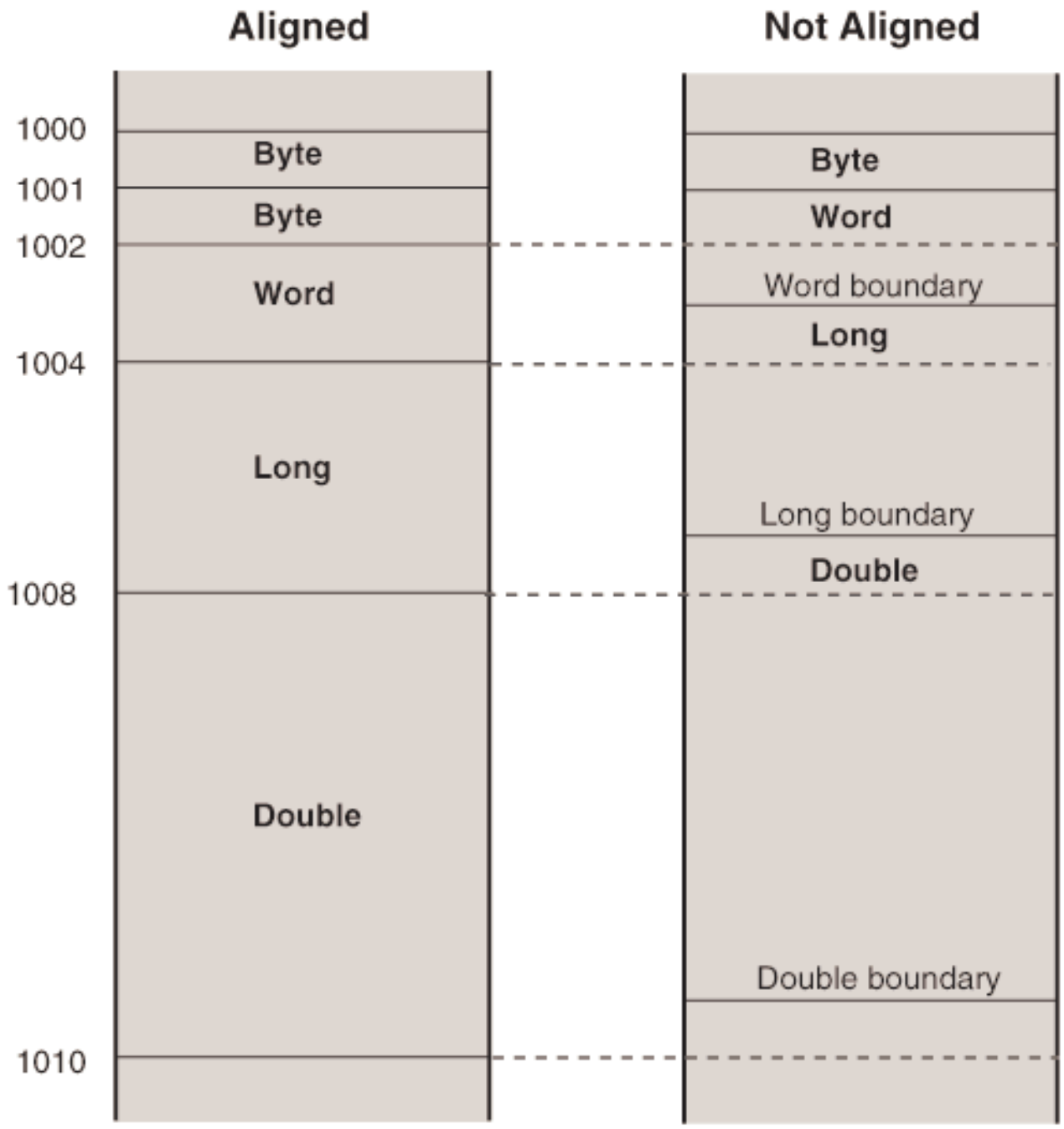
Data alignment describes data that is within the boundaries that are associated with each data type. Each data type, BYTE, WORD, LONG, and DOUBLE have rules where their starting addresses should begin. These rules are summarized in the following table.

Table auto. Data Type Address Boundaries

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



On 386 and 486 based processors, accessing variables that are not long-aligned adds extra clock cycles to your program's execution. You take a three clock cycle hit (for alignment) each time you access a variable that is not long-aligned.

For example, if *var* is a 32-bit variable, the instruction

```
MOV EAX, var
```

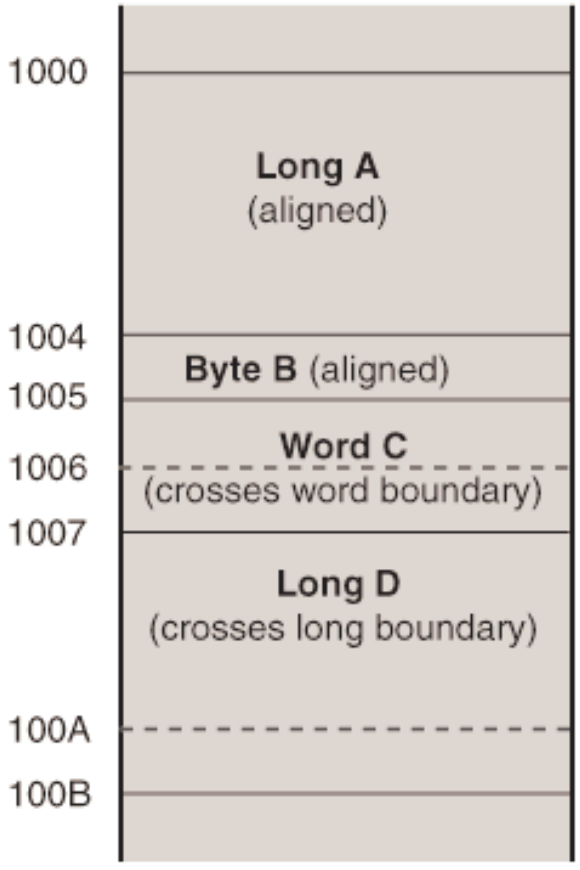
takes one clock cycle to complete if the value is long-aligned and four clock cycles if it is not (one cycle for the instruction, three cycles for alignment).

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 7. Alignment of BadStruct

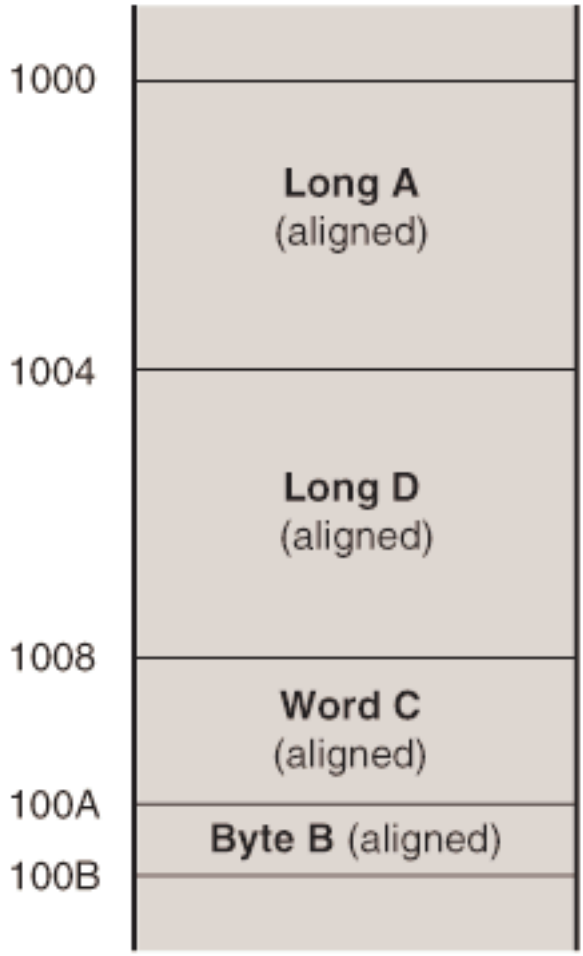


To solve the alignment problem (and to speed up the NLM™ application), you can arrange the structure fields as follows:

```
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 8. Alignment of GoodStruct



Daylight Savings Time Information Fields

The OS interface for setting and controlling Daylight Savings Time (DST) information is implemented in **SET** parameters. The values stored in the Synchronized Clock Structure should not be changed except by using the **SET** parameter interface. Otherwise, inconsistent information might be displayed by the set parameters.

The *daylight* field indicates whether the value of *tzname*[1] is valid. It is true (nonzero) if the second abbreviation was detected in the time zone string.

The *daylightOffset* field contains the number of seconds by which local time is adjusted when DST is in effect. It is controlled by the **SET DAYLIGHT SAVINGS TIME OFFSET** parameter.

When true (nonzero), the *daylightOnOff* field indicates that DST is in effect. The state is determined by the OS after analyzing the information provided by the **SET START/END OF DAYLIGHT SAVINGS TIME**parameters.

The *startDSTime* and *stopDSTime* fields hold the times at which the next DST transitions are scheduled to occur. The times are determined by the information provided in the **SET START/END OF DAYLIGHT SAVINGS TIME** parameters.

Drive Duplexing

Drive duplexing, a major element of SFT™ Level II, is important to the operation of SFT III™. To the MSEngine, NetWare® partitions residing on two mirrored machines appear as if they are duplexed partitions residing on the same standard NetWare server. This makes it possible for SFT III developers to ignore the details of file system fault tolerance just as standard NetWare developers are able to do.

One benefit offered by the implementation of drive duplexing of SFT III is that the secondary server is able to fulfill read requests as if it were a standard NetWare duplexed drive controller. This "split seek" capability can speed file service when two servers are mirrored.

Another benefit offered by this implementation of drive duplexing is that remirroring of NetWare partitions after either the primary or secondary server suffers a fault occurs in the background after the two servers are synchronized. This makes it possible to restore redundant operation very quickly after a fault.

Dual-Processing Support

NetWare® SFT III™ provides optional support for dual-CPU server hardware. When running on such hardware, NetWare SFT III performs **asymmetric** multiprocessing, meaning that the load placed on each respective CPU is not equal. When NetWare SFT III is running on a dual-CPU machine, one processor performs all hardware-related I/O, whereas the other processor performs file service,NCP™ service, and other processing not directly related to the server hardware.

Flat Memory Model

VAPs use many memory models (such as small, large, and huge). In contrast, NLM™ applications use only the flat memory model.

In the flat memory model, all calls to functions and references to data are made with 32-bit near pointers. Using 32-bit offsets, the flat memory model provides full memory addressability of up to 4 GB of memory, referenced to a single selector for data and a single selector for code. (A **selector** is a value loaded into a segment register.)

Segment registers CS, DS, ES, FS, GS, and SS are initialized to constants at load time. Because you do not need to alter the segment registers to access code or data, all references to far objects should be modified to reference near objects.

One quick way to remove the far calls from your C language code is to insert the following line:

```
#define far
```

This statement defines far as nothing, and so the program ignores subsequent far references.

The following VAP C language code references a far object:

```
extern int far myfunc(void);
```

To convert this code to an NLM, change it to reference the function as a near object, as in the following example:

```
extern int myfunc(void);
```

In addition, you should remove all far pointers from the source code. For example, the following VAP assembly language code uses a far pointer:

```
call far myfunction
```

To convert the code to an NLM, change it to use a near pointer or no pointer, as in the following example:

```
call near myfunction
```

```
/* or */
call myfunction
```

The following example of VAP code uses far pointers:

```
char obj[10];
char far *mydata;
mydata = (char far *)obj;
```

To convert it to an NLM eliminate the far pointers, as in the following example:

```
char obj[10];
char *mydata;
mydata = obj;
```

Following Exit Steps

The registered function calls and signal raising shown in the figures that show exit steps occur only if the NLM™ application has specified them, using **AtUnload**, **atexit**, and **signal**. If no registered functions or signal handling exist in the NLM, the associated step is skipped.

In addition, steps are performed depending on the way the NLM terminates. For example, the NetWare API performs some resource cleanup at every NLM termination, but the **CHECK** function and the **AtUnload** function are called only if the NLM is unloaded at the command line. The **atexit** functions are only called in the self-termination and unload processes; they are not called during the abnormal exit process.

Former and Present Header Names

The following table lays out, by file name, the correspondence of header files between the unified, "monolithic" CLIB that shipped with the NLM™ SDK to the modular CLIB. It also identifies which CLIB module contains the functions declared in each migrated header file. Files marked with an asterisk (*) in the first column did not change names in the transition to modular CLIB. Note that some files were divided between two CLIB modules and retained the original name in both modules.

"Monolithic" CLIB Header	Modular CLIB Module	Modular CLIB Header	Comment or Explanation
advanced.h	NLMLIB.NLM	nwadv.h	
assert.h*	CLIB.NLM	assert.h	ANSI-compliant
conio.h	THREADS.NLM	nwconio.h	

ctype.h*	CLIB.NLM	ctype.h	ANSI-compliant
datamig.h	NIT.NLM	nwdatamg.h	
debugapi.h	THREADS.NLM	nwdebug.h	
dfs.h	NLMLIB.NLM	nwdfs.h	
direct.h	NLMLIB.NLM	nwfattr.h	
dos.h	NLMLIB.NLM	nwdos.h	
dynarray.h	NLMLIB.NLM	nwdynarr.h	
errno.h*	CLIB.NLM	errno.h	ANSI-compliant
extattr.h	NIT.NLM	nwextatt.h	
fcntl.h*	NLMLIB.NLM	fcntl.h	
fileengd.h	NLMLIB.NLM	nwfile.h	
fileengd.h	NLMLIB.NLM	nwfattr.h	
float.h*	CLIB.NLM	float.h	ANSI-compliant
fshooks.h	NLMLIB.NLM	nwfshook.h	
io.h	NLMLIB.NLM	nwfattr.h	
library.h	NLMLIB.NLM	nwlib.h	
limits.h*	CLIB.NLM	limits.h	ANSI-compliant
locale.h*	CLIB.NLM	locale.h	ANSI-compliant
malloc.h	THREADS.NLM	nwmalloc.h	
math.h*	CLIB.NLM	math.h	ANSI-compliant
namespc.h	NLMLIB.NLM	nwfile.h	
ncpext.h	NLMLIB.NLM	nwncpext.h	
niterror.h	THREADS.NLM	nwerrno.h	
nwaccntg.h*	NIT.NLM	nwaccntg.h	
nwafp.h*	NIT.NLM	nwafp.h	
nwbindry.h*	NIT.NLM	nwbindry.h	
nwclient.h*	REQUESTR.NLM	nwclient.h	
nwcntask.h*	REQUESTR.NLM	nwcntask.h	
nwcntask.h*	NLMLIB.NLM	nwcntask.h	
nwconn.h*	REQUESTR.NLM	nwconn.h	
nwconn.h*	NLMLIB.NLM	nwconn.h	
nwenvrn.h*	NIT.NLM	nwenvrn.h	

nwenvrn1.h*	NIT.NLM	nwenvrn1.h	
nwfile.h*	NLMLIB.NLM	nwfile.h	
nwipxspx.h*	NLMLIB.NLM	nwipxspx.h	
nwlocale.h*	NLMLIB.NLM	nwlocale.h	
nwmedian.h*	NLMLIB.NLM	nwmedian.h	
nwmisc.h	NLMLIB.NLM	nwfile.h	
nwmisc.h	NIT.NLM	nwdir.h	
nwmisc.h	NLMLIB.NLM	nwtime.h	
nwmisc.h	THREADS.NLM	nwstring.h	
nwmsg.h*	NIT.NLM	nwmsg.h	
nwmsg.h	THREADS.NLM	nwpre.h	New header file
nwqueue.h*	NIT.NLM	nwqueue.h	
nwsemaph.h*	THREADS.NLM	nwsemaph.h	
nwserial.h*	NIT.NLM	nwserial.h	
nwservst.h*	NIT.NLM	nwservst.h	
nwsmp.h*	THREADS.NLM	nwsmp.h	New header file
nwsync.h*	NIT.NLM	nwsync.h	
nwtts.h*	NIT.NLM	nwtts.h	
nwtypes.h*	THREADS.NLM	nwtypes.h	
process.h	THREADS.NLM	nwthread.h	
sap.h	NLMLIB.NLM	nwsap.h	
setjmp.h*	CLIB.NLM	setjmp.h	ANSI-compliant
signal.h	THREADS.NLM	nwsignal.h	
signal.h*	CLIB.NLM	signal.h	ANSI-compliant
stdarg.h*	CLIB.NLM	stdarg.h	ANSI-compliant
stddef.h*	CLIB.NLM	stddef.h	ANSI-compliant
stdio.h*	CLIB.NLM	stdio.h	ANSI-compliant
stdlib.h*	CLIB.NLM	stdlib.h	ANSI-compliant
string.h*	CLIB.NLM	string.h	ANSI-compliant
sys/bsdskt.h*	NLMLIB.NLM	sys/bsdskt.h	
sys/filio.h*	NLMLIB.NLM	sys/fileio.h	
sys/ioctl.h*	NLMLIB.NLM	sys/ioctl.h	

sys/socket.h*	NLMLIB.NLM	sys/socket.h	
sys/sockio.h*	NLMLIB.NLM	sys/sockio.h	
sys/stat.h*	NLMLIB.NLM	sys/stat.h	POSIX-compliant
sys/time.h	NLMLIB.NLM	sys/timeval.h	
sys/types.h*	NLMLIB.NLM	sys/types.h	POSIX-compliant
sys/uio.h*	NLMLIB.NLM	sys/uio.h	
sys/utsname.h*	NLMLIB.NLM	sys/utsname.h	POSIX-compliant
time.h	NLMLIB.NLM	nwtime.h	
time.h*	CLIB.NLM	time.h	ANSI-compliant

Freeing Resources upon Exit

NLM™ applications are responsible for freeing the resources they allocate, such as memory, sockets, screens, devices, semaphores, and so on. NLM applications 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.

Functions Unsupported by the IOEngine

The three functions listed below are not supported in the IOEngine because they provide the calling NLM™ application with direct access to file system cache buffers.

AsyncRead
AsyncRelease
gwrite

For the same reason, you also cannot call the Direct File System functions with the IOEngine.

AFP functions always return an error indicating that AFP is not supported on the MSEngine. This is not an abnormal situation: because the MSEngine in NetWare SFT III™ 3.11 does not support AppleTalk*, these functions always return with an error code indicating that the Macintosh* name space is not present. However, AppleTalk will be supported in a future version of NetWare SFT III.

General Purpose Registers

The Intel* 386* and 486 processors have eight general purpose registers that are available for programmers to use. These registers are EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP.

When using a language to write functions or procedures to be used by another language, you need to understand how each language uses these registers. For example, C automatically restores the state of some of the registers when it returns from a function, but an assembly procedure does not restore the state unless you explicitly save the registers and restore them.

How NLM Applications Are Loaded

The executable file that first establishes the NetWare® 3.x and 4.x OS's 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 NLM™ applications. The loader knows how to load NLM applications, 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 NLM applications.

All NLM applciations 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. (For the NetWare 4.x OS, 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 NLM applications 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 NLM applications to call services directly by calling their code.

Related Topics::

Loading Multiple NLM Applications

Autoloading Prerequisite NLM Applications

Importing and Exporting NLM Applications

Importing and Exporting NLM Applications

After processing the autoload list, the loader processes the import list and then the export list. The loader checks the import list of theNLM™ application, 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.

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**.

Interactions between Local Time, UTC, and Other Variables

The synchronized clock structure holds UTC and information that determines how local time can be calculated from UTC. (For efficiency, local time is kept as a separate clock structure that is incremented along with UTC during each timer interrupt.) The relationship between local time and UTC is described by the equation:

Local = UTC - *timezoneOffset* + *daylightOffset* * *daylightOnOff*

assuming that *daylightOnOff* is always 0 or 1. This equation contains five variables, of which only UTC cannot be set from the console. To maintain the relationship between local time and UTC, one or the other must be recalculated when any variable changes. The general rule is that when UTC is set, the local time is recalculated, but when any other variable is set, UTC is recalculated.

When calling **SetSyncClockFields** to change the UTC clock, local time is automatically recalculated. The calculation of local time is performed after any field updates are applied. Because local time is updated at the same rate as UTC time, any adjustments made to the clock also affect the calculation of local time.

An interface already exists for setting the local time from the system console or under program control. That interface updates the UTC clock in the synchronized clock structure. However, because only the whole seconds can be specified when setting local time, the subsecond part of the UTC clock is set to 0 when local time is set.

Interactions with TIMESYNC

Although the synchronized clock interface was designed to support the needs of TIMESYNC NLM, the OS and TIMESYNC have been separated as much as possible. The OS knows very little about the functionality of TIMESYNC.

Because the TIMESYNC NLM can be unloaded, it is possible to write a different synchronization NLM. If you do so, you must be aware of the two places where the OS interacts with time synchronization:

The OS clears the CLOCK_IS_NETWORK_SYNCHRONIZED bit in the

status flags to indicate that critical time parameters might have changed and UTC time might need correction.

The status flags, which includes a server type field, is passed to clients, both in the server and on workstations. You should set the field to one of the known values (described in Synchronized Clock Status Flags) that best reflects how time is controlled on the server.

It might be useful to augment the TIMESYNC NLM by supplying an external time signal or detecting and correcting for an inaccurately ticking clock. This can be accomplished by setting the UTC clock or adjusting the *tickIncrement* field as needed and allowing TIMESYNC to continue functioning as usual. TIMESYNC was designed not to change the tick increment when correcting the clock so that an add-on product can safely do so.

Because there is no convenient interface with TIMESYNC that allows you to detect when the next synchronization period will start, you must rely on the behavior of TIMESYNC and do some monitoring. If you need to know when the next attempt to synchronize will occur, look at the *adjustmentCount* value. The value that TIMESYNC places in the *adjustmentCount* is calculated to complete the adjustments a few ticks before TIMESYNC begins the next synchronization attempt. Of course, if the value is 0, you don't know whether synchronization is about to begin or that no adjustment was necessary during the last attempt.

The most reliable way to cooperate with TIMESYNC is to set the *adjustmentCount* to 0 and clear the network synchronized status flag at the time you change other fields. This has the effect of cancelling the previous adjustment while notifying TIMESYNC that it should immediately begin another synchronization attempt. The following example demonstrates this operation.

Cooperating with TIMESYNC NLM

```
SetTickIncrement (long value)
{
   /* Sets the tick increment and notifies TIMESYNC that something has
   /* Cancels any pending adjustments */
   /* Parameter value is a fractional second -
      the whole seconds will be set to zero */

   long   mask;
   Synchronized_Clock_T   aclock;

   /* Be sure to save the status bits we aren't changing */
   GetSyncClockFields (SYNCCLOCK_STATUS_BIT, &aclock);
   aclock.statusFlags &= (~CLOCK_IS_NETWORK_SYNCHRONIZED);
                                    /* Notifies TIMESYNC of changes *

   aclock.tickIncrement[0] = 0;       /* Clear whole seconds */
   aclock.tickIncrement[1] = value;   /* Set fractional seconds */
   aclock.adjustmentCount = 0;        /* Stop applying old adjustments
```

```
    /* Copy the new values into the synchronized clock structure */
    mask = SYNCCLOCK_STATUS_BIT | SYNCCLOCK_TICK_INCREMENT_BIT |
           SYNCCLOCK_ADJUSTMENT_COUNT_BIT;
    SetSyncClockFields (mask, &aclock);
}
```

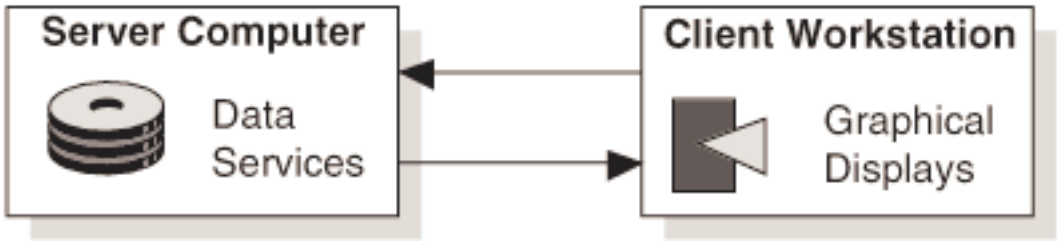
IMPORTANT: If an external time signal is being supplied through a separate NLM, the TIMESYNC HARDWARE CLOCK and TYPE options must be configured so that TIMESYNC does not fight the new time signal. That means on single and reference servers the hardware clock option should be off.

Introduction to Client-Server Applications

Distributed applications running on the NetWare® OS typically use the client-server model. (Peer-to-peer models can also be used, but here we address the more common client-server model.) In the client-server model, the client program makes service requests of---and receives replies from---a separate server program. These services can be data management, print and communication services, computation, and so on.

The following figure shows a client-server application where the computer specializing in database file management acts as the server, and the workstation specializing in graphical displays acts as the client.

Figure 9. Client-Server Application



The client-server model enhances distributed applications in many ways. For example, the client-server model offers the following advantages:

By offloading some application processing to the server, the application can take advantage of specialized, high-performance software and hardware on the server.

By moving client and server tasks closer to the network resources they use, the client and server can accomplish their tasks more efficiently.

By reducing client resource use, the client workstation can run larger

applications.

Through centralization, security, concurrency, and data integrity, the application can be improved.

In general, the NLM™ client-server model has these characteristics:

The server program resides on the server as an NLM.

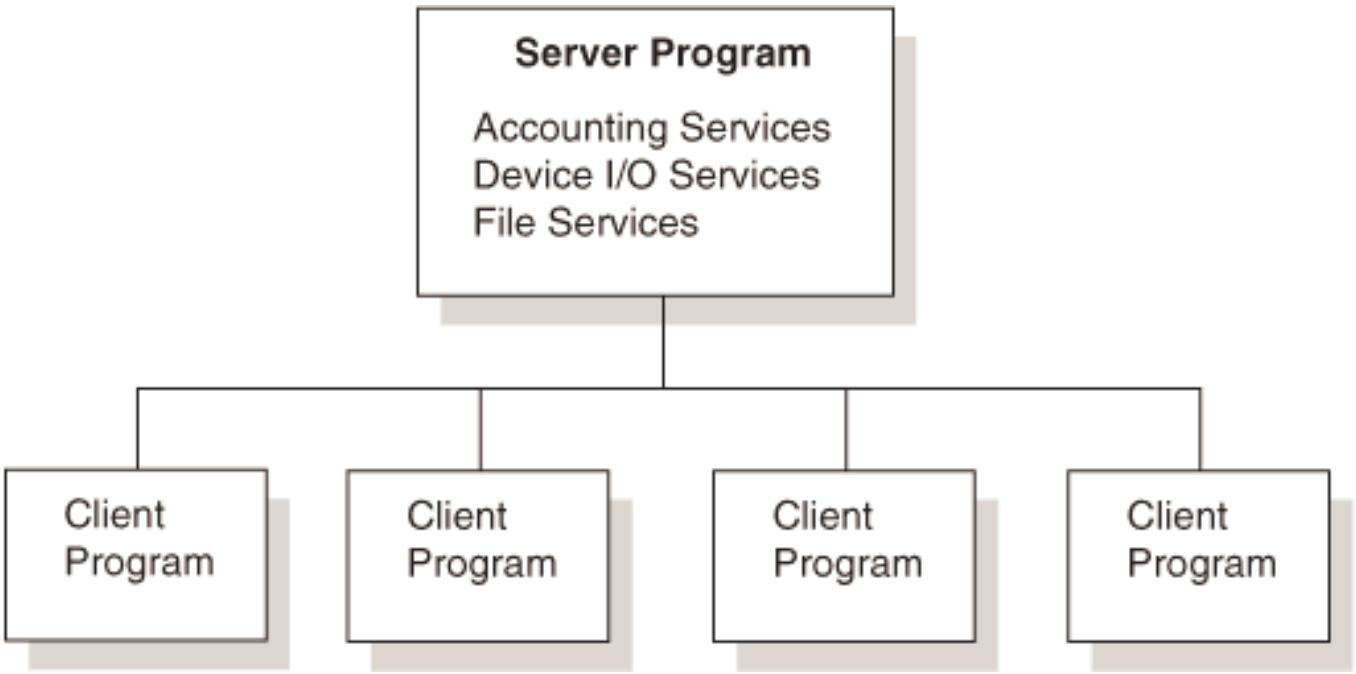
The server program sets up its services, initializes variables, and waits for client requests. The server advertises its services, but does not try to find clients.

When a client program needs a service, the program finds the server and then issues a request for service.

Client-server communication occurs as a request/reply pair, with the client making the request and the server issuing the reply.

The server program (NLM) often services multiple clients. The following figure shows the distribution of an example archiving application, in which a single server program offers file backup services to multiple client programs. When a client needs those services, it sends a request to the server.

Figure 10. Server with Multiple Clients

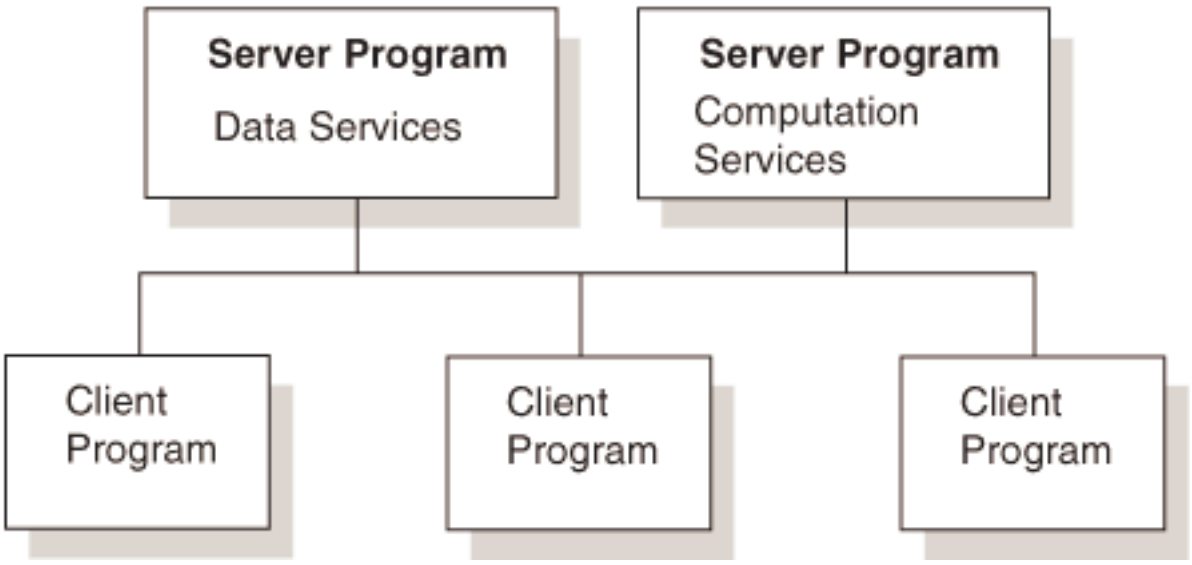


The example archiving application uses accounting services to charge

clients for network usage. The server uses file services to find and copy client files. These files could be stored anywhere on the network.

Similarly, in a more complex configuration, each client might access more than one type of server program to complete its task, as shown in the following figure. This figure shows the distribution of a database application in which multiple server programs provide data and computation services for multiple client programs.

Figure 11. Distributed Application



In this sample database application, multiple servers offer special services to each client. The data services server offers data storage and retrieval. The computation services server offers calculation-intensive services, such as regression analysis. When a client needs a service, it sends a request to the appropriate server. The use of multiple servers can enhance performance by optimizing the use of network resources. For example, the server that offers computation services might be able to calculate more quickly than other servers.

Related Topics::

Designing Client-Server NLM Applications

Locating Services

Client-Server Communication

Introduction to Remote Server Support

Remote server support provides an NLM™ application 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 into is considered a **remote** server.

Servers are identified by a server ID number.

> **NOTE:** 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 function provides remote support, see the "Remote Servers" notation on its function description.

Related Topics:

Accessing Remote Servers

Changing the Current Server

Logging Out from Remote Servers

Remote and Local Server Operations

Introduction to SFT III

SFT III™ is a method of mirroring the state of a NetWare® server on a second server such that the resulting entity, the **SFT III server**, is resilient to any single hardware failure. If the primary server fails or is halted by the console operator, the secondary server instantly (and transparently to the client) becomes the active component of the logical server. The result is uninterrupted service to the client. Clients using the SFT III server see no loss of state or service after any failure (other than perhaps a slight glitch as the LAN communications protocol changes its route to the server).

The primary server (machine #1) and the mirrored secondary server (machine #2), connected by a Mirrored Server Link™ (MSL™) connection, function together as one logical server. Only the primary server appears to clients on the internetwork as a NetWare "server." The secondary server maintains an event-by-event mirror of the primary server's activities.

All server application modules that don't require direct access to the

All server application modules that don't require direct access to the hardware can be mirrored transparently, including data base applications and alternative file service protocols.

Related Topics:

Mirrored Server Link

IOEngine and MSEngine

Primary and Secondary Server

SFT III Server Memory Management

NLM Applications and SFT III

Dual-Processing Support

Introduction to the NLM Assembly Interface

This information does not intend to be complete reference on calling C functions from assembly code. Rather, this information provides guidelines for your assembly code that will call functions in theNetWare® API or will provide assembly routines to be called by the NetWare OS. This chapter refers to stack-based parameter passing that is used by the NetWare API.

This information describes the guidelines you should follow when combining C code and assembly code. It includes guidelines for assembly code that calls the NetWare API. It also includes guidelines for assembly procedures that will be called by the NetWare OS; for example, callbacks.

> **NOTE:** The following topics make assumptions about the behavior of C compilers. Do not assume that your compiler and assembler make these same assumptions. Instead, first check your compiler's and assembler's manuals to verify that they behave in the manner described here.

Related Topics:

General Purpose Registers

Segment Registers

C Code That Calls Assembly Procedures

Assembly Code That Calls C Functions

Parameter Passing

IOEngine and MSEngine

The NetWare SFT III™ OS is divided into two distinct executable modules

called the IOEngine and MSEngine. Both the IOEngine and the MSEngine can be viewed as entities in and of themselves---they each have their own scheduler and memory manager.

The **IOEngine** (input/output engine) contains all code that requires direct access to server machine hardware. All device drivers are loaded and executed by the IOEngine.

The **MSEngine** (mirrored server engine) contains only code that does not require direct access to servermachine hardware. This includes the NetWare logical file system, bindery, NCP™ service, queue services, and so on.

When two SFT III servers are mirrored, their IOEngines contain different memory images, and consequently are not mirrored. This allows two mirrored servers to have different hardware configurations. All hardware-specific details are compartmentalized within the IOEngine and sealed off from the logical workings of the MSEngine.

The MSEngines of two mirrored SFT III servers contain identical memory images, and are mirrored such that there is only one logical MSEngine. Because the MSEngine provides all the characteristics clients associate with a NetWare "server," the two mirrored servers present internetwork clients with the illusion that the mirrored machines are a single logical "server." (The two IOEngines appear to clients as two stand alone internetwork routers.)

On a single NetWare SFT III server, the IOEngine and MSEnine communicate with each other using a special message-passing protocol. The IOEngine converts hardware-related processing into a stream of "events," which it passes to the MSEngine. Likewise, the MSEngine converts calls to hardware device drivers into a stream of "requests," which it passes to the IOEngine.

Most device drivers that adhere to the NetWare ODI™ specification work unmodified with NetWare SFT III. Both the IOEngine and the MSEngine maintain the standard higher-level device driver interface.

IOEngine Applications

Any application that requires specific knowledge of or access to server hardware must be loaded in the IOEngine. Typical IOEngine applications are device drivers, although other types of applications can be required to run in the IOEngine. Applications that do the following are required to run in the IOEngine:

Hook interrupt vectors

Call interrupt vectors

Generate real-mode interrupts directly

Execute conditionally upon hardware configuration

Require specific hardware information

It is possible that some applications that are not required to run in the IOEngine can find an advantage in doing so. In order to run in the IOEngine, an application cannot use any of the following NetWare® services:

File services (including TTS™ services, OS I/O, and any service which uses the NetWare file system)

Connection-oriented services

Queue services

Bindery services (including any service which makes use of the bindery)

NCP™ services

Accounting services

AFP services

NOTE: Some of the services listed above can be available to IOEngine applications if there are remote servers on the internetwork. For more information, see The NetWare API and SFT III.

However, when the MSEngine is loaded, NLM™ applications can call functions provided by the services listed above **provided such NLM applications are written and compiled using the NetWare API**. The NetWare API provides NLM applications running in the IOEngine with access to MSEngine services, including all services related to the NetWare logical file system.

Other services are available to applications running in the IOEngine, including synchronization and thread control, memory allocation and management, string manipulation, screen handling, and so on. The IOEngine provides standard NetWare routing and communications services, including STREAMS, IPX™, and SPX™ services.

Unless required to do so by its design, it is inappropriate for any application that benefits from mirroring to run in the IOEngine. However, applications that add value to the internetwork environment and do not gain benefit from mirroring can be appropriate for running in the IOEngine. The following types of applications are examples:

Specialized routers

Network management agents

Monitoring agents

The IOEngine itself performs internetwork-related utility functions, including IPX routing and acting as a network bridge.

LAN and Device Driver Modules

LAN drivers control communication between the OS and the network boards; disk drivers control communication between the OS and the hard disk controllers installed in your server.

A number of LAN drivers (NE1000™ and NE2000™) and disk drivers (ISADISK) are provided by Novell® and are ready to load into NetWare®.

Developers of LAN driver NLM applications use the LSL™ functions, which are part of the ODI™ specification. This specification, which Novell and Apple* developed together, allows more than one communication protocol stack (such as IPX/SPX™, AppleTalk*, or TCP/IP) to share a driver/adapter, and allows one protocol stack to use more than one driver/adapter.

A driver written to the ODI specification is known as a MLID™ application. MLID applications receive data from adapters and transmit it to the LSL. An MLID does not attempt to interpret the data in a packet. Instead, it passes packets to the LSL, which acts as a type of switchboard to route packets.

Novell Labs provides and supports a LAN Driver Developer's Kit and a Device Driver Developer's Kit. For more information, call 1-801-429-5544.

Loading Multiple NLM Applications

The NetWare® API allows multiple NLM™ applications to run simultaneously. Most NLM applications rely on services that other modules provide. This interdependence is why many NLM applications 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 a new NLM that requires services provided by an NLM that is not present, and the NLM does not autoload the needed NLM applications, the new NLM will not load.

Loading NLM Applications

NLM™ applications can be loaded and unloaded from the server while the server is running. You can load NLM applications 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. (See Using the LOAD Command for more information about NetWare API parameters and Adding Console Commands: Example.)

As NetWare loads the NLM, it resolves all unresolved external calls and initializes the module.

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

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

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

Regardless of how an NLM is loaded, there are certain rules that the NetWare OS 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 DOS 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.

If you do not specify an extension, the extensions .NLM, .LAN, .DSK, and .NAM are applied, in that order.

The OS must be able to find the NLM applications when a **LOAD** command is issued. You can load NLM applications 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:

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
```

Related Topics:

How NLM Applications Are Loaded

Using the LOAD Command

Adding Console Commands: Example

Unloading NLM Applications

Locating Bindery Services

The NetWare® bindery is a database stored on the server. This database contains information about resources (such as other servers, print servers, and database servers) and about the users who can access and use those resources. Because each server on a network stores resource information, clients can scan the bindery to find all available services.

On NetWare 2.x and 3.x servers, the bindery is stored on each NetWare server. NetWare 4.x servers do not store a bindery; instead, they simulate a bindery, using objects found at a specific context within the Directory tree.

For servers before NetWare 4.0, the bindery is the basis on which NetWare login security mechanisms are built, including password protection and client restrictions. (NetWare 4.x servers can use the security of the Directory or the security of the bindery.)

> **NOTE:** The security of the bindery is independent of the security of the file system. The bindery does not store any of the file system's directory trustee information. Directory trustees are stored in directory entries, which are an integral part of the NetWare directory structure.
>
> The main relationship between the bindery and the file system is that the file system stores each directory's trustees in the form of a bindery object ID.

The bindery is composed of **objects** and **properties**. An object can be any logical or physical entity that has been given a name. For example, an object might be a job queue.

Each object also has associated with it a set of characteristics called properties, and each property has one or more property values. For example, the object DAN, a user, might have associated with it the

example, the object DAN, a user, might have associated with it the properties GROUPS_I'M_IN, ACCOUNT_BALANCE, and PASSWORD. The property PASSWORD contains user DAN's login password.

For more information on the bindery, see Bindery.

Locating BSD Socket Services

The BSD Socket API helps you port TCP/IP-based applications to the NetWare® environment. In addition, you can use these functions to write new TCP/IP-based applications for NetWare.

The BSD Socket API resembles the UNIX* file system interface. Instead of opening a file on a disk, the application creates a socket and uses the socket's I/O descriptor to read, write, and close a socket as if it were a disk file. Each socket created by an application has a unique network address.

Because of the complex nature of network communications, a NetWare application must invoke several operations that are not required by the UNIX file system interface. For example, whereas a typical NetWare application might send data to a filename, a UNIX application can specify the address of the destination socket.

For more information on the BSD Socket API, see BSD Socket.

Locating Connection and Task Services

Before a client receives service, it usually establishes a connection with the server. In a distributed environment, a server often manages several clients concurrently. Likewise, a client can attach to several different servers.

When an NLM™ application offers service to more than one client, that NLM must be able to associate its tasks with the requesting clients. Otherwise, tasks might return values to the wrong clients.

A NetWare® connection is established when the client attaches to the server. NetWare servers identify their client connections through **connection numbers**.

> **NOTE:** Do not confuse generic connections with NetWare connection numbers. In general, a connection is a link between two computers. Specifically, a NetWare connection number is the unique number assigned by a NetWare server to identify its attached clients.

NLM applications can use these NetWare connection numbers to their benefit, but they are not required to. NLM applications can use connection numbers to request service on behalf of a client. This technique allows the NLM to use NetWare features, such as security, accounting, and resource cleanup.

NetWare connection numbers and task numbers allow the OS to provide the following:

Security

When an NLM requests a service, NetWare uses the connection number to verify that the client or NLM has the appropriate access rights.

Accounting

In a system where the NetWare accounting feature is installed, the price of servicing a request is charged to a client or NLM based on its connection number.

Resource management

When a connection logs out, NetWare frees the second-level files (opened by **fopen** or **fdopen**) allocated to that connection number.

Alternatively, an NLM can bypass these features by using its own method for identifying clients. For example, an NLM might identify clients by their network addresses. However, any NetWare features (such as security) that the NLM might want to use would need to be provided by the NLM itself.

Related Topics:

NetWare Connection Numbers

NetWare Task Numbers

Locating IPX/SPX Services

The NetWare® API provides IPX/SPX™ services for use with the IPX/SPX protocols. These communications protocols are derived from theXerox* Network Systems (XNS*) Internet Transport Protocols.

The Communication Services functions enable a program to send IPX or SPX packets directly to other programs on an internetwork or to receive such packets directly from other programs. This capability is known as peer-to-peer communication.

IPX is a best-effort (datagram) delivery service. LAN drivers make a best-effort attempt to deliver packets, but do not guarantee delivery.

IPX communication requires more of the developer's time. However, for NLM™ applications that value speed over guaranteed delivery, IPX is a good solution. The NLM runs faster than with SPX because IPX does not maintain the overhead required to establish connections, sequence packets, guarantee delivery, or track undelivered packets.

In contrast, SPX is a guaranteed delivery service. Applications sending SPX packets form SPX connections with destination applications, and SPX retransmits any unacknowledged packets after appropriate timeout

intervals. After a certain number of unacknowledged retransmissions, SPX assumes that the destination application is no longer listening and breaks the connection.

For more information on NetWare IPX/SPX, see IPX/SPX.

Locating NDS Services

The NDS™ database, which is new with NetWare® 4.x, introduces a new component to the NetWare environment: the Directory. The Directory is a distributed database of network objects. It replaces the bindery, which served as the system database for previous releases of NetWare. Although the bindery is designed to support the operation of a single server, the Directory is designed to support the entire network. Now, clients can scan the Directory to find information about objects on the entire network.

> **NOTE:** It is critical that you learn to program using the Directory. Although there are Bindery Services in the NetWare 4.x OS, the Directory offers substantially more information and flexibility.

For more information about how to scan for objects in the Directory, see NDS.

Locating SAP Services

NLM™ applications can use SAP to advertise their presence on the network. When a server receives the service advertising packet, it adds that service object to its bindery. Servers broadcast their service availability over the network. In turn, servers receiving those broadcasts rebroadcast the availability to other servers. In this way, service availability is propagated throughout the network. Any client on the network can find a service provider by logging into a server and scanning its bindery.

By using the bindery, it is possible for clients to easily locate service providers by both name and type. Each server object in the bindery has a name and type that identifies it uniquely on the network. In cases where multiple servers offer the same type of service (such as print services), clients can also request "nearest" service.

Client applications can also bypass the bindery's service lookup and use SAP to directly query for the presence of servers on the network. Using **QueryServices**, an application can send a service query broadcast and wait for replies.

SAP is based on the IPX™ protocol. SAP defines special kinds of IPX packets, called service advertising packets.

For more information on SAP, see SAP.

Locating Services

When a client program needs a particular service, it first must locate a service provider. The client can use one of many methods for locating the server:

NDS™ database

NetWare® 4.x replaces the bindery with the Directory. The Directory is a hierarchical database of the complete network. A client can search through the Directory, looking for objects of the type it needs. (See NDS.)

Bindery Services

Every NetWare 3.x server contains a bindery that, among other things, identifies available services and service providers. Clients can scan this bindery to find services. You must use the bindery if your NLM™ application is loaded on a NetWare 3.x server. (See Bindery.)

NetWare 4.x servers do not have a bindery; instead they simulate a bindery, using the objects at a specific location within the Directory tree. This means that an NLM that uses the bindery can run on NetWare 4.x servers as well as 3.x servers.

SAP Services

Service providers can use SAP to advertise their presence on the network. In addition, clients can query for services by broadcasting a packet that specifies the type of service they want and whether they want to find all services on the network or the nearest service. (See SAP.)

Communication Services

If the application does not require communication through servers or bridges, the client can use the Communication Services functions to accomplish service-to-address resolution through IPX/SPX™ protocols. (See IPX/SPX.)

TLI

The client can use TLI to accomplish service-to-address resolution through protocol suites such as the ISO protocols, TCP/IP, and IPX/SPX. SPX II was added for NetWare 4.x. (See TLI.)

BSD Socket API

The client can use the BSD Socket API for service-to-address resolution through TCP/IP. (See BSD Socket.)

The client typically establishes a connection with the service provider, then, depending on the client program, either waits for a reply or continues running other processes.

Related Topics::

Locating TLI Services

TLI is a transport service interface that enables applications and higher-layer protocols to be implemented without knowledge of the underlying protocol suites.

TLI provides a common interface to several protocol suites, including:

- ISO protocols
- TCP/IP
- IPX/SPX™ protocols

TLI is implemented as a user library using the STREAMS I/O mechanism. Therefore, many services available to STREAMS applications are also available to users of NetWare® TLI.

TLI provides two modes of service, a connectionless mode and connection-oriented mode. Although connectionless mode is faster, it is less reliable because packets are not sequenced and delivery is not guaranteed. By contrast, the connection-oriented mode guarantees delivery and preserves packet order.

By defining a set of services common to many transport protocols, TLI offers protocol independence for user software. Thus, a distributed application written using NetWare TLI is portable to a wide range of transport protocols.

For more information on NetWare TLI, seeTLI.

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.

> **CAUTION: 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.**

For more information on resource locking, see NLM Synchronization.

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.)

Memory Considerations When Converting VAPs

The NetWare® 3.x and 4.x OS's run on 32-bit computers that can address up to 4 GB of memory. When converting a VAP to an NLM, you must alter the program's memory usage. NLM and VAP memory usage differs in two areas:

Memory models---NLM applications use the flat memory model (see Flat Memory Model).

Register size---For those developers working in assembly language, NLM

applications use 32-bit registers (see 32-bit Registers).

Mirrored Applications

The primary advantage of the NetWare SFT III™ development environment is that applications can take advantage of server mirroring automatically. That is, an application running on a mirrored server is also mirrored by default. The single requirement is that the mirrored application must be loaded into the MSEngine. Applications loaded into the IOEngine are never mirrored (just as the IOEngine is not mirrored).

Application mirroring is especially attractive to developers of distributed (client-server) applications. Just as NCP™ clients of a mirrored server experience no loss of state when the primary server suffers a fault, clients of a mirrored server application experience no loss of state when the application's server hardware suffers a fault.

Mirrored Server Link

Two mirrored NetWare SFT III™ servers must be connected by a high-speed Mirrored Server Link™ (MSL™). The MSL should be a low-latency medium with a capacity equal to the combined bandwidth of all network boards attached to either mirrored server. The MSL is the channel that two mirrored servers use to communicate with each other and to maintain their synchronization. For more information on the MSL, see the *Mirrored Server Link Driver Specification*.

More Detail for High Accuracy Time Synchronization Users

Careful readers will note that setting local time normally clears the subsecond counter in the UTC clock. When **SetSyncClockFields** recalculates local time, it saves and restores the subsecond counter rather than letting it be set to 0. This is a pragmatic solution to a problem that results from implementing a request to clear the counter. If a timer interrupt occurs during the time that **SetSyncClockFields** is setting local time and the hardware clock, the tick is lost when the subsecond counter is restored. Furthermore, if the lost tick caused the whole seconds to increment, the restored value causes it to increment again during the next interrupt---that is, the lost tick results in a gained second.

To avoid this situation, check the subsecond counter before updating the hardware clock and only perform the update when the subsecond counter is small. Alternatively, delay setting the UTC clock and hardware clock until the subsecond counter in the UTC clock can be set to a known value (such as

0) so that the problem does not occur.

Modular CLIB

CLIB.NLM versions previous to 4.0 were very large libraries containing the funcitonality now divided among the following modular libraries:

CLIB.NLM---ASNI compatability
THREADS.NLM---Support for NetWare® threads
REQUESTR.NLM---NetWare Requester™ support
NLMLIB.NLM---POSIX and NetWare support
FPSM.NLM---Floating-point decimal support
NIT.NLM---Old NetWare server functions now provided in cross-platform libraries through CAL32NLM.NLM.
MATHLIB.NLM---Stub
MATHLIBC.NLM---Stub

When this modular version of CLIB.NLM is loaded, its dependencyNLM™ libraries load automatically unless they are already loaded.

These modules collectively contain NetWare API functions that your NLM applications can call 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.

Header Files in Modular CLIB

The headers files in the old CLIB have gone through extensive changes in the transition to modular CLIB. These changes are laid out in tabular form in Modular CLIB Header Files.

Modular CLIB, NetWare 3.12, and NWSNUT

When modular CLIB is installed on a server running NetWare 3.12, the NWSNUT user interface can be used only of the following NLM applications are loaded:

AFTER311.NLM
A3112.NLM

These two NLM applications can be found on the Novell® SDK CD-ROM at the followng path:

NWSDK\LIB\NLM\4_X

Parent Topic:

NetWare API

Parent Topic:

Modular CLIB Header Files

MSEngine Applications

Most server-based applications for NetWare SFT III™ should run in the MSEngine. By doing so, they gain the benefit of mirrored execution automatically. However, applications running in the MSEngine must remain ignorant of the server's underlying hardware configuration. Because the mirrored "shadow application" might be running on a server with a different hardware configuration, any knowledge of server hardware obtained by the mirrored application breaks the SFT III architecture.

The following operations are invalid for MSEngine applications:

- Hooking interrupt vectors
- Calling interrupt vectors
- Generating real-mode interrupts directly
- Executing conditionally upon hardware configuration
- Obtaining specific hardware information

Most server-based applications do not need to attempt any of the above operations, because in every instance, the SFT III OS provides applications with the necessary support, either through virtualized devices or using its inter-engine message passing protocol.

For example, the MSEngine provides a function that mirrored applications

can use to load NLM applications from the server's DOS partition. When a mirrored application calls this function, the call eventually resolves to a real-mode interrupt in the IOEngine. Nevertheless, the details are hidden from the mirrored application, thus preserving the integrity of the SFT III architecture.

Applications loaded in the MSEngine gain access to all of the higher-level services of NetWare, such as file service, TTS™, NCP™ service, queue services, the bindery, and so on.

Related Topics:

SFT Level II

Drive Duplexing

Transaction Tracking and SFT III

Multithreaded Programming

Multithreading is common in multiclient distributed applications. Typically, a client-server NLM™ application uses a different thread group for each client to which it provides service. This allows the NLM to service multiple clients concurrently. In addition, NLM applications often establish specialized threads, such as display, input, and communication threads. These threads can be used, for example, to accept commands from the server console, receive incoming requests, and send outgoing replies.

An efficient way to handle multiple clients is for the NLM to create a new thread for each client it services. As the NLM receives client requests, it creates a new thread to process each request. Then, after the request is serviced, the thread runs to the end of its initial procedure and is terminated.

There are cases where the method mentioned above would not be efficient. For example, if you are servicing 250 users and have 250 threads with 8 KB stacks, then just the stacks of these threads take up 2,000 KB of memory. In this case you might want to establish a pool of threads to handle multiple clients. As the NLM receives client requests, it selects a free thread to process the request. After the thread processes the request, it returns to the pool of free threads.

Using multiple threads has many advantages. It allows you to:

Simplify code through modularization

By separating processes into threads, programs become more easily read, maintained, and updated. Multithreading relieves the developer of having to use task switching logic.

Increase throughput

By dividing the NLM into multiple threads, you can reduce the

amount of time the CPU remains idle. Instead of blocking during I/O requests, the OS switches control to another thread and more fully utilizes the CPU.

Enhance response time

Because the server is able to switch between threads (thus preventing a single thread from monopolizing the CPU), clients receive faster replies to their requests. A lengthy I/O-intensive request from one workstation does not preclude the completion of a smaller request from another workstation.

Develop multiple contexts through thread grouping

A thread group consists of one or more threads, as defined by the programmer. Threads in the same thread group share the same context, such as the CWD and current connection. This provides the programmer with shortcuts, such as the ability to use the CWD instead of specifying the full pathname.

The advantages of a multithreaded application are increased performance and efficiency. In a multithreaded application, processes get more equal time to use system resources. Additionally, any thread can process separately from other threads. For example, a process can update files in the background while a foreground process produces data that needs to be written to those files. Similarly, one thread can be used to interact with a client process while performing complex, time-consuming computations in the background.

The following is a simple example showing the creation of multiple threads:

Creating Multiple Threads

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


int   getOut = FALSE;
int   twoOut = FALSE;
int   threeOut = FALSE;
int   fourOut = FALSE;


void ThreadTwo();
void ThreadThree();
void ThreadFour(void *data);


main()
{
   BeginThreadGroup(ThreadThree, NULL, NULL, NULL);
   ThreadSwitch();

   BeginThread(ThreadTwo, NULL, NULL, NULL);
```

```
   ThreadSwitch();

   while (!kbhit())
      printf("Thread One.\n");
   getOut = TRUE;

   // allow all threads to clean up before NLM exits
   while (!(twoOut && threeOut && fourOut))
      ThreadSwitch();
}


void ThreadTwo()
{
   while (!getOut)
   {
      printf("           Thread Two\n");
      ThreadSwitch();
   }
   twoOut = TRUE;
}


void ThreadThree()
{

   BeginThread(ThreadFour, NULL, NULL, "THREAD FOUR");
   while (!getOut)
   {
      printf("In Thread Three\n");
      ThreadSwitch();
   }
   threeOut = TRUE;
}


void ThreadFour(void *data)
{
   while (!getOut)
   {
   printf("            %s\n",(char *) data);
   ThreadSwitch();
   }
   fourOut = TRUE;
}
```

See Thread for more information about threads, thread groups, and the context that the NetWare API maintains. The Context also discusses context issues.

Name Space Modules

Name space NLM™ applications allow a file to be accessed using a variety of naming conventions. The NetWare® OS allows multiple name spaces on the same volume. Therefore, a single NetWare server running multiple name space NLM applications can offer file storage for a range of file naming conventions.

For example, a server could offer support for DOS, NFS*, OS/2*, and OSI (via FTAM.NLM) naming conventions on the same volume. A client workstation running DOS could read a file created by an OS/2 workstation. The OS/2 filename could be displayed on the DOS workstation.

Support for DOS naming conventions is hard-coded in the NetWare OS.

NetWare API

The NetWare® API provides over 1,000 NLM™ functions, allowing an NLM direct access to NetWare OS services. The NetWare API is a load-time link library consisting of several modules that can be accessed concurrently by multiple NLM applications.

The modules available for application development, combined with the underlying architecture of the NetWare OS, provide the functionality required to build high-performance, server-based systems.

The following NLM modules ship as part of the NetWare API:

CLIB.NLM
NIT.NLM
NLMLIB.NLM
FPSM.NLM
REQUESTR.NLM
THREADS.NLM
CLIBDEB.NLM
DSAPI.NLM
MATHLIB.NLM
MATHLIBC.NLM
NWSNUT.NLM
TLI.NLM

Related Topics:

Modular CLIB

Cross-Platform Functions in NLM Applications

NetWare Connection Numbers

NetWare® assigns a unique connection number to each client as the client attaches to the server. Numbers are assigned sequentially, beginning with connection number 1. As long as the client maintains an open connection with the server, NetWare maintains that connection number for the client. When a client closes a connection, its connection number is held in reserve for a period of time in case of client reconnect; then the connection number goes back into the pool of available connections.

An NLM™ application can specify one of three types of NetWare connection numbers:

Connection number 0

This bypasses the regular client security and provides the NLM with supervisor rights to the NetWare file system. Multiple NLM applications can use connection number 0 concurrently. This connection number could be used, for example, by monitor or control NLM applications whose resource usage does not require restrictions or accounting.

Existing connection number assigned to a client

This allows NLM applications to make requests on behalf of the client associated with that connection number. Thus, the NLM assumes the client's access rights and privileges while it is executing the client's request. In this way, an NLM can maintain network security and accounting when accessing resources on behalf of clients.

New connection number assigned to the NLM

This assigns the NLM a new connection number with the same access rights as an attached (but not logged-in) client.

If an NLM running on a NetWare 3.11 server logs in, it must log in as a bindery object. If the NLM is running on a NetWare 4.x server, it can log in as a Directory object or as a bindery object (if bindery context is set). This object can be created by the NLM itself or by another program. If the NLM logs in as a user, its access to the server can be restricted.

The NetWare 3.11 OS reserves 100 "hidden" connection numbers for NLM applications. These numbers begin after the last client connection number. For instance, in a 250-user system, NLM connection numbers run from 251 through 350. Likewise, in a 100-user system, the NLM connection numbers are 101 through 200.

The NetWare 4.x OS does not place a limit on the number of "hidden" connections that NLM applications can use.

For more information about connections, seeIPX/SPX.

NetWare Functions Not Supported by the MSEngine

The MSEngine provides a "standard" environment for NLM™ applications, except for the restrictions discussed earlier in this document. Specifically, NLM applications loaded in the MSEngine cannot obtain or require any specific knowledge regarding the underlying server hardware.

All services associated with a NetWare® server are available to NLM applications that are loaded in the MSEngine. However, you should not issue the following functions by a NLM running in the MSEngine:

ClearHardwareInterrupt
_disable
_enable
SetHardwareInterrupt

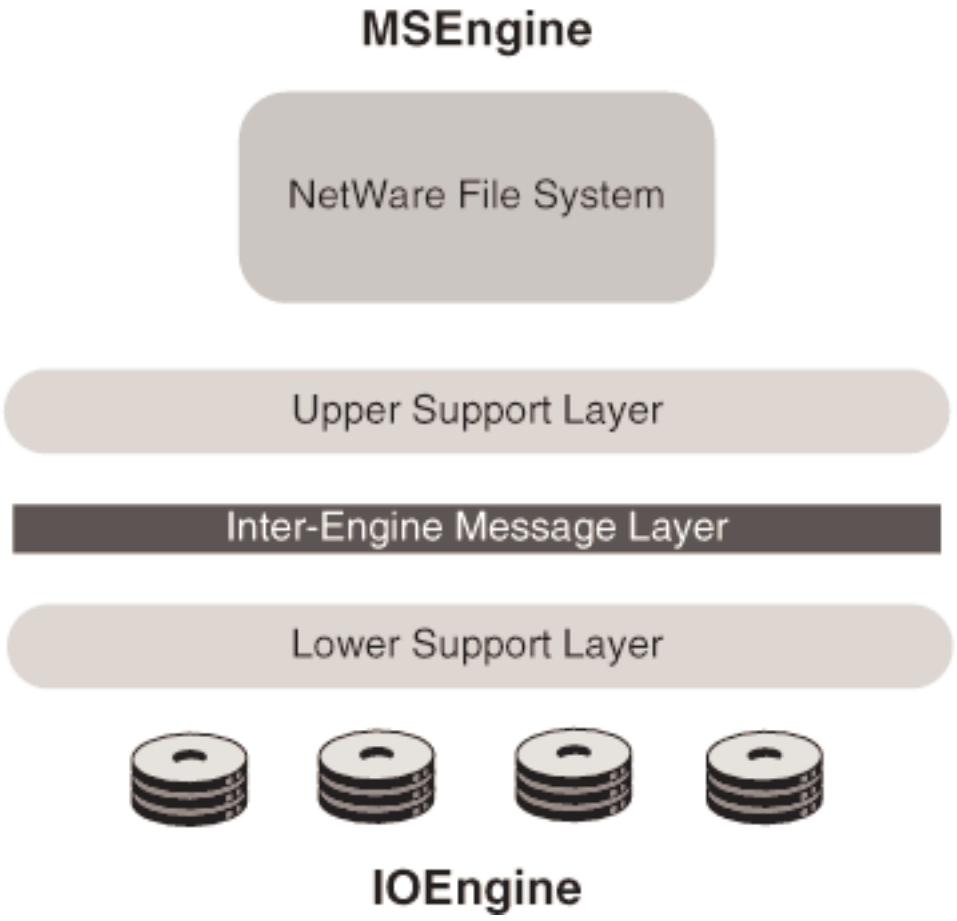
NetWare SFT III Support Layer Architecture

NetWare SFT III™ maintains the standard NetWare support layer architecture and extends it to support the division of the OS into the IOEngine and the MSEngine.

Support layers in SFT III maintain their two-phased interface. Hence, logical structures such as file systems or protocol stacks can run unaltered in the MSEngine, despite the fact that all device drivers are loaded in the IOEngine. Conversely, device drivers can run unaltered in the IOEngine, despite the fact that the logical structures that they are supporting reside in the MSEngine. Standard NetWare device driver compatibility is maintained by the insertion of an inter-engine support layer between the upper and lower interfaces of the device driver support layers.

The inter-engine layer, which resides in the "middle" of all SFT III support layers, is the same message-passing protocol mentioned in Introduction to SFT III. Data submitted to the support layer through its lower-level interface is converted into MSEngine events and submitted to the MSEngine side of the support layer. Data submitted to the support layer through its higher-level interface is converted into IOEngine requests and submitted to the IOEngine side of the support layer.

Figure 12. SFT III Support Layer Architecture



Most device driver developers can therefore remain unconcerned with the inner workings of SFT III, because all relevant interfaces remain unaltered. The exception to this rule concerns MSL or multiprocessing device drivers, which are specific to SFT III.

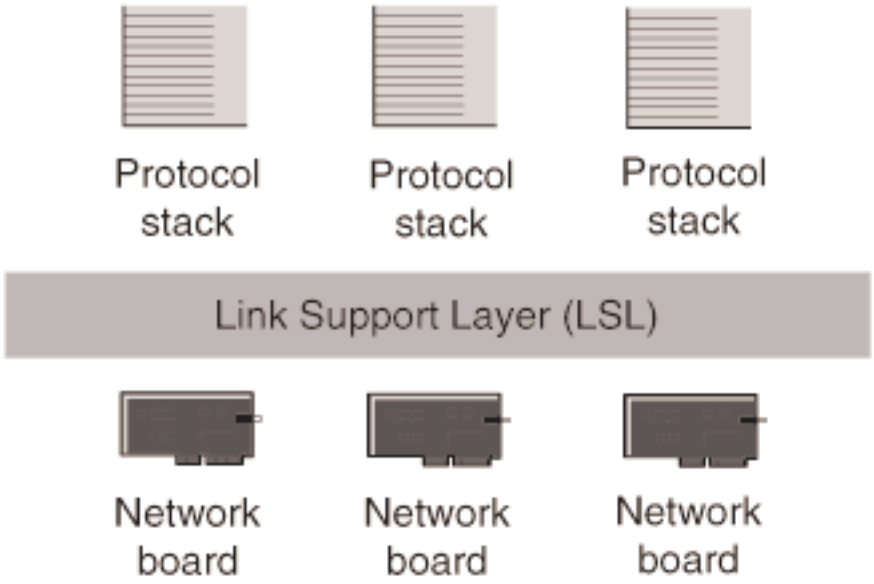
NetWare Support Layer Architecture

The support layer architecture for device drivers provides the NetWare® OS with important capabilities. For example, it allows the OS to maintain the same logical structures regardless of the underlying hardware.

For example, the IPX™ protocol stack implemented by the OS can remain unaware of the physical network media over which it is operating. This allows a single NetWare server to support a wide range of network media, such as ethernet, token ring, ARCnet*, and FDDI/DX*. The ability of NetWare to bridge different media types within a server is a direct result of the support layer architecture.

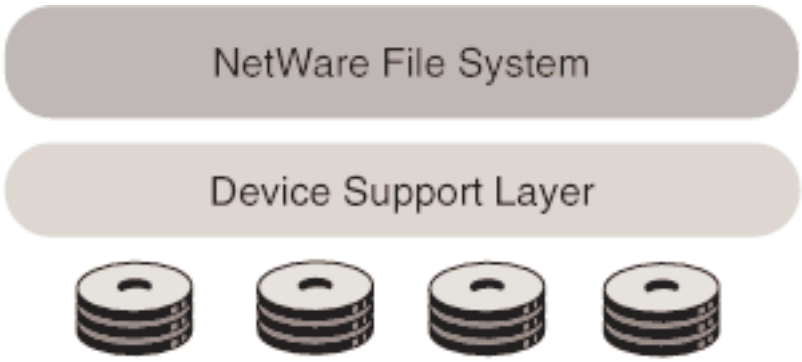
Likewise, the physical device driver can remain unaware of the logical format of the data it is transmitting or storing. For example, the ethernet driver can remain unaware of the transport stack it is supporting, whether it is AppleTalk, IPX, or IP. This is key to ability of NetWare to support multiple transport protocols within a single server.

Figure 13. Communication Support Layer Architecture



The NetWare disk driver specification allows logical structures such as volumes or file systems to remain unaware of issues such as drive mirroring or duplexing, split-seeking, partitions, and so on. Disk drivers can also remain unconcerned about these issues because mirroring, duplexing, partitioning, and so on, are all implemented at the support layer.

Figure 14. Device Support Layer Architecture



The NetWare support layer architecture requires a specific support layer to implement a two-phased API. For example, the LSL™ software must provide an interface that allows protocol stacks to receive and submit

network data. The LSL also must provide an interface that allows device drivers to receive and submit network data. One phase of the LSL API is strictly for stack communication between the LSL and the protocol, whereas the other phase of the LSL API is strictly for communication between the LSL and the device driver.

It is useful to speak of support layers as having upper-level and lower-level interfaces. The upper-level interface of the LSL provides communication between logical structures (protocol stacks) and the LSL, whereas the lower-level interface provides communication between physical device drivers and the LSL.

Regardless of the type of device, all NetWare support layers have the same basic architecture. They maintain a data base containing information about specific physical devices and their drivers, and act as a specialized router by ensuring that logical information is sent to or received from the correct physical device driver.

NetWare Task Numbers

In NetWare®, a task number identifies a program running on a network workstation or server. NetWare assigns task numbers sequentially, beginning with task number 1. The combination of the connection number and the task number yields a unique connection/task number pair. This connection number/task number is unique only to a given server.

NetWare uses the connection number/task number to manage network resources. Because client-server NLM™ applications can access resources on their own behalf or on behalf of a client, NLM applications must specify both a connection number and a task number when making requests.

> **NOTE:** If an NLM uses connection number 0 or a client's connection number to access a resource, the NLM should request a new task number to distinguish it from other tasks using the same connection number.

For more information on connections and tasks, seeConnection Number and Task Management.

New NetWare SDK Headers

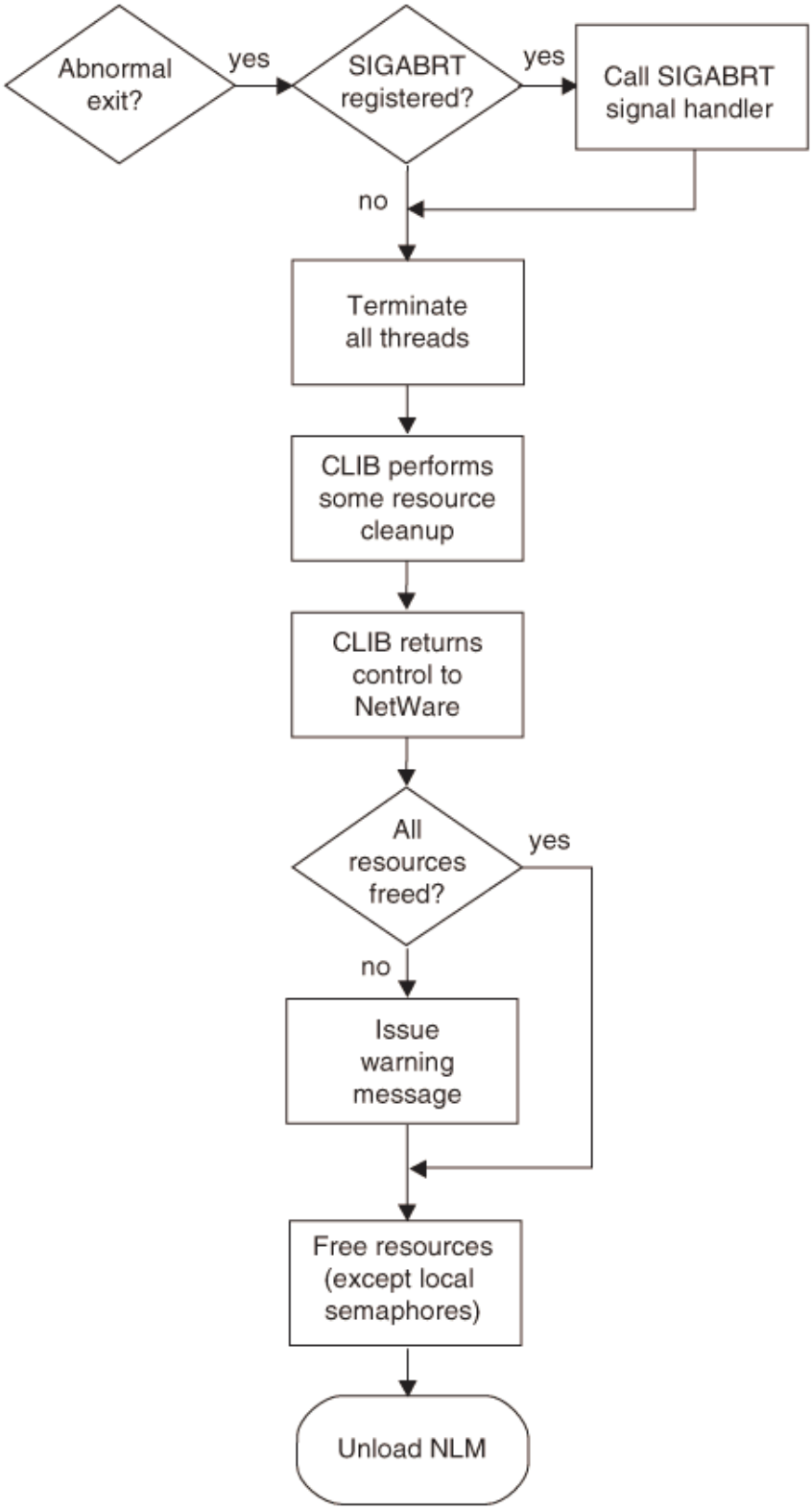
The following header files are new to CLIB 4.11, and had no equivalent in the "monolithic" CLIB:

NWSDK Header	Modular CLIB Module	Comment or Explanation

nwieeefp.h	CLIB.NLM	
dirent.h	NLMLIB.NLM	POSIX-compliant
nwbitops.h	NLMLIB.NLM	
nwbitops.h	NLMLIB.NLM	
nwtoolib.h	NLMLIB.NLM	
unistd.h	NLMLIB.NLM	POSIX-compliant
utime.h	NLMLIB.NLM	POSIX-compliant

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.


Abnormal exit?
yes
SIGABRT registered?
yes
Call SIGABRT signal handler
no
Terminate all threads
CLIB performs some resource cleanup
CLIB returns control to NetWare
All resources freed?
yes
no
Issue warning message
Free resources (except local semaphores)
Unload NLM

NLM Applications

The building blocks that customize the NetWare® 3.x and 4.x OS's are known as NLM™ applications. They are programs built to run in server memory with the NetWare OS. You can load or unload NLM applications from server memory while the server is running. Once loaded, NLM applications become part of the OS. This means that NLM applications can access NetWare services directly without using a service protocol such as the Novell® NCP™ service. The server procedures that NLM applications 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 NetWare 3.x and 4.x OS services.

In summary, NLM applications add openness, modularity, and flexibility to the NetWare OS, as follows:

- They can be loaded and unloaded as needed
- They can allocate and deallocate memory as needed
- They can link with and access the NetWare OS and other NLM applications

NLM applications, such as print and communication servers and server-based utilities, enable NetWare users to extend the flexibility and capability of their networks ever further.

NLM Applications and SFT III

Both the IOEngine and the MSEngine are allowed to load and run NLM™ applications. However, NLM applications loaded by the IOEngine cannot use any services implemented by the MSEngine, such as the logical file system, bindery, NCP™ service, and so on. (See The NetWare API and SFT III for exceptions to this rule.) Conversely, NLM applications loaded by the MSEngine cannot hook any hardware interfaces and must remain ignorant of machine hardware.

NLM applications loaded by the MSEngine are mirrored automatically whenever the SFT III server is mirrored. This is a major benefit of the SFT III™ design, because NLM programmers achieve mirrored operation without any special effort on their part. On the other hand, NLM applications loaded by the IOEngine are never mirrored.

NLM applications that use the NetWare® API should run unmodified on SFT III. The exception to this is the NLM that gains access to hardware directly (by hooking interrupts or some other method) and that also uses the

logical services of the NetWare OS.

Similarly, NLM applications which use undocumented system calls or that rely on the existence or location of undocumented labels can fail to load or work correctly under SFT III.

NLM Equivalents for VAP Functions

If your VAP uses functions from the NetWare® C Interface---DOS Library, you can easily convert the program to an NLM™ application that uses functions from the NetWare API. Most of the NetWare C Interface---DOS Library functions are the same in the NetWare API. This includes naming conventions, parameter types, return types, and structures.

The following section outlines similarities between the NetWare C Interface---DOS Library and the NetWare API. In cases where VAP functions do not have NLM function counterparts, substitute functions are suggested.

> **NOTE:** This is not intended as a comprehensive reference source. Thus, it does not discuss the exact differences between two particular functions. Rather, it highlights the areas of difference between the two libraries so that you can refer to the appropriate manual for additional information.

See the following tables for VAP equivalents:

VAP Equivalents for Accounting Functions

VAP Equivalents for AFP Functions

VAP Equivalents for Bindery Functions

VAP Equivalents for Communication Functions

VAP Equivalents for Connection Functions

VAP Equivalents for File System Functions

VAP Equivalents for Message Functions

VAP Equivalents for Miscellaneous Functions

VAP Equivalents for Queue Management Functions

VAP Equivalents for SAP Functions

VAP Equivalents for Server Environment Functions

VAP Equivalents for Synchronization Functions

VAP Equivalents for TTS Functions

VAP Services

Workstation Environment Services

NOTE: For equivalents for directory functions, see VAP Equivalents for File System Functions.

In addition to the above services, the NetWare API provides other services. To locate more information about specific services, see SDK Roadmap.

NLM Level Context

The NLM™ level context is shared by all thread groups and threads in the NLM, and these data items have only one value for the entire NLM. The data items are global to all the thread groups and threads in the NLM. Any changes made to the values of NLM global data items affect all the thread groups and threads in the NLM.

NLM applications maintain the following context on a per NLM basis:

"Active advertisers"

Each NLM can have a set of "active advertisers" (started by **AdvertiseService (NLM)**).

argv array

This is the *argv* array passed to **main**.

atexit, **AtUnload**

These functions register functions that are to be called when the NLM exits normally or is unloaded.

Libraries' work areas pointers

These are pointers to the data areas of any NLM libraries that the NLM has called, as described in Library.

Locale settings

These settings are used by the locale functions.

Open directories

The set of directories (opened by **opendir**) the NLM has open.

Open IPX/SPX/SPX II sockets

The set of IPX/SPX/SPX II sockets the NLM has open.

Open files

The set of files the NLM has open. First-level open files include those opened with **open**, **sopen**, **creat**; second-level files include those opened with **fopen**, **fdopen**, **freopen**.

Open network semaphores

The set of network semaphores (opened by **NWOpenSemaphore**) the NLM has open.

open screens

The set of screens the NLM has open.

original command line

A copy of the original command line when the NLM was started is saved (used by the **getcmd** function).

thread name

This pattern is used for naming new threads (used by **BeginThread** and **BeginThreadGroup**).

NLM Limited Support Functions

NLM™ Limited Support functions are functions that continue to be provided by the NetWare® SDK, but are only support by Developer Support and no longer have any engineering or documentation support. The Limited Support functions have been replaced by functions in NWCALLS (the cross-platform API). Documentation for these functions have been removed from this SDK because they are no longer maintained. Documentation for NLM Limited Support functions can be found in the NetWare Limited Support SDK.

The Limited Support functions are as follows:

Accounting Services

AccountingInstalled
GetAccountStatus
SubmitAccountCharge
SubmitAccountChargeWithLength
SubmitAccountHold
SubmitAccountNote

AFP Services

AFPAllocTemporaryDirHandle
AFPCreateDirectory
AFPCreateFile
AFPDelete
AFPDirectoryEntry
AFPGetEntryIDFromName
AFPGetEntryIDFromNetWareHandle
AFPGetEntryIDFromPathName
AFPGetFileInformation
AFPOpenFileFork

AFPRename
AFPScanFileInformation
AFPSetFileInformation
AFPSupported

Auditing Services

NWAddRecordToAuditingFile
NWGetAuditingIdentity
NWSetAuditingIdentity

Bindery Services

AddBinderyObjectToSet
ChangeBinderyObjectPassword
ChangeBinderyObjectSecurity
ChangePropertySecurity
CloseBindery
CreateBinderyObject
CreateProperty
DeleteBinderyObject
DeleteBinderyObjectFromSet
DeleteProperty
GetBinderyAccessLevel
GetBinderyObjectID
GetBinderyObjectName
IsBinderyObjectInSet
OpenBindery
ReadPropertyValue
RenameBinderyObject
ScanBinderyObject
ScanBinderyObjectTrusteePaths
ScanProperty
VerifyBinderyObjectPassword
WritePropertyValue

Connection Services

AttachByAddress
AttachToFileServer
GetConnectionInformation
GetConnectionList
GetConnectionNumber
GetDefaultFileServerID
GetFileServerID

GetInternetAddress
GetLANAddress
GetMaximumNumberOfStations
GetObjectConnectionNumbers
GetStationAddress
GetUserNameFromNetAddress
LoginToFileServer
Logout
LogoutFromFileServer
NWGetSecurityLevel
NWSetSecurityLevel
SetConnectionCriticalErrorHandler

Data Migration Services

NWGetDataMigratorInfo
NWGetDefaultSupportModule
NWGetDMFileInfo
NWGetDMVolumeInfo
NWGetSupportModuleInfo
NWIsDataMigrationAllowed
NWMoveFileFromDM
NWMoveFileToDM
NWPeekFileData
NWSetDefaultSupportModule

Extended Attribute Services

CloseEA
CopyEA
EnumerateEA
GetEAInfo
OpenEA
ReadEA
WriteEA

File System Services

AddSpaceRestrictionForDirectory
AddTrustee
AddUserSpaceRestriction
ChangeDirectoryEntry
DeleteTrustee
DeleteUserSpaceRestriction
GetAvailableUserDiskSpace

GetAvailableUserDiskSpace
GetDiskSpaceUsedByObject
GetEffectiveRights
GetMaximumUserSpaceRestriction
GetNumberOfVolumes
GetVolumeInformation
GetVolumeInfoWithNumber
GetVolumeName
GetVolumeNumber
GetVolumeStatistics
ModifyInheritedRightsMask
PurgeTrusteeFromVolume
ReturnSpaceRestrictionForDirectory
ScanTrustees
ScanUserSpaceRestrictions
SetDirectoryInfo
UpdateDirectoryEntry

Message Services

BroadcastToConsole
DisableStationBroadcastss
EnableStationBroadcasts
GetBroadcastMessage
SendBroadcastMessage

Miscellaneous Services

GetNetworkSerialNumber
VerifyNetworkSerialNumber

Queue Management Services

AbortServicingQueueJobAndFile
AttachQueueServerToQueue
ChangeQueueJobEntry
ChangeQueueJobPosition
ChangeToClientRights
CloseFileAndAbortQueueJob
CloseFileAndStartQueueJob
CreateAQueue
CreateQueueJobAndFile
DestroyQueue
DetachQueueuServerFromQueue
FinishServicingQueueJobAndFile

GetQueueJobList
GetQueueJobsFileSize
NWQAbortJob
NWQAbortJobService
NWQBeginJobService
NWQChangeJobEntry
NWQChangeJobPosition
NWQChangeJobQueue
NWQChangeToClientRights
NWQCreate
NWQCreateJob
NWQDestroy
NWQDetachServer
NWQEndJobService
NWQGetJobEntry
NWQGetJobFileSize
NWQGetServers
NWQGetServerStatus
NWQGetStatus
NWQMarkJobForService
NWQRemoveJob
NWQRestoreServerRights
NWQScanJobNums
NWQServiceJob
NWQSetServerStatus
NWQSetStatus
ReadQueueCurrentStatus
ReadQueueJobEntry
ReadQueueServerCurrentStatus
RemoveJobFromQueue
RestoreQueueServerRights
ServiceQueueJobAndOpenFile
SetQueueCurrentStatus
SetQueueServerCurrentStatus

Server Environment Services

CheckConsolePrivileges
CheckNetWareVersion
ClearConnectionNumber
DisableFileServerLogin
DisableTransactionTracking
DownFileServer
EnableFileServerLogin
EnableTransactionTracking

GetBinderyObjectDiskSpaceLeft
GetConnectionSemaphores
GetConnectionsOpenFiles
GetConnectionsTaskInformation
GetConnectionsUsageStats
GetConnectionsUsingFile
GetDiskCacheStats
GetDiskChannelStats
GetDiskUtilization
GetDriveMappingTable
GetFileServerDateAndTime
GetFileServerDescriptionStrings
GetFileServerLANIOStats
GetFileServerLoginStatus
GetFileServerMiscInformation
GetFileServerName
GetFileSystemStats
GetLANDriverConfigInfo
GetLogicalRecordInformation
GetLogicalRecordsByConnection
GetPathFromDirectoryEntry
GetPhysicalDiskStats
GetPhysicalRecordLocksByFile
GetPhysRecLockByConnectAndFile
GetSemaphoreInformation
GetServerInformation
GetServerMemorySize
GetServerUtilization
SSGetActiveConnListByType
SSGetActiveLANBoardList
SSGetActiveProtocolStacks
SSGetCacheInfo
SSGetCPUInfo
SSGetDirCacheInfo
SSGetFileServerInfo
SSGetGarbageCollectionInfo
SSGetIPXSPXInfo
SSGetKnownNetworksInfo
SSGetKnownServersInfo
SSGetLANCommonCounters
SSGetLANConfiguration
SSGetLANCustomCounters
SSGetLoadedMediaNumberList
SSGetLSLInfo

SSGetLSLLogicalBoardStats
SSGetMediaManagerObjChildList
SSGetMediaManagerObjInfo
SSGetMediaManagerObjList
SSGetMediaNameByNumber
SSGetNetRouterInfo
SSGetNetworkRoutersInfo
SSGetNLMInfo
SSGetNLMLoadedList
SSGetNLMResourceTagList
SSGetOSVersionInfo
SSGetPacketBurstInfo
SSGetProtocolConfiguration
SSGetProtocolCustomInfo
SSGetProtocolNumbersByLANBoard
SSGetProtocolNumbersByMedia
SSGetProtocolStatistics
SSGetRouterAndSAPInfo
SSGetServerInfo
SSGetServerSourcesInfo
SSGetUserInfo
SSGetVolumeSegmentList
SSGetVolumeSwitchInfo
SendConsoleBroadcast
SetFileServerDate AndTime
TTSGetStats

Synchronization Services

ClearFile
ClearFileSet
ClearLogicalRecord
ClearLogicalRecordSet
ClearPhysicalRecord
ClearPhysicalRecordSet
CloseSemaphore
ExamineSemaphore
LockFileSet
LockLogicalRecordSet
LockPhysicalRecordSet
LogFile
LogLogicalRecord
LogPhysicalRecord
OpenSemaphore
ReleaseFile

ReleaseFileSet
ReleaseLogicalRecord
ReleaseLogicalRecordSet
ReleasePhysicalRecord
ReleasePhysicalRecordSet
SignalSemaphore
WaitOnSemaphore

TTS Services

TTSAbortTransaction
TTSBeginTransaction
TTSEndTransaction
TTSGetApplicationThresholds
TTSGetWorkstationThresholds
TTSIsAvailable
TTSSetApplicationThresholds
TTSSetWorkstationThresholds
TTSTransactionStatus

Parent Topic:

Renamed NLM Functions

NLM Coding Issues

Besides the issues of the NetWare® OS, the programmer needs to understand the structure of NLM™ applications as well as services that are provided by the NetWare API. The following gives an overview of the features that should be used when writing code for NLM applications.

Threads

Thread Groups

Multithreaded Programming

Context

Structure of an NLM

NLM Startup

Relinquishing Control

Critical Sections

NLM Synchronization

Shared Memory

NLM Screen Handling

Freeing Resources upon Exit

Termination Process

NLM Screen Handling

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

Related Topics:

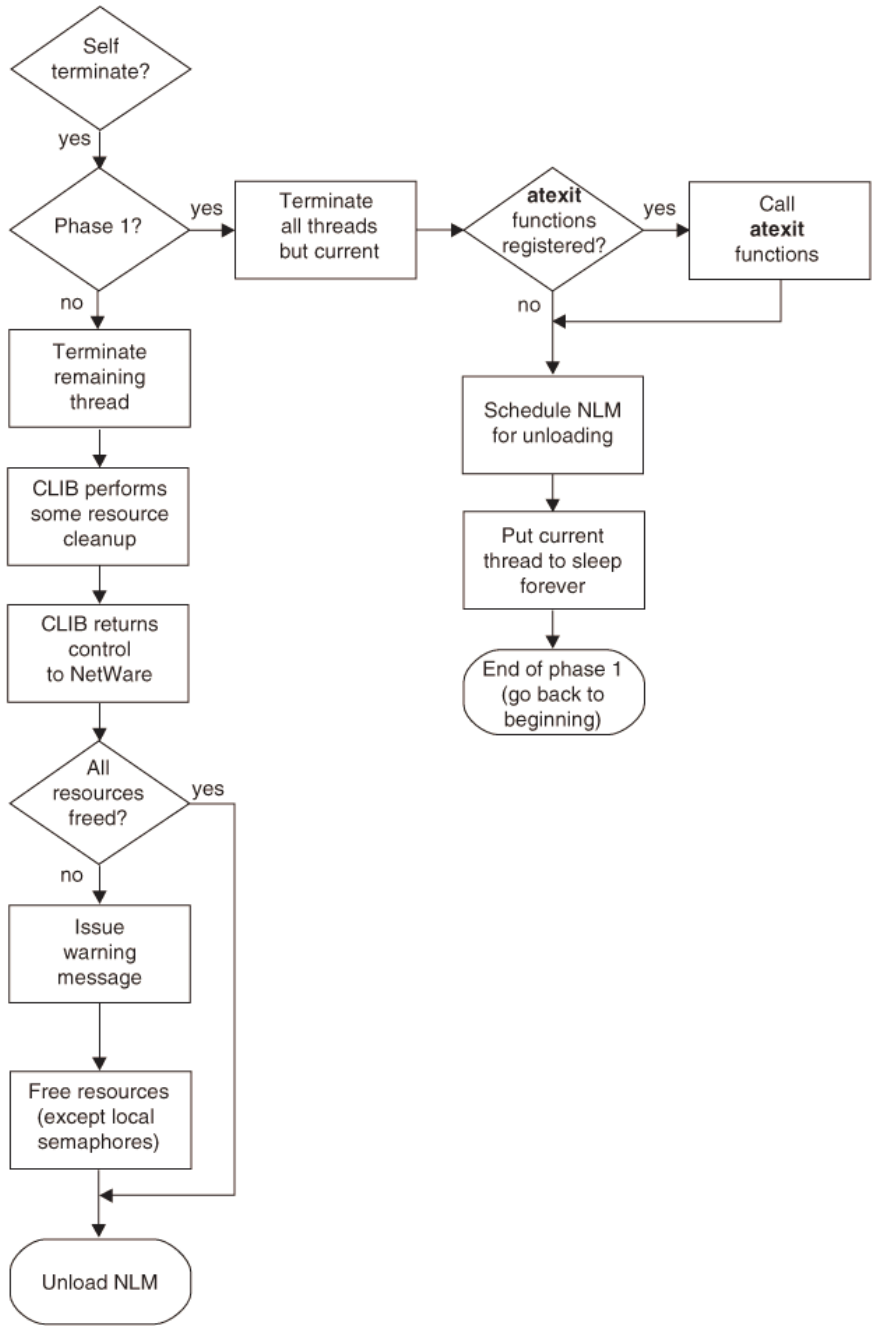
Screen Creation

Input and Output Cursors

Screen Deletion

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.



NLM Startup

When the NLM™ application first loads, some initialization must be done,

usually with the **_Prelude** function that is called automatically when the NLM is loaded. The **_Prelude** function preforms the following initialization steps:

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.

The **_Prelude** function is part of PRELUDE.OBJ or NWPRE.OBJ, one of which must be linked in with the OBJ files for the NLM. A discussion of the differences between these two objects follows.

Related Topics:

PRELUDE.OBJ or NWPRE.OBJ

Reentrant NLM Applications

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.

Related Topics::

Locking

Semaphores

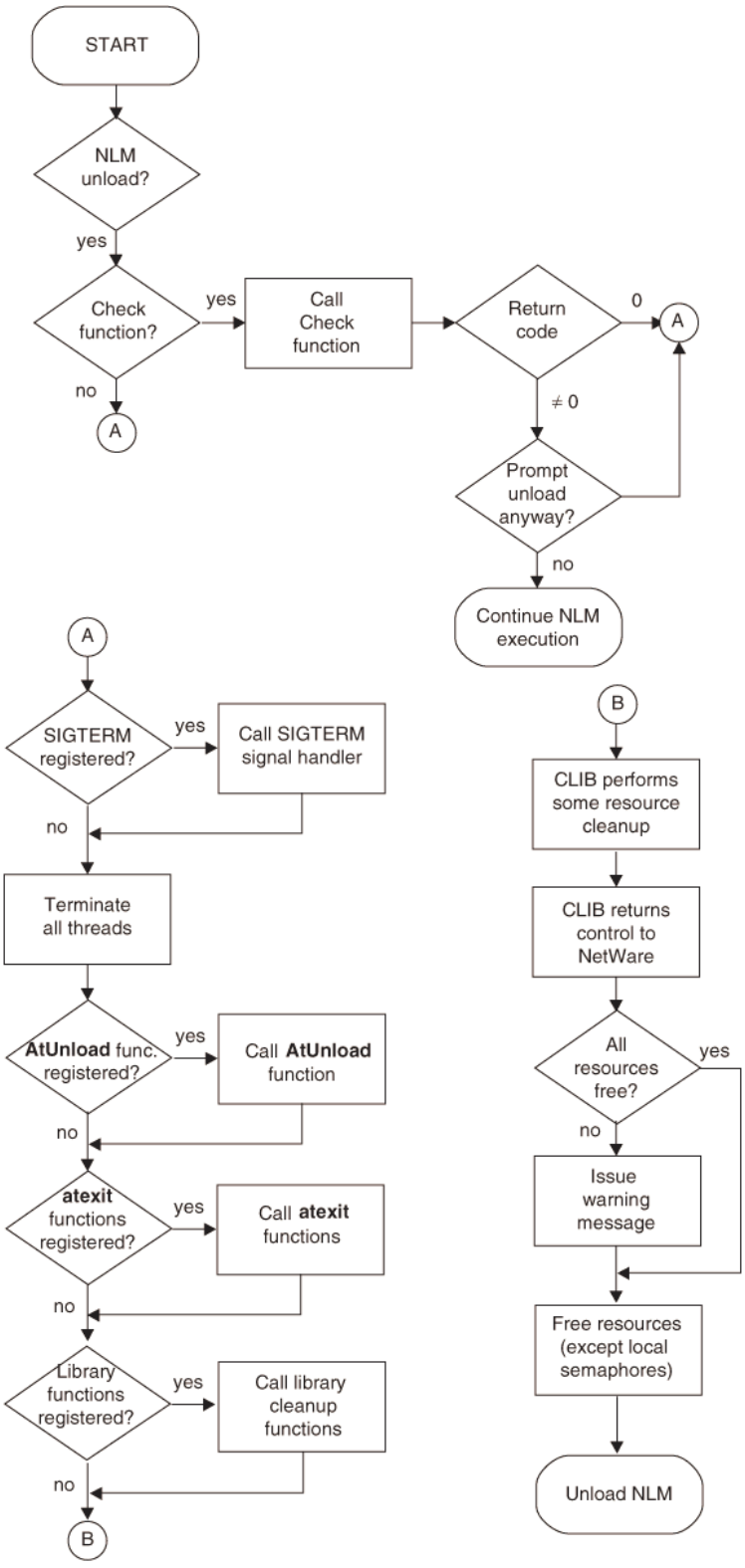
Synchronization

Parent Topic:

NLM Coding Issues

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.


START
NLM unload?
yes
Check function?
yes
Call Check function
Return code
0
A
≠ 0
Prompt unload anyway?
no
Continue NLM execution
no
A
A
SIGTERM registered?
yes
Call SIGTERM signal handler
no
Terminate all threads
AtUnload func. registered?
yes
Call AtUnload function
no
atexit functions registered?
yes
Call atexit functions
no
Library functions registered?
yes
Call library cleanup functions
no
B
B
CLIB performs some resource cleanup
CLIB returns control to NetWare
All resources free?
yes
no
Issue warning message
Free resources (except local semaphores)
Unload NLM

Nonpreemptive Environment

The NetWare® OS is designed specifically as a network server OS. A server OS works most efficiently if it does not perform time slicing (preemption). Because the nonpreemptive scheduler has fewer cases of task switching, the NetWare OS achieves superior performance. This nonpreemptive environment rewards the developer with applications that run faster, perform more efficiently, and are easier to develop.

With access to shared memory being a common occurrence in a NetWare environment, nonpreemption puts threads (processes) in the offensive position of being able to ensure, without mandatory locking, that their shared data remains coherent. This assurance is the result of a thread being able to run until it knowingly calls a function that blocks (relinquishes control to the OS). The advantage of nonpreemption is that all threads run quicker, and there is less overhead when switching operations between threads.

Because the OS waits for threads to block, NLM™ applications operate in a "nice guy" environment, where they 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 (from 3,000 to 10,000 machine instructions) and then relinquish control.

> **NOTE:** Because you determine when your the threads in your NLM block, you might be tempted to have your NLM retain control of the CPU for long periods of time in order to speed it up. To do so is a mistake because it violates the "nice guy" environment.

Obsolete CLIB Headers

The following headers files from the unified, "monolithic" CLIB have been obsoleted, with no modular CLIB equivalent:

cdecl.h
newin400.h

Operation of the Clock

During NetWare® initialization, the hardware clock on the host system is

read and the UTC clock set. The value of *tickIncrement* is also determined at this time. The influence of time zone and daylight savings time information on time calculations is discussed in Interactions between Local Time, UTC, and Other Variables. The following illustrates what happens during each timer interrupt (clock tick):

```
clock = clock + tickIncrement;
if (adjustmentCount > 1)
{
   clock = clock + adjustmentValue;
   adjustmentCount = adjustmentCount - 1;
}
else if (adjustmentCount == 1)
{
   clock = clock + grossCorrection;
   adjustmentCount = adjustmentCount - 1;
}
```

After the UTC clock is updated, a second clock structure holding local time is updated at the same rate. Local time calculations are affected by the clock adjustments just as UTC is.

This algorithm was chosen with the task of time synchronization in mind. The *adjustmentValue* determines how much gradual correction is applied to the clock during each timer interrupt and the *grossCorrection* holds a remainder. Thus, if a clock error of *ERR* seconds is to be corrected over a period of *N* clock interrupts, the following calculations provide appropriate values (ignoring potential problems such as division by zero):

adjustmentCount = N

adjustmentValue = ERR/(N-1)

grossCorrection = ERR mod (N-1)

Note that the *grossCorrection* and *adjustmentValue* can be positive or negative. In practice, a further restriction should be enforced because time should not move backwards: when negative, the *adjustmentValue* and *grossCorrection* should be strictly smaller in magnitude than the tick increment. That is, the clock should never tick backwards. A primary use of UTC time is as a timestamp for ordering events. As such, UTC should increase monotonically.

> **CAUTION: Many applications use timestamps to sort events and rely heavily on the fact that time does not move backwards. Particularly, NDS™ operations rely on timestamps derived from UTC time. Almost any application that uses a timestamp is adversely affected if time appears to move backwards. For that reason, even setting the time backwards on an operating server to correct an obvious error can have drastic consequences and should be avoided when possible. It is important that time be set correctly when the server boots, or adjusted as quickly as possible thereafter so that applications are not adversely affected by backwards corrections.**

The difference between setting the time directly and applying one or more

adjustments to the clock is that applying adjustments does not generate a time change event, which is detectable through the event notification interface. The adjustment mechanism is intended to allow a way to correct small time errors gradually so that they are not noticed. It is possible to apply very large adjustments, but it might not be wise to do so.

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.

Related Topics:

Nonpreemptive Environment

Protected or Nonprotected Environment

Current Working Directory

Connection and Task Numbers

Screens and the NetWare OS

Parameter Passing

When writing assembly procedures to be called by C code, or when calling C functions from assembly code, you must understand how parameter passing is accomplished. The following discusses stack-based parameter passing.

C Parameter Ordering

Parameter Size

Parameter Processing by Assembly Procedures

When an assembly procedure is called by C, the assembly procedure finds its parameters on the stack in the order shown in the figure inC Parameter Ordering. After saving the necessary registers, the assembly procedure can address these values on the stack. For example, the procedure **MyProc** could be coded as follows:

```
Proc MyProc
      push ebp
      mov ebp,esp
      ...
```

```
        mov eax, [ebp+8]; "A"
        mov ecx, [ebp+12]; "B"
        mov edx, [ebp+16]; "C"
        ...
        mov esp,ebp
        pop ebp
    RET
```

Parameter Size

When examining the figure in C Parameter Ordering, you will notice the parameters A, B, and C, are all pushed on the stack as long (32-bit) values, to keep the stack long aligned. C does this automatically when it passes parameters; your assembly code must do the same.

When calling C functions from assembly, you should never push a BYTE as a parameter on the stack; instead, you must first convert it to a long. One way to convert it to a long is shown in the following example.

```
p   ;to push a BYTE on the stack as a 32-bit value
    XOR EAX, EAX
    MOV AL, ByteValue
    PUSH EAX
    ;or
    MOVZX EAX, ByteValue
    PUSH EAX
```

Passing Parameters to C Code

When assembly code makes a call to a C function, it must provide the parameters to the C function in the order that the C function expects. That is, it must place the parameters on the stack just as C would.

For example, when assembly code calls the C function, **MyFunc**, which has the parameters A, B and C. The assembly code must issue the following commands:

```
PUSH C
    PUSH B
    PUSH A
    CALL MyFunc
```

NOTE: There is no need for the assembly code to save the EBX, ESI, EDI, or EBP registers because C functions preserve these registers.

PRELUDE.OBJ or NWPRE.OBJ

Whether to link PRELUDE.OBJ or NWPRE.OBJ depends on the need for backward compatibility measured against the need for ANSI or POSIX compliance. Do not attempt to link to both objects.

Link to PRELUDE.OBJ for full backward compatibility with versions of CLIB.NLM previous to version 4.11. Even with the use of CLIB version 4.11 or later, if there is a need to run with the old CLIB binaries, this is the object to link to.

Link to NWPRE.OBJ to get full ANSI and POSIX compliance. Although this object does not provide full backward compatibility without explicitly breaking ANSI compliance, it offers some important improvements:

Improved UNIX* like command line for redirection is enabled (although the old CLIB_OPT switches still work. The redirection characters, which are set on the system console commnad line as they would be under DOS or UNIX, are as follows:

`< filename`---redirect *stdin* from filename

`> filename`---redirect *stdout* to filename

`2< filename`---redirect *stderr* to filename

Implementation of return values for **fputs** is corrected according to ANSI. Watcom* previously returned -1 for failure and 0 for success. It now returns the number of characters put on success.

An update correction of the *tm_isdst* field in the tm structure passed by the **mktime** function is included.

The object provides more silent behavior for math errors when no handler has been registered. If no error handler has been registered with **RegisterMatherrHandler** and an error occurs, the function returns -1 and *errno* is set.

The accuracy in the values of global variables *daylight*, *daylightOffset*, *daylightOnOff*, *altzone*, *timezone*, and *tzname* in multithreaded applications is improved. The values of these global variables remain loyal to the thread that set them, although threads still overwrite each other's values until all application source code using these global variables has been recompiled.

The buffer passed to **fgets** is left untouched of the return is set to nil. Previously Watcom put NULL in the first character.

A new distinction between pronouns **P** and **p**, and **R** and **r** in **strftime** is enabled. The uppercase pronouns generate **AM** and **PM**, and the lowercase pronouns generate **am** and **pm**.

The old global variable *_IsTime* from ctype.h exists and is supported, but old functions and newly-compiled macros in applications now use *__ctype*.

Primary and Secondary Server

When two NetWare SFT III™ servers are running as a mirrored pair, one server is the **primary** server and the other server is the **secondary** server. Initially, the primary server is the server on which the operator issued the **ACTIVATE SERVER** command. However, if the (initial) primary server suffers a fault, the secondary server takes over as the primary server. When the former primary server comes back up, it becomes the secondary server.

The primary server of a mirrored pair always controls the sequence of events that are processed by the MSEngine. Network clients always send their request packets to the primary server, and always receive their response packets from the primary server. The secondary server mirrors the state of the primary server.

Programming Languages

You can write NLM™ applications in either C language or assembly language. You do not need to learn new coding conventions or a new language standard to begin writing NLM applications. Because the NetWare® OS runs in protected mode on 80386 and 80486 computers, you should use a compiler or assembler that generates 32-bit code. For more information about compilers, see NLM Compilers.

Related Topics:

NLM Assembly Interface on the Intel Platform

Protected or Nonprotected Environment

In the NetWare® 3.x environment, NLM™ applications run at ring 0 for faster execution. This does not offer memory protection, however, and ill-behaved NLM applications can access memory that is not theirs. While developing NLM applications, you can add a limited amout of memory protection to the server using PROTECT.NLM, which catches many illegal memory accesses.

You can use PROTECT.NLM while developing your NLM applications, but end users do not run PROTECT.NLM.

A new feature of the NetWare 4.x OS is the option to run with or without memory protection. The NetWare 4.x OS provides two domains of protection: OS and OS_PROTECTED. The OS domain is always present in the OS. The OS_PROTECTED domain is created when DOMAIN.NLM is loaded. There is a tradeoff here: modules running in the OS_PROTECTED domain have the overhead of ring transition code when accessing OS calls. Modules in the OS domain run faster because they do not have the ring transition overhead, but the OS is not protected from them.

All NetWare 4.x NLM applications should be tested in the OS_PROTECTED domain while they are being developed because this domain catches more addressing errors. You might want to test all of your NetWare 3.x applications on NetWare 4.x servers because DOMAIN.NLM can catch more errors than PROTECT.NLM.

Unlike PROTECT.NLM, which is strictly a development tool, end users can run DOMAIN on their servers.

QMS for Client-Server Communication

Using the Queue Management System (QMS), you can take advantage of the queuing features inherent in the NetWare® OS. This frees the server from the responsibility of queuing and allows it to process requests more efficiently.

When clients issue service requests, QMS places the requests in a queue and the server services them. Multiple servers can access a single queue to process requests. Likewise, a single server can handle multiple queues.

QMS uses the features of the Directory or the bindery to represent queues and to define their characteristics. Each queue is represented in the bindery and in the Directory as an object.

Client and server applications manipulate the queues using QMS functions. QMS functions form a complete set of tools for working with the bindery aspect of NetWare jobs. If an application needs more control in defining the structure of jobs, it can use QMS functions exclusively to supply services.

Generally, QMS works best for applications that:

Process large units of work

Large jobs and batch jobs, such as print spooling or sending data by modem, can benefit from QMS.

Do not require an immediate response

Queues process requests on a FIFO basis. Therefore, the promptness of the service varies with the size of the queue.

Need flexibility and control

QMS can sort and manage jobs by properties such as job size and priority. In addition, QMS offers a secure and dependable operating environment.

Reentrant NLM Applications

A reentrant NLM™ application 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 NLM applications call the startup function **_Prelude** each time they are loaded. Reentrant NLM applications, on the other hand, only call **_Prelude** 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 statup 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.

RENTRANT.C is a working example of a reentrant NLM. It is located in the EXAMPLES directory. Code fragments from RENTRANT.C are used in the following example.

The startup function for RENTRANT.C is **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 OS displays the message "`Attempt to reinitialize reentrant module FAILED`" even if the NLM successfully loads.

> **NOTE:** When an NLM is loaded, its startup thread is an OS 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 OS 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 directive file (.LNK), use the REENTRANT option (withWLINK and NLMLINK) to specifiy that the NLM is reentrant. Use theSTART option to specify the function you want to use as the startup function. The following is an example of a WLINK directive 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 NLM Linkers.

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.)

Relinquishing Control

Because the NetWare® OS does not timeslice or preempt thread execution, the responsibility of relinquishing control falls to the thread itself. To relinquish control of the processor, a thread can do one of the following:

Call a function that can relinquish control

For example, if a thread calls **printf**, it can relinquish control because **printf** writes to a device. However, this method should not be used in a program that must be guaranteed that control is relinquished.

Functions that might block are identified in the function descriptions.

Call **ThreadSwitch**

ThreadSwitch passes control of the CPU to the OS, which then passes control to the next thread in the run queue. The calling thread is placed at the end of the run queue.

Call **delay** or **ThreadSwitchWithDelay**

These functions suspend thread execution for a specified time (in milliseconds). **ThreadSwitchWithDelay** is new for the NetWare 4.x OS, but it has been added to the 3.11 version of CLIB.

> **CAUTION: Threads that do busy waiting or spin locks in NetWare 4.x need to allow low priority threads to run. For this reason, these threads should call ThreadSwitchWithDelay, instead of ThreadSwitch. Low priority threads can only run when there are no threads waiting on the RunList, and ThreadSwitch places the threads that call it on the RunLIst. ThreadSwitchWithDelay places threads that call it on the DelayedList.**

Call **EnterCritSec**

The **EnterCritSec** function prevents all other threads in the NLM™ application from running but it does not stop threads in other NLM applications.

> **NOTE:** If a new thread is started while the NLM is in a critical section, the thread will not be in the critical section.

Call **SuspendThread**

The **SuspendThread** function puts a thread to sleep until it is awakened.

> **NOTE:** A sleeping thread can be awakened only by calling **ResumeThread** from another thread.

Call **ThreadSwitchLowPriority**

The **ThreadSwitchLowPriority** function suspends thread execution and places the thread in the Low-Priority Queue. This function is new for the NetWare 4.x OS.

Wait on an event

The OS automatically puts to sleep any threads waiting on events. For example, if a thread waits in a semaphore queue, it relinquishes control.

> **CAUTION: Do not use this method when waiting to read a file from a disk. If the file is stored in cache memory, the thread does not have to wait and does not relinquish control.**

One side effect of failing to relinquish control is that incoming client requests are still received by the server, but the packets cannot be processed. Thus, without acknowledgment of the request, the client connection eventually times out.

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. For example:

Blocking Information: This function blocks if called in blocking mode.

Renamed Functions in Connection Number and Task Management Services

The functions listed in the following table were renamed to reduce confusion.

Old Name	New Name
GetCurrentConnectionID	GetCurrentFileServerID
SetCurrentConnectionID	SetCurrentFileServerID

Renamed Functions in Connection Services

The functions listed in the following table were renamed to reduce confusion.

Old Name	New Name
GetDefaultConnectionID	GetDefaultFileServerID
SetDefaultConnectionID	SetDefaultFileServerID

Renamed Functions in Screen Handling Services

MapScreenIDToHandle was renamed **GetScreenInfo** because the new name was less confusing.

Renaming VAP Functions

If function names in your VAP differ from those you will use in your NLM™ application, you can rename (map) the functions used in your VAP. For example, if you use the **ConsoleDisplay** function in your VAP, you can add the following line to the NLM to call the **ConsolePrintf** function:

```
#define ConsoleDisplay ConsolePrintf
```

Many functions are more efficient and more flexible in an NLM than in a VAP. For example, the **ConsolePrintf** function in NLM applications allows for embedded strings that define variables, an option not allowed in VAPs. The following code example is not valid in VAPs but is allowed in NLM applications.

```
ConsolePrintf(You are user %s, logged in to %s.\n",
  login_name, fileserver_name);
```

If you are mapping a VAP function for use in an NLM, see the function references for specific information on syntax, return values, data structures, and so on.

Screen Creation

Screens are created using **CreateScreen**, 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**. 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.

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**. Screens can also be destroyed within the NLM by calling **DestroyScreen**.

For more information about screens and how to write to them, seeScreen Handling.

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 NLM™ applications are loaded.

NLM applications 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**.

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.

The NetWare® 3.x OS does not limit the number of screens than an NLM can have open. The NetWare 4.x OS limits the number of screens to 100.

An example of creating and using multiple screens is presented in Multithreaded Programming.

Segment Registers

Your NLM™ applications should not change the CS, DS, ES, FS, GS, or SS segment registers. These registers are used for memory protection, and changing them causes unpredictable results.

Semaphores

Whereas locking allows only one thread to access a file-related resource, 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.

Local semaphores

These apply to resources available only to a single server. For more information on local semaphores, see Interprocess Synchronization.

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

NOTE: You can use local semaphores among multiple NLM applications; 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.

> **CAUTION: 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.**

Server Connections and VAP Conversion

Both VAPs and NLM™ applications use connections to communicate with the NetWare® OS. However, NLM applications can receive file service without logging in to the server, whereas VAPs must log in as an object.

When a VAP communicates with the server, it logs in as a user and acquires a connection number as if it were a user. In some previous versions of NetWare, the number of connections available to client workstations could be significantly reduced if many VAPs were installed on the server.

In contrast, NLM applications are not required to attach (or log in) to the local server in order to access the NetWare OS. NLM applications can directly access NetWare file services to perform file I/O, bindery operations, transactions, and so on.

Whereas a VAP must use a flag in the VAP header to indicate that it needs a connection, an NLM can establish connections as needed. NLM applications can call **SetCurrentConnection**(0) to bypass the typical user login sequence and gain Supervisor rights to the file system.

NLM applications can log in to a server if necessary.The NetWare 3.11 OS reserves 100 "hidden" connection numbers for NLM applications. These numbers begin after the last client connection number. For instance, in a 250-user system, NLM connection numbers run from 251 to 350. Likewise, in a 100-user system, the NLM connection numbers are 101 to 200. The NetWare 4.x OS does not limit the number of connections for NLM applications.

For more information on connections, seeConnection Number and Task Management and Connection.

SFT III and Device Drivers

NetWare SFT III™ does not depart from the standard NetWare device driver framework. For each type of device supported by NetWare (LAN, disk, or other), NetWare implements a support layer that separates logical structures such as file systems or transport protocols from physical device drivers such as ethernet drivers or SCSI drivers.

Compare the NetWare Support Layer Architecture with the NetWare SFT III Support Layer Architecture.

SFT III Application Design Issues

NetWare SFT III™ presents important opportunities to server application developers. Generally, developing applications for NetWare SFT III is not unlike developing applications for standard NetWare. However, some areas of development require special attention.

NLM developers using the NetWare API must address some additional issues, which are discussed in The NetWare API and SFT III.

The following types of applications must be considered:

Mirrored Applications

IOEngine Applications

MSEngine Applications

SFT III Considerations for Time Synchronization

Although the I/O engine of an SFT III™ server generally isolates hardware dependencies, it does not isolate the timer interrupt. Therefore, both engines service the timer interrupts independently. However, **SetSyncClockFields** (see The Synchronized Clock Interface) updates the synchronized clock structures in both engines. This greatly simplifies time synchronization on SFT III servers.

Parent Topic:

Controlling the Server Clock under NetWare 4.x

SFT III LAN Configuration

The primary difference in LAN configuration between standard NetWare® and NetWare SFT III™ is that both the IOEngine and the MSEngine maintain internal IPX™ networks.

The internal IPX network provides a stable destination address for client request packets (and conversely, a stable source address for server reply packets). The server's physical LANs can come and go dynamically as the operator loads and unloads LAN drivers; however, the server's internal IPX LAN remains operational for as long as the server itself is operating.

When you load the NetWare SFT III IOEngine, you must assign to it an IPX

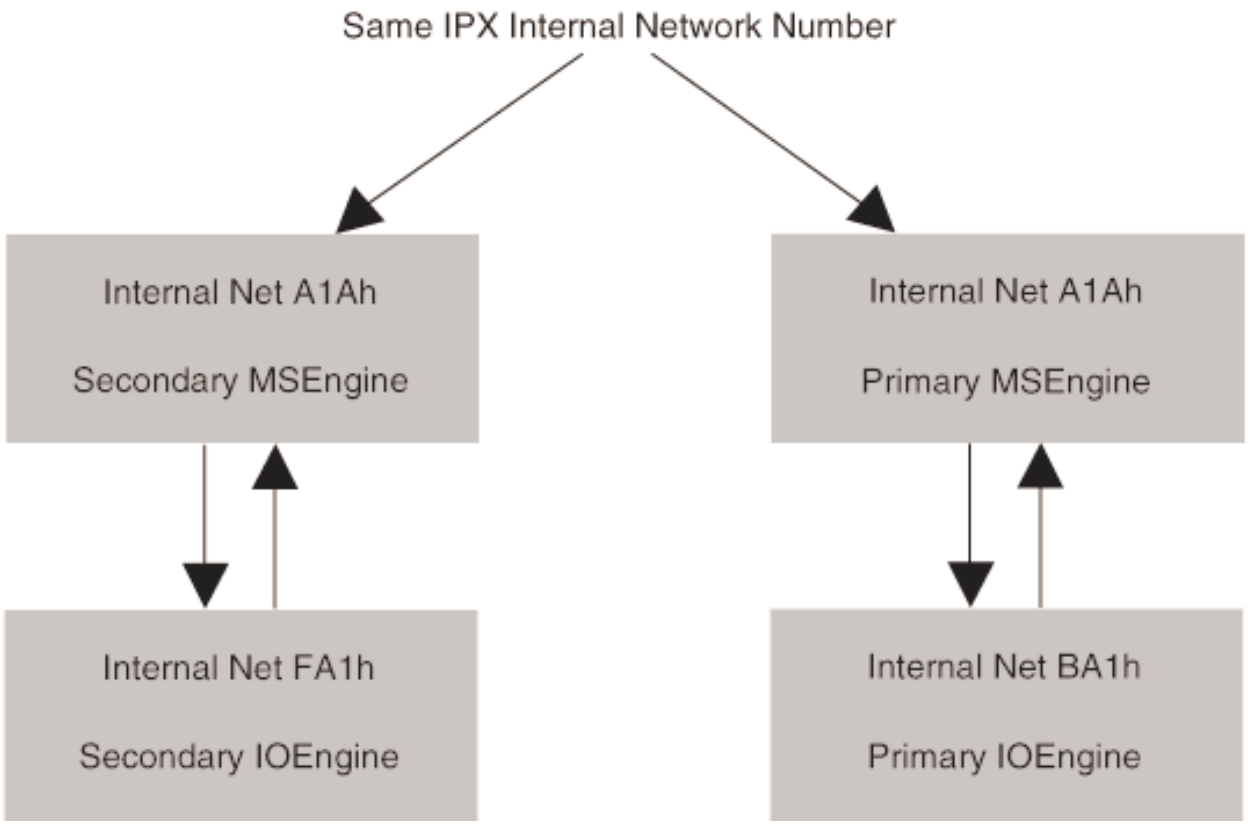
internal network number. This number remains the ultimate source and destination address for packets sent to and from the IOEngine for as long as the IOEngine is running.

The IOEngines of two mirrored SFT III servers must have **unique** IOEngine internal network numbers. (Just because they are mirrored partners does not mean their IOEngines can have the same IPX internal network number.) In fact, IOEngine internal network numbers must be completely unique on an internetwork.

The MSEngine internal network number must be different from both IOEngine internal network numbers, and also unique to the entire internetwork. However, because there is only one **logical** MSEngine, both servers of a mirrored pair run the MSEngine using the **same** IPX internal network number. Remember, however, that the MSEngine on the **secondary** server is invisible to network clients, and is strictly a "shadow" of the MSEngine on the primary server.

If the primary server fails, it is critical that the MSEngine on the secondary server have the same internal IPX network address. This allows network clients to continue sending request packets and receiving response packets from the **same address**.

Figure 15. SFT III LAN Configuration Figure



The LAN configuration of the IOEngine can be radically different for two servers operating as a mirrored pair. This is because IOEngine network addresses are never the ultimate source or destination address of request and reply packets. Network clients view IOEngine network addresses as part of the route to the MSEngine internal network, which is the ultimate source and destination of reply and request packets.

All physical LAN segments attached to a SFT III server are part of the IOEngine LAN configuration---they are **never** part of the MSEngine LAN configuration. This condition exists because only the IOEngine can load device drivers.

The IOEngine, because it contains all hardware-specific data structures, is never mirrored between two SFT III servers. Consequently, when a network board (or cabling) fails, that network segment disappears. Therefore, it is important that systems designers configure LAN segments in a redundant manner, providing network clients with multiple routes to the SFT III MSEngine IPX internal network address.

A redundant LAN configuration is similar in principal to the NetWare drive duplexing logic. When a drive adapter fails, that drive "disappears." However, the NetWare drive addressing logic allows the OS to perform drive I/O using an "alternate route" (in this case, a redundant drive and controller). Similarly, when a physical LAN segment attached to an SFT III server suffers a fault, clients should be able to perform LAN I/O using an alternate route.

SFT III Server Memory Management

Like standard NetWare®, NetWare SFT III™ implements a flat memory model using 32-bit memory addresses. Also as in standard NetWare, DOS can occupy SFT III server memory addresses below 1 MB.

The IOEngine occupies 2 MB of memory beginning at the end of DOS memory. When the IOEngine loads the MSEngine as the result of an **ACTIVATE SERVER** command, the MSEngine occupies the remainder of physical memory, beginning at the end of IOEngine memory. (You can limit the amount of memory occupied by the MSEngine by performing the memory alignment procedure by means of **SET** parameters in the IOEngine before the server is activated. See the *NetWare SFT III* manual for more information.)

Both the IOEngine and the MSEngine have their own memory management kernels. When both engines are active, they are able to address the same memory space. However, memory sharing between engines should occur rarely, only under certain circumstances, and according to a strict protocol. In all cases, inter-engine memory sharing is accomplished by the OS for its own purposes. **Applications should never attempt to address memory**

across engine boundaries.

Developers using documented NetWare memory allocation functions do not need to worry about allocating memory belonging to an unintended engine. However, pointer bugs can cause unintended memory addressing. As in standard NetWare, pointer errors can bring down the SFT III server.

SFT III Support in the NetWare API

Modifications to the NetWare® API for SFT III™ are relatively slight. As previously mentioned, most functions were originally coded to support NCP™ requests to remote servers as well as direct system calls to a local server. Because there is no "local" server for an NLM™ application running in the IOEngine, most of the modifications to the NetWare API involve establishing an implicit connection to the MSEngine at NLM load time. This implicit connection forces all file system-related calls to resolve to an NCP request to the MSEngine. In addition, certain functions were altered to be aware of the implicit connection to the MSEngine, including **AttachToFileServer**, **GetConnectionNumber**, **GetDefaultConnection**, **GetDefaultFileServer**, **Logout**, **LogoutFromFileServer**, **LogoutObject**, **LoginToFileServer**, **LoginObject**, **SetCurrentConnection**, and **SetCurrentFileServer**.

The implicit connection to the MSEngine is established by the startup code of the NetWare API, just before the OS calls the **main** function of the NLM (provided the MSEngine is loaded).

If the NetWare API determines that the NLM is running on an IOEngine and that the MSEngine is loaded, it establishes a connection to the MSEngine by calling **AttachToFileServer**. Next, the NetWare API logs the IOEngine in to the MSEngine by calling **LoginUser**. The implicit connection to the MSEngine (which is technically a remote server) is now established. Note that all of this occurs before the OS calls the **main** function of the NLM.

> **NOTE:** The connection established with the MSEngine is a licensed connection, not an NLM connection. Each NLM that uses the NetWare API establishes a licensed connection with the MSEngine when the NLM is loaded in the IOEngine. The connection in the IOEngine is an NLM connection.

SFT Level II

SFT™ Level II is the group of fault-tolerant features included with standard NetWare®, including Hot Fix™, TTS™, and drive duplexing services.

All SFT Level II services are essentially unchanged in SFT III™. However, the services are not active unless the MSEngine is running. The data structures and logic required for SFT Level II are part of the NetWare file

system, which is implemented in the MSEngine. As a result, applications running in the IOEngine do not benefit from SFT Level II services. This is yet another reason to design applications to run in the MSEngine whenever possible.

Shared Memory

Shared memory allows multiple threads to communicate. To share memory among threads in the same NLM™ application, use a global or static pointer to a single block of memory. The following example uses a global pointer to share memory among thread groups in the same NLM:

Using a Global Pointer to Share Memory among Thread Groups

```
int SharedMemoryFlag = 0;
char *SharedMemory;
void ThreadGroup2()
{
   while (!SharedMemoryFlag)
      ThreadSwitch();
   strcpy (SharedMemory,
      "ThreadGroup2 has accessed shared memory.");
   SharedMemoryFlag = 0;
}

main()
{
   /* Start the second thread group */
   if (BeginThreadGroup (ThreadGroup2,0,0,0) == EFAILURE)
   {
      printf ("BeginThreadGroup failed.\n");
      exit(0);
}

/* Allocate the memory to be shared. Note that SharedMemory
 * could have been defined as an array, if desired. */
   SharedMemory = malloc (100);
   if (SharedMemory == NULL)
   {
      printf ("Could not allocate memory.\n");
      exit(0);
   }

/* Store a string in the allocated memory and print it */
strcpy (SharedMemory,
   "Main ThreadGroup has accessed shared memory.");
printf ("%s\n",SharedMemory);

/* Let ThreadGroup2 know it is OK to access the memory */
SharedMemoryFlag = 1;
```

```
/* Wait for ThreadGroup2 to access the memory */
while (SharedMemoryFlag)
   ThreadSwitch();

/* Print the message stored by ThreadGroup2 */
printf ("%s\n",SharedMemory);
free (SharedMemory);
}
```

If you want to use shared memory with multiple NLM applications, write a function that passes the memory address pointer among the modules. The following example shows the first NLM setting up values to share with the second NLM:

Setting up Values to Share Memory with Another NLM

```
/*  ----------------- FIRST NLM -----------------  *
 * This NLM must be loaded first. Its .LNK file    *
 * exports the two shared values, SharedMemoryFlag *
 * and SharedMemory.                               *
 *  ---------------------------------------------- */
int SharedMemoryFlag = 0;
char *SharedMemory;

main()
{
   /* Allocate the memory to be shared. Note that
    * SharedMemory could have been defined as an array,
    * if desired. */
   SharedMemory = malloc (100);
   if (SharedMemory == NULL)
   {
      printf ("Could not allocate memory.\n");
      exit(0);
   }

   /* Store a string in the allocated memory and print it */
   strcpy (SharedMemory,
      "The main NLM has accessed shared memory.");
   printf ("%s\n",SharedMemory);

   /* Let the other NLM know it is OK to access the
      memory */
   SharedMemoryFlag = 1;

   /* Wait for the other NLM to access the memory */
   while (SharedMemoryFlag)
   ThreadSwitch();

   /* Print the message stored by the other NLM */
   printf ("%s\n",SharedMemory);
   free (SharedMemory);
```

```
}
```

The following directive file should be used to link the first NLM:

```
form novell nlm 'Example of Shared Memory between NLM applications'
name     nlm1
file     prelude,nlm1
import   @clib.imp
export   SharedMemoryFlag, SharedMemory
```

The following example shows the second NLM using the shared values set up by the first NLM:

A Second NLM Sharing Memory with the First

```
/*  ----------------- SECOND NLM -----------------  *
 * This NLM must be loaded after the first. Its     *
 * .LNK file imports the two shared values.         *
 *  ------------------------------------------------  */
extern int SharedMemoryFlag = 0;
extern char *SharedMemory;
main()
{
   while (!SharedMemoryFlag)
      ThreadSwitch();
   strcpy (SharedMemory,
      "The second NLM has accessed shared memory.");
   SharedMemoryFlag = 0;
}
```

The following directive file should be used to link the second NLM:

```
form novell nlm 'Example of Shared Memory between NLM applications'
name     nlm2
file     prelude,nlm2
import   @clib.imp,
      SharedMemoryFlag, SharedMemory
```

NOTE: For the NetWare® 4.x OS, NLM applications can only share memory with modules that are loaded in the same domain. For example, modules loaded in the OS_PROTECTED domain can share memory among themselves, but they cannot access memory of modules running in the OS domain. (Domains are not an issue for the NetWare 3.x OS.)

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**.

Signal handling allows previously registered functions to gain control of

critical shutdown processes. The following are typical signals your NLM might handle:

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.

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.

SIGTERM

This ANSI standard signal is raised only 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**.

Although NLM resources can be freed using either **AtUnload** or **atexit**, these methods do not work in some situations. For instance, local semaphore handles can be stored on the local stack of a thread, where they are hidden from external functions such as **AtUnload** and **atexit**.

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 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.)

NLM applications 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");
}
```

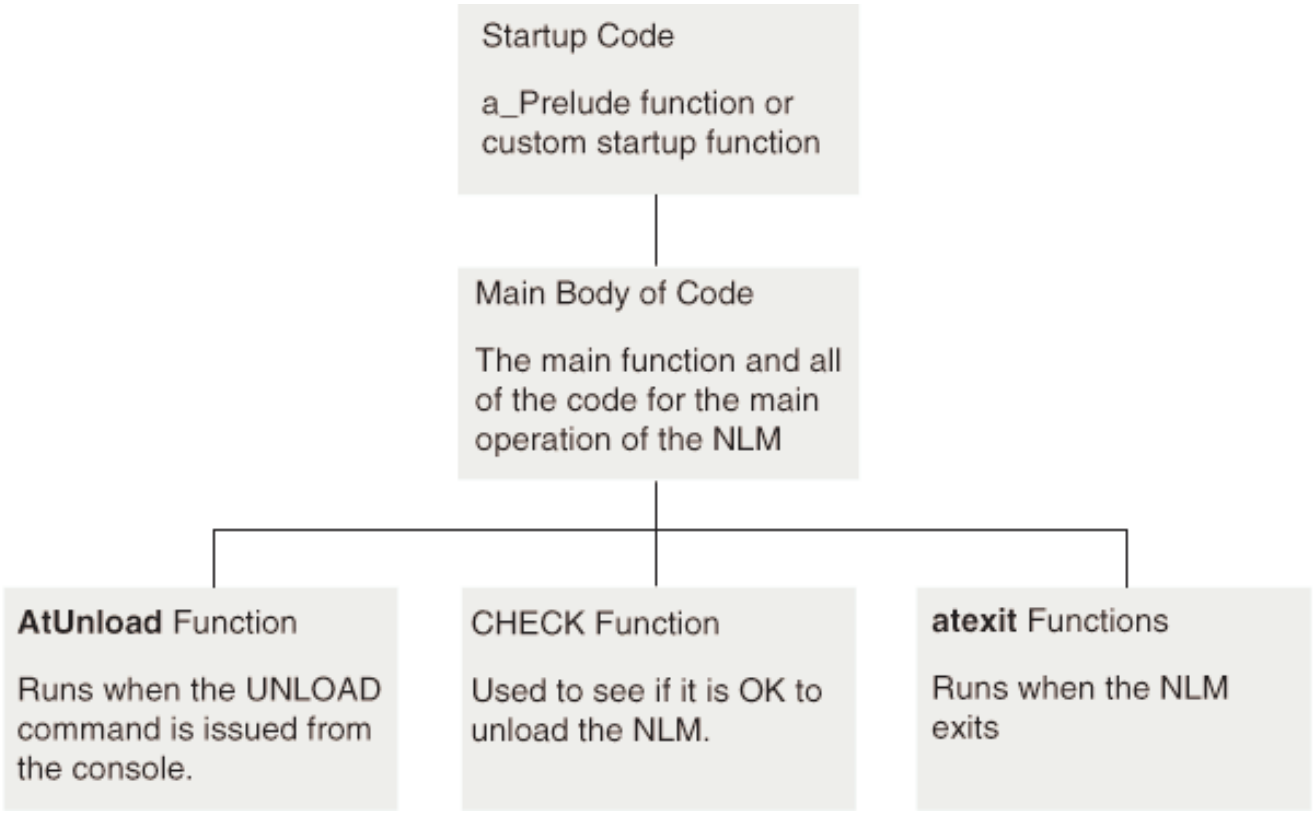
By writing a signal handler function that defines a global flag, you can manage resources that are allocated from a local stack. When the signal is raised, it can set a global flag that each thread in the NLM reads. Those threads then can return their own resources and exit.

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.

Structure of an NLM

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

Figure 16. NLM Structure



Synchronized Clock Status Flags

The following defines the constants for several flags that isolate bit fields within the *statusFlags* field:

```
#define   CLOCK_IS_SYNCHRONIZED              0x00000001L
#define   CLOCK_IS_NETWORK_SYNCHRONIZED      0x00000002L
#define   CLOCK_SYNCHRONIZATION_IS_ACTIVE    0x00000004L
#define   CLOCK_STATUS_SERVER_TYPE           0x00000F00L
```

Three status bit fields allow you to determine whether time synchronization is active on a server and the status of the synchronization effort. The synchronizing agent is responsible for managing the bits.

CLOCK_SYNCHRONIZATION_IS_ACTIVE indicates that the TIMESYNC NLM™ (or some equivalent) is active. When this bit is set, it is reasonable to expect that the CLOCK_IS_NETWORK_SYNCHRONIZED bit might eventually be set. When synchronization is active, the setting of CLOCK_IS_SYNCHRONIZED mirrors the setting of CLOCK_IS_NETWORK_SYNCHRONIZED. However, if synchronization is not active, CLOCK_IS_SYNCHRONIZED should always be set to 1. CLOCK_IS_NETWORK_SYNCHRONIZED is used by the synchronizing agent to indicate that the UTC clock is accurate.

> **NOTE:** The interaction of these bits might not be obvious. If you want a reliable network timestamp, you should first ensure that synchronization is active and watch for the CLOCK_IS_NETWORK_SYNCHRONIZED flag to be set before using the UTC clock field. If time synchronization is not active, network synchronization will not be achieved, so an application watching only CLOCK_IS_NETWORK_SYNCHRONIZED blocks. If you want an application to operate whenever time is as synchronized as possible, watch the CLOCK_IS_SYNCHRONIZED bit so that the application does not block when time synchronization is not active.

The NetWare® OS clears CLOCK_IS_NETWORK_SYNCHRONIZED whenever time-related information changes the accuracy of the UTC clock. The synchronizing agent should monitor this bit to know when the OS is signalling that corrective action might be necessary. The TIMESYNC NLM checks this bit once a second.

The CLOCK_STATUS_SERVER_TYPE field is used by the TIMESYNC NLM to report the server type as follows:

SINGLE (5)
REFERENCE (4)
PRIMARY (3)
SECONDARY (2)

Some applications can look at this field to determine the nature of the TIMESYNC NLM that is loaded. The value is not used internally in the OS, but is returned by some functions that report the clock status to clients. The value should reflect the nature of the synchronizing agent---that is, SINGLE and REFERENCE servers generally have an external time source and do not adjust their clocks to agree with PRIMARY and SECONDARY servers. PRIMARY and SECONDARY time sources adjust their clocks to agree with other time sources.

Other bits within the status flags are undefined and reserved for use by Novell®.

Termination Process

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

Self-termination, such as when the NLM exits on its own

Unload, such as when the NLM is unloaded at the system console command line, before the NLM exits on its own

Abnormal exit, such as when an error causes unexpected shutdown, such as when **abort** and **raise** are called

Related Topics:

NLM Self-Termination Process

NLM Unload Process

NLM Abnormal Exit Process

Following Exit Steps

CHECK Function

Signal Handling

AtUnload and atexit Functions

Thread Termination

The Effect of Setting the Hardware Clock Bit

In the field mask used by **GetSyncClockFields** and **SetSyncClockFields** (see The Synchronized Clock Interface) there is a SYNCCLOCK_HARDWARE_CLOCK_BIT. However, there is no hardware clock field in the synchronized clock structure. The effect of this bit differs between the two functions. This is the mechanism that the TIMESYNC NLM

uses to implement the **SET TIMESYNC HARDWARE CLOCK** parameter.

When the hardware clock bit is set, the first action of **GetSyncClockFields** is to read the hardware clock and set both local and UTC time from it. However, the subsecond counter in the UTC clock is not set to 0.

In a complementary manner, when the hardware clock bit is set, the last action of **SetSyncClockFields** is to update the hardware clock. This is accomplished by setting the local time using UTC as a base and applying the necessary time zone and DST adjustments.

The eventOffset and eventTime Fields

When UTC is greater than or equal to *eventTime*, an internal OS routine is notified and adds *eventOffset*[0] (the whole second part) to the UTC clock. This is the mechanism by which TIMESYNC schedules a time adjustment. *eventOffset*[1] is used as a status word by TIMESYNC. If TIMESYNC is not loaded, the *eventTime* and *eventOffset* fields can be used similarly.

The NetWare API and SFT III

The NetWare® API shields developers from worrying about the system-level changes between different versions of the NetWare OS. This is also true with the NetWare API and SFT III™. See the following for information about the NetWare API implementation on SFT III.

The NetWare API and the IOEngine

NetWare Functions Not Supported by the MSEngine

The NetWare API and the IOEngine

The IOEngine does not support the NetWare® file system, the bindery, NCP™ service, queue services, or any other service that relies on the logical NetWare file system or related modules. Consequently, NetWare system calls designed to work with the logical file system are not contained in the IOEngine. (Rather, they are contained in the MSEngine.)

However, the NetWare API functions, which provide services based on the logical NetWare file system, in many cases support local and remote servers. When an NetWare API function is targeted to the local server, the function resolves to one or more direct system calls. On the other hand, when the function is targeted to a remote server, the NetWare API resolves that function into one or more NCP requests to that remote server. The differences between functions targeted to local and remote servers are hidden from the NLM™ developer.

An NLM loaded in the IOEngine can call **open**, **ScanBinderyObject**, **fseek**, and so on, despite the fact that the IOEngine has no file system, provided the target file system is located on a remote server. Without support for SFT III™ within the NetWare API, NLM applications running in the IOEngine would have to target all file system-related functions to a remote server explicitly. However, this is not required, as discussed below.

When the MSEngine is loaded, it is viewed by the IOEngine as a "remote" server, despite the fact that both the IOEngine and the MSEngine are running on the same machine. Remember, the MSEngine and IOEngine have different IPX internal network numbers.

Related Topics:

API Support for NLM Applications Running in the IOEngine

SFT III Support in the NetWare API

Functions Unsupported by the IOEngine

The Synchronized Clock Interface

Two functions are provided to access the synchronized clock structure, **GetSyncClockFields** and **SetSyncClockFields**. The prototypes follow:

```
void GetSyncClockFields (LONG bitMask, Synchronized_Clock_T *);
void SetSyncClockFields (LONG bitMask, Synchronized_Clock_T *);
```

The following defines the bitmask:

```
#define   SYNCCLOCK_CLOCK_BIT              0x00000001L
#define   SYNCCLOCK_TICK_INCREMENT_BIT     0x00000002L
#define   SYNCCLOCK_ADJUSTMENT_BIT         0x00000004L
#define   SYNCCLOCK_GROSS_CORRECTION_BIT   0x00000008L
#define   SYNCCLOCK_ADUSTMENT_COUNT_BIT    0x00000010L
#define   SYNCCLOCK_STATUS_BIT             0x00000020L
#define   SYNCCLOCK_STD_TICK_BIT           0x00000040L
#define   SYNCCLOCK_EVENT_TIME_BIT         0x00000080L
#define   SYNCCLOCK_EVENT_OFFSET_BIT       0x00000100L
#define   SYNCCLOCK_HARDWARE_CLOCK_BIT     0x00000200L
#define   SYNCCLOCK_RESERVED1_BIT          0x00000400L
#define   SYNCCLOCK_DAYLIGHT_BIT           0x00000800L
#define   SYNCCLOCK_TIMEZONE_OFFSET_BIT    0x00001000L
#define   SYNCCLOCK_TZNAME_BIT             0x00002000L
#define   SYNCCLOCK_TIMEZONE_STR_BIT       0x00004000L
#define   SYNCCLOCK_DAYLIGHT_OFFSET_BIT    0x00008000L
#define   SYNCCLOCK_DAYLIGHT_ON_OFF_BIT    0x00010000L
#define   SYNCCLOCK_START_DST_BIT          0x00020000L
#define   SYNCCLOCK_STOP_DST_BIT           0x00040000L
```

These functions are not provided by the NetWare® API, but can be

imported directly from the OS. Their use does not require CLIB context, and they should be considered blocking functions. Each function expects a pointer to a clock structure and a bitmask indicating which fields are of interest. Bits in the mask must be set to transfer data to or from the OS structure, so you do not have to supply all of the information to change one field.

A NetWare API interface to the UTC clock and status fields of the synchronized clock structure is provided by **GetClockStatus**.

> **NOTE:** Debugging hint: The OS structure is named SynchronizedClock and might be visible by that name in the debugger. However, don't attempt to read or write this structure directly from your NLM™ application or you might encounter memory protection problems. Always use the Get and Set functions to prevent enormous debugging problems.
>
> **WARNING: It might appear that the interface to information for time zone, timezoneOffset, daylight savings time, and so on, is provided. This is not the case. Although information about current settings is stored in the synchronized clock structure and used in some calculations, the interface for setting these parameters is inside the NetWare OS. Attempting to change anything but the fields controlling the clock and tick rate can result in unpredictable behavior.**

To use the interface, it is important to understand how the status flags are interpreted and what the effects of setting various fields are.

Related Topics:

Synchronized Clock Status Flags

Clock Control Fields

The eventOffset and eventTime Fields

Daylight Savings Time Information Fields

The Synchronized Clock Structure

Following is an internal NetWare® structure commonly called the Synchronized Clock. Access to this structure is provided by **GetSyncClockFields** and **SetSyncClockFields** (see The Synchronized Clock Interface).

```
typedef struct   Synchronized_Clock
{
   LONG   clock[2];       /* [0]=whole seconds, [1]=fractional - UTC*/
   LONG   statusFlags;    /* bit fields as defined below */
   LONG   adjustmentCount;
   long   adjustmentValue[2];   /* [0]=whole seconds, [1]=fractional*/
```

```
    long   grossCorrection[2];   /* [0]=whole seconds, [1]=fractional*/
    long   tickIncrement[2];     /* [0]=whole seconds, [1]=fractional*/
    LONG   stdTickIncrement[2];  /* [0]=whole seconds, [1]=fractional*/
    long   eventOffset[2];       /* [0]=whole seconds, [1]=TimeSync flag
    LONG   eventTime;            /* whole seconds only*/
    LONG   daylight;             /* 0 if DST name was not in timezone in
    long   timezoneOffset;       /* seconds to UTC; LocalTime+timezone=U
    long   tzname[2];            /* offset to normal and daylight names
                                    timezoneString */
    char   timezoneString[80];
    long   daylightOffset;       /* seconds of additional offset during
                                    Daylight Savings Time*/
    long   daylightOnOff;        /* 0=not in daylight savings time (OFF)
                                    nonzero=ON */
    LONG   startDSTime;          /* Next time DST starts */
    LONG   stopDSTime;           /* Next time DST ends */
} Synchronized_Clock_T;
```

This structure is not the only one involved in keeping track of time in the NetWare OS, but provides a method for obtaining information about the parameters that effect the time of day clock on the server and altering some of that information.

The fields of most interest are *clock*, *tickIncrement*, *adjustmentCount*, *adustmentValue*, and *grossCorrection*. The *clock* field, as well as the *tickIncrement*, *adjustmentValue*, and *grossCorrection*, consists of two parts, whole seconds and fractional seconds. Conceptually, there is a binary point between the whole and fractional seconds parts. The whole seconds part of the clock holds UTC time in seconds since the start of 1970. NetWare UTC time does not account for leap seconds.

Thread Group Level Context

All of the threads in a thread group share the same thread group level context. Any change that one thread makes to the value of a thread group data item affects all the threads in the group. The context of a thread group, however, is not shared with other thread groups, so changes within one thread group do not affect another group.

Thread groups maintain the following context:

Current connection

The current connection number is described in Connection Number and Task Management.

Current screen

The current screen is the target of screen I/O functions, as described in Screen Handling.

Current task

The current task number is described in Connection Number and Task Management.

CWD

Current working directory, as described in File System.

Signal settings

Signal handler functions are set by **signal**. (The **signal** and **raise** functions are discussed in Signal Handling.)

stdin, *stdout*, *stderr*

These data items are the second-level standard I/O handles, as described in Stream I/O.

umask flags

These flags are set by the **umask** function, as described in File System.

Current user

The "current user" is the user context used by NDS™ functions (NetWare® 4.x only).

Thread Level Context

Thread level context is the most private level of context information within an NLM™ application; the context of each thread is available only to each thread. These values are separate for each thread. The data items of one thread cannot be referenced by another thread.

Threads maintain the following context:

asctime char string pointer

This character string is only allocated if the **asctime, asctime_r** function is called. The **asctime, asctime_r** function returns a char *.

critical section count

This count contains the number of outstanding **EnterCritSec** calls against a thread.

ctime, ctime_r, **gmtime, gmtime_r**, and **localtime, localtime_r** tm structure pointer

The **ctime, ctime_r**, **gmtime, gmtime_r**, and **localtime, localtime_r** functions return a pointer to a tm structure. Each thread has its own tm structure. The tm structure is only allocated if one of these three functions is called.

errno

Some functions set the *errno* return code to the last error code that was detected.

Last value from the function **rand**

Each thread has its own seed value (to start or continue a sequence of

random numbers). (See **rand, rand_r**.)

NetWareErrno

This error code is a NetWare specific error code. Some functions set both *NetWareErrno* and *errno*.

Stack

This points to the block of memory **BeginThread** allocated for the thread's stack.

strtok pointer

The **strtok** function maintains a pointer into the string being parsed.

t_errno

This error code is used by TLI functions.

Suspend count

This count contains the number of outstanding **SuspendThread** calls against a thread.

Thread Termination

If an operator unloads an NLM™ application, all threads are destroyed except the current thread, which started the unloading process (usually the console command thread) and executes the remaining shutdown code. The NetWare® API performs this step automatically.

During self-termination, the NetWare API destroys all threads created by the NLM, except the thread that caused the termination. This thread remains in order to execute the remaining shutdown code.

During an abnormal exit, all threads (including the current thread) are destroyed. The NetWare API performs this step automatically.

Transaction Tracking and SFT III

Benefits of TTS™ services are not superseded by SFT III™ mirrored operation. You should still use TTS for database applications. TTS should be viewed as a safeguard against logical (rather than physical) data corruption, because it makes atomic a series of granular write operations. SFT III by itself does not provide this capability; it relies on TTS to do so, just as the standard NetWare® OS does.

In practice, SFT III developers should rely on TTS more than developers of standard NetWare applications should. The use of SFT III implies that all the server's data is critical, which in turn implies that the data should be protected by TTS.

Transferring Functions from VAPs to NLM Applications

If your VAP uses functions from the NetWare® C Interface---DOS Library, you can easily convert the program to an NLM that uses functions from the NetWare API. See NLM Equivalents for VAP Functions to see the differences between these two interfaces.

Renaming VAP Functions

VAP Presentation Functions

Custom VAP Functions

Types of NLM Applications

NLM™ applications allow you to add functionality in increments to NetWare® 3.x and 4.x server platforms. You can produce your own customized utilities, LAN and disk drivers, and server applications. The following types of NLM applications add flexibility and a wide range of capabilities to NetWare:

Service, management, utility modules, including serial I/O drivers

LAN driver modules

Disk driver modules

Modules that define file system name spaces

NetWare includes NLM applications from each of these categories. For example:

INSTALL and MONITOR are utility NLM applications.

NE1000.LAN and NE2000.LAN are LAN drivers.

ISADISK.DSK is a disk driver.

OS2.NAM supports the OS/2* High Performance File System naming convention. (Support for DOS and FAT-based OS/2 naming conventions are part of the NetWare OS and not loaded as NLM applications.)

NOTE: This manual describes the process of writing management, utility, and server NLM applications. It does not explain how to write LAN drivers or device drivers. This information is available in the Novell LAN Driver Developer's Kit and in the Novell Device Driver Developer's Kit, which are Novell Labs™ publications.

Related Topics:

Utility and Service Modules

LAN and Device Driver Modules

Name Space Modules

Unloading NLM Applications

NLM™ applications 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 OS generates a warning for each resource that has not been explicitly released.

You can unload NLM applications 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 function registered with **AtUnload** is called. The functions registered with **atexit** are called as well.

> **NOTE:** A detailed explanation of the unload process is presented in NLM Code Development.

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.

Using the LOAD Command

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

Command line parameters that are passed to the NLM™ application

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. If no filename extension is specified, then files with extensions .NLM, .LAN, .DSK, and .NAM are searched

for.

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.

(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 the paths are specified, "\\" must be used because the "/" is interpreted as a new option. (You can also you redirection in ways similar to how it is done in DOS and UNIX*.)

/<*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 the paths are specified, "\\" must be used because the "/" is interpreted as a new option.

See Stream I/O 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. If your NLM does not run with DOMAIN.NLM loaded, try this option.

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 NLM applications 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>
```

Detailed information on this security enhancement is available in the readme, SECURE.DOC, includedwith the security enhancement. The files are available on the NetWire[SM] service. For information call 1-800-REDWORD.

/N

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

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

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

/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 4.x server.

For NLM applications 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 for information on how to write NLM applications 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
```

Utility and Service Modules

Some of the well-known NetWare® utilities such as DISKSET, INSTALL, VREPAIR, and MONITOR are NLM™ applications. You can load these utilities after the NetWare OS is loaded.

Developers of service and utility NLM applications use the NetWare API to create them. The NetWare API includes the following:

ANSI C runtime library

POSIX functions

De facto standard DOS C runtime library functions

OS/2* execution thread APIs

Interface APIs to the NetWare OS

The NetWare API is a set of NLM applications that link to the NetWare OS through its dynamic linking mechanism. The interface consists of the modules listed in the following table.

Table auto. NLM Applications in the NetWare API

NLM	Description
AFTER311	An NLM that can be loaded on NetWare 3.11 servers to allow NLM applications written for the NWSNUT interface to run on NetWare 3.11 and 3.12 servers.
AIO.NLM	Provides an interface through which applications can access asynchronous communication services in the NetWare 3.11 and 4.x environments. For detailed information on its use, see Asynchronous I/O.
CLIB.NLM	Contains the core set of functions needed for NLM development. In addition to functions unique to NetWare, it includes functions derived from the ANSI C Runtime Library, BSD Socket API, POSIX API, WATCOM* C Runtime Library, and UNIX* System Calls.
CLIBDEB.NLM	A version of CLIB.NLM that exports debugging information and implements stack checking. This NLM should be used for development, but it is not available to users.
DSAPI.NLM	Contains the NDS™ functions that are necessary to access the Directory. (This is new with the NetWare 4.0 OS.)
MATHLIB.NLM	Contains trigonometric, logarithmic, and other advanced math functions. Use this NLM if an 80387 numeric data processor is installed. This NLM requires that CLIB.NLM be loaded.
MATHLIBC.NLM	Contains the same set of functions as MATHLIB.NLM. Use MATHLIBC.NLM if an 80387 numeric data processor is not installed. This NLM requires that CLIB.NLM be loaded.
PATCH311.NLM	This is a dummy file that is used for NetWare 3.11 NLM applications that autoload PATCH311.NLM. The functions that were in PATCH311.NLM have been moved to CLIB.NLM.
TLI.NLM	Contains a set of functions that emulate theSystem V™ STREAMS TLI, a transport-independent network access method on UNIX systems. TLI.NLM autoloads STREAMS.NLM and CLIB.NLM if they are not already loaded.

The STREAMS based TLI functions are documented in TLI.

In addition to the NetWare API, the service module NWSHELL is supplied with this SDK. NWSHELL provides a command interface for the NetWare 3.x and 4.x environments, similar to the DOS command shell, COMMAND.COM. For more information about this command processor,

see the README file in the NOVM\NWSHELL directory.

VAP Equivalents for Accounting Functions

The Accounting Services functions for NLM™ applications and VAPs are the same. The following table shows VAP functions, their NLM counterparts, and the associated header file. For more information on Accounting Services, see Accounting.

VAP Function	NLM Function	NLM Header
AccountingInstalled	AccountingInstalled	nwaccntg.h niterror.h
GetAccountStatus	GetAccountStatus	nwaccntg.h niterror.h
SubmitAccountCharge	SubmitAccountCharge	nwaccntg.h niterror.h
SubmitAccountHold	SubmitAccountHold	nwaccntg.h niterror.h
SubmitAccountNote	SubmitAccountNote (This function does not use the *commentLen* parameter.)	nwaccntg.h nwtypes.h

VAP Equivalents for AFP Functions

The AFP Services functions for NLM™ applications and VAPs are the same. However, whereas VAPs can log in to a maximum of eight servers, NLM applications can log in to any number, as long as there are connections available.

VAP Function	NLM Function	NLM Header
AFPAllocTemporaryDirHandle	AFPAllocTemporaryDirHandle	nwafp.h nwtypes.h
AFPCreateDirectory	AFPCreateDirectory	nwafp.h nwtypes.h
AFPCreateFile	AFPCreateFile	nwafp.

		h nwtypes.h
AFPDelete	AFPDelete	nwafp.h nwtypes.h
AFPDirectoryEntry	AFPDirectoryEntry	nwafp.h nwtypes.h
AFPGetEntryIDFromName	AFPGetEntryIDFromName	nwafp.h nwtypes.h
AFPGetEntryIDFromNetWareHandle	AFPGetEntryIDFromNetWareHandle (Additional parameter)	nwafp.h nwtypes.h
AFPGetEntryIDFromPathName	AFPGetEntryIDFromPathName	nwafp.h nwtypes.h
AFPGetFileInformation	AFPGetFileInformation (Different structure)	nwafp.h nwtypes.h
AFPOpenFileFork	AFPOpenFileFork	nwafp.h nwtypes.h
AFPRename	AFPRename	nwafp.h nwtypes.h
AFPScanFileInformation	AFPScanFileInformation (Different structure)	nwafp.h nwtypes.h
AFPSetFileInformation	AFPSetFileInformation (Different structure)	nwafp.h nwtypes.h
AFPSupported	AFPSupported	nwafp.h

		nwtypes.h

VAP Equivalents for Bindery Functions

The Bindery Services functions available to NLM™ applications are the same as those available for VAPs. For more information on the Bindery functions, see Bindery.

VAP Function	NLM Function	NLM Header
AddBinderyObjectToSet	AddBinderyObjectToSet	nwbindry.h nwtypes.h
ChangeBinderyObjectPassword	ChangeBinderyObjectPassword	nwbindry.h nwtypes.h
ChangeBinderyObjectSecurity	ChangeBinderyObjectSecurity	nwbindry.h nwtypes.h
ChangePropertySecurity	ChangePropertySecurity	nwbindry.h nwtypes.h
CloseBindery	CloseBindery	nwbindry.h nwtypes.h
CreateBinderyObject	CreateBinderyObject	nwbindry.h nwtypes.h
CreateProperty	CreateProperty	nwbindry.h nwtypes.h
DeleteBinderyObject	DeleteBinderyObject	nwbindry.h nwtypes.h

DeleteBinderyObjectFromSet	DeleteBinderyObjectFromSet	nwbindry.h nwtypes.h
DeleteProperty	DeleteProperty	nwbindry.h nwtypes.h
GetBinderyAccessLevel	GetBinderyAccessLevel	nwbindry.h nwtypes.h
GetBinderyObjectID	GetBinderyObjectID	nwbindry.h nwtypes.h
GetBinderyObjectName	GetBinderyObjectName	nwbindry.h nwtypes.h
isBinderyObjectInSet	IsBinderyObjectInSet	nwbindry.h nwtypes.h
OpenBindery	OpenBindery	nwbindry.h nwtypes.h
ReadPropertyValue	ReadPropertyValue	nwbindry.h nwtypes.h
RenameBinderyObject	RenameBinderyObject	nwbindry.h nwtypes.h
ScanBinderyObject	ScanBinderyObject	nwbindry.h nwtypes.h
ScanBinderyObjectTrusteePaths	ScanBinderyObjectTrusteePaths	nwbindry.h nwtypes.h
ScanProperty	ScanProperty	nwbindr

		y.h nwtypes.h
VerifyBinderyObjectPassword	VerifyBinderyObjectPassword	nwbindry.h nwtypes.h
WritePropertyValue	WritePropertyValue	nwbindry.h nwtypes.h

The NetWare® API provides the following additional defined properties listed in the following table.

Table auto. Bindery Properties

Property Name	Object Type
ACCOUNT_HOLDS	
ACCT_LOCKOUT	ACCT_LOCKOUT
ACCT_LOCKOUT	User
BLOCKS_READ	File Server
BLOCKS_WRITTEN	File Server
CONNECT_TIME	File Server
DISK_STORAGE	File Server
MODE_CONTROL	User
REQUESTS_MADE	File Server

MANAGERS is not a defined property in the Bindery Services of the NetWare API.

VAP Equivalents for Communication Functions

The IPX™ packet header for NLM™ applications is different from the one for VAPs. (It is longer and contains different fields.) The SPX™ packet header is similar for both NLM applications and VAPs with one main exception: whereas the VAP fields are either bytes or words, the NLM fields are unsigned chars and unsigned shorts. The ECB structure is also different.

The following table shows VAP functions that can be mapped to NLM functions. Note the spelling differences between the two libraries.

VAP Function	NLM Function	NLM Header
IPXCancelEvent	IpxCancelEvent or IpxCancelPacket	nwipxspx.h
IPXCloseSocket	IpxCloseSocket	nwipxspx.h
IPXDisconnectFromTarget	Not applicable for NLM	
IPXGetDataAddress	Not applicable for NLM	
IPXGetInternetworkAddress	IPXGetInternetworkAddress	nwipxspx.h
IPXGetIntervalMarker	SpxGetTime	nwipxspx.h
IPXGetLocalTarget	IPXGetLocalTarget	nwipxspx.h
IPXInitialize	Not applicable for NLM	
IPXListenForPacket	IpxReceive (Different parameters)	nwipxspx.h
IPXOpenSocket	IPXOpenSocket	nwipxspx.h
IPXRelinquishControl	Not applicable for NLM	
IPXScheduleIPXEvent	Not applicable for NLM	
IPXSendPacket	IpxSend (Different parameters)	nwipxspx.h
SPXAbortConnection	SpxAbortConnection	nwipxspx.h
SPXEstablishConnection	SpxEstablishConnection	nwipxspx.h

VAP Equivalents for Connection Functions

Many of the Connection Services functions are the same between the two libraries. The following tables show VAP functions, theirNLM™ counterparts, and the associated header file.

VAP Function	NLM Function	NLM Header
AttachToFileServer	Not applicable in NetWare 3.x	
AttachToFileServerWithAddress	Not applicable in NetWare 3.x	

ddress	NetWare 3.x	
DetachFromFileServer	Not applicable in NetWare 3.x	
EnterLoginArea	Not applicable in NetWare 3.x	
GetConnectionInformation	GetConnectionInformation	nwconn.h nwtypes.h
GetConnectionNumber	GetConnection Number	nwconn.h nwtypes.h
GetInternetAddress	GetInternetAddress (Different parameters)	nwconn.h nwtypes.h
GetObjectConnectionNumbers	GetObjectConnectionNumbers	nwconn.h nwtypes.h
GetStationAddress	GetStationAddress	nwconn.h nwtypes.h
LoginToFileServer	LoginToFileServer	nwconn.h nwtypes.h
Logout	Logout	nwconn.h nwtypes.h
LogoutFromFileServer	LogoutFromFileServer	nwconn.h nwtypes.h

VAP Equivalents for File System Functions

Several File Services functions are different between the two libraries. The following table shows VAP functions, their NLM™ counterparts, and the associated header file. For more information on File Services, see File System .

Table auto. Directory Functions

VAP Function	NLM Function	NLM Header
AddTrusteeToDirectory	AddTrustee	nwdir.h
AllocPermanentDirectory Handle	Not applicable for NLM	
AllocTemporaryDirectory Handle	Not applicable for NLM	
ClearVolRestrictions	DeleteUserSpaceRestriction (Different parameters)	nwdir.h
CreateDirectory	mkdir (Different parameters)	direct.h nwtypes.h

DeallocateDirectoryHandle	Not applicable for NLM	
DeleteDirectory	rmdir (Different parameters)	direct.h nwtypes.h
DeleteTrustee	Delete Trustee (Different parameters)	nwdir.h
DeleteTrusteeFromDirectory	DeleteTrustee (Different parameters)	nwdir.h
GetCurrentDirectory	getcwd (Different parameters)	direct.h nwtypes.h
GetDirectoryHandle	Not applicable for NLM	
GetDirectoryPath	Not applicable for NLM	
GetDirEntry	readdir (Different parameters and structure)	nwdir.h
GetDirInfo	Not applicable for NLM	
GetDirveInformation	Not applicable for NLM	
GetEffectiveDirectoryRights	GetEffectiveRights (Different parameters)	nwdir.h
GetEffectiveRights	GetEffectiveRights	nwdir.h
GetObjectDiskRestrictions	GetDiskSpaceUsedByObject or GetMaximumUserSpaceRestrictions (Different parameters)	
GetSearchDriveVector	Not applicable for NLM	
GetVolumeInformation	GetVolumeInformation (Different Structure)	nwdir.h
GetVolumeInfoWithHandle	GetVolumeStatistics (Different parameters)	nwdir.h
GetVolumeInfoWithNumber	GetVolumeInfoWithNumber	nwdir.h
GetVolumeName	GetVolumeName	nwdir.h
GetVolumeNumber	GetVolumeNumber	nwdir.h
GetVolUsage	GetVolumeStatistics (Different parameters)	nwdir.h
IsSearchDrive	Not applicable for NLM	
MapDrive	Not applicable for NLM	
MapDriveUsingString	Not applicable for NLM	
MapV2TrusteeRightsToV3	Not applicable for NLM	
MapV3TrusteeRightsToV	Not applicable for NLM	

2		
ModifyMaximumRights Mask	ModifyInheritedRightsM ask	nwdir.h
MoveEntry	rename (Different parameters)	stdio.h
RenameDirectory	rename (Different parameters)	stdio.h
RestoreDirectoryHandle	Not applicable for NLM	
SaveDirectoryHandle	Not applicable for NLM	
ScanBinderyObjectTruste ePaths	ScanBinderyObjectTrusst eePaths	nwdir.h
ScanDirectoryForTrustees	Scan Trustees (Different parameters)	nwdir.h
ScanDirectoryInformatio n	stat (Different parameters and structure)	nwfile.h stdlib.h
ScanDirEntry	stat (Different parameters and structure)	nwfile.h stdlib.h
ScanDirRestrictions	ScanUserSpaceRestriction s (Diferent parameters)	nwdir.h
ScanEntryForTrustees	ScanTrustees (Different parameters)	nwdir.h
ScanVolForRestrictions	ScanUserSpaceRestriction s (Different parameters)	nwdir.h
SetDirectoryHandle	Not applicable for NLM	
SetDirectoryInformation	SetDirectoryInfo (Different Parametes)	nwdir.h
SetDirRestriction	AddSpaceRestrictionFor Directory (Different parameters)	nwdir.h
SetEntry	ChangeDirectoryEntry (Different parameters)	nwdir.h
SetSearchDriveVector	Not applicable for NLM	
SetTrustee	AddTrustee (Different parameters)	nwdir.h
SetVolRestriction	AddUserSpaceRestriction	nwdir.h

Table auto. File Functions

VAP Function	**NLM Function**	**NLM Header**
EraseFiles	remove	stdio.h
FileServerFileCopy	FileServerFileCopy	nwfile.h

		nwtypes.h direct.h
GetExtendedFileAttributes	GetExtendedFileAttributes	nwfile.h nwtypes.h direct.h
PurgeAllErasedFiles	PurgeErasedFile (Different parameters)	nwfile.h nwtypes.h direct.h
PurgeErasedFiles	PurgeErasedFile (Different parameters)	nwfile.h nwtypes.h direct.h
PurgeSalvagableFile	PurgeErasedFile (Different parameters)	nwfile.h nwtypes.h direct.h
RecoverSalvagableFile	SalvageErasedFile (Different parameters)	nwfile.h nwtypes.h direct.h
RestoreErasedFile	SalvageErasedFile (Different parameters)	nwfile.h nwtypes.h direct.h
ScanFileEntry	stat (Different parameters and structure)	nwfile.h stdlib.h
ScanFileInformation	SalvageErasedFile (Different parameters)	nwfile.h stdlib.h
ScanFilePhysical	SalvageErasedFile (Different parameters)	nwfile.h stdlib.h
ScanSalvagableFiles	ScanErasedFiles (Different parameters)	nwfile.h nwtypes.h direct.h
SetExtendedFileAttributes	SetExtendedFileAttributes	nwfile.h nwtypes.h direct.h
SetFileInformation	SetFileInfo (Different parameters)	nwfile.h nwtypes.h direct.h

VAP Equivalents for Message Functions

The following table shows the VAP Message Services functions, their NLM™ counterparts, and the associated header file. For more information

on Message Services, see Message.

VAP Function	NLM Function	NLM Header
BroadcastToConsole	BroadcastToConsole	nwmsg.h nwtypes.h
CheckPipeStatus	Not applicable for NLM	
CloseMessagePipe	Not applicable for NLM	
DisableBroadcasts	DisableStationBroadcasts	nwmsg.h nwtypes.h
EnableBroadcasts	EnableStationBroadcasts	nwmsg.h nwtypes.h
GetBroadcastMessage	GetBroadcastMessage	nwmsg.h nwtypes.h
GetBroadcastMode	Not applicable for NLM	
GetPersonalMessage	Not applicable for NLM	
LogNetworkMessage	Not applicable for NLM	
OpenMessagePipe	Not applicable for NLM	
SendBroadcastMessage	SendBroadcastMessage	nwmsg.h nwtypes.h
SendPersonalMessage	Not applicable for NLM	
SetBroadcastMode	Not applicable for NLM	

VAP Equivalents for Miscellaneous Functions

The following table shows VAP Miscellaneous Services functions, their NLM™ counterparts, and the associated header file. For more information on Miscellaneous Services, see Reorganization of Miscellaneous Service for the current location of information.

VAP Function	NLM Function	NLM Header
ASCIIZToLenStr	ASCIIZToLenStr	nwstring.h nwtypes.h
GetNetworkSerialNumber	GetNetworkSerialNumber	nwserial.h nwtypes.h
IntSwap	IntSwap	nwstring.h

		nwtypes.h
IsV3Supported	Not applicable for NLM	
LenStrCat	LenStrCat	nwstring.h nwtypes.h
LenStrCmp	LenStrCmp	nwstring.h nwtypes.h
LenStrCpy	LenStrCpy	nwstring.h nwtypes.h
LenToASCIIZStr	LenToASCIIZStr	nwstring.h nwtypes.h
LongSwap	LongSwap	nwstring.h nwtypes.h
StripoFileServerFromPath	StripoFileServerFromPath	nwdir.h nwtypes.h
VerifyNetworkSerialNumber	VerifyNetworkSerialNumber	nwserial.h nwtypes.h

VAP Equivalents for Queue Management Functions

All of the Queue Services functions are different between the two libraries. The following table shows VAP Queue Services functions, their NLM™ counterparts, and the associated header file.

VAP Function	NLM Function	NLM Header
AbortServicingQueueJobAndFile	NWQAbortJobService (Different parameters)	nwqueue.h nwtypes.h
AttachQueueServerToQueue	NWQAttachServer (Different parameters)	nwqueue.h nwtypes.h
ChangeQueueJobEntry	NWQChangeJobEntry (Different parameters)	nwqueue.h nwtypes.h
ChangeQueueJobPosition	NWQChangeJobPosition (Different parameters)	nwqueue.h

		nwtypes.h
ChangeToClientRights	NWQChangeToClientRights (Different parameters)	nwqueue.h nwtypes.h
CloseFileAndAbortQueueJob	NWQAbortJob (Different parameters)	nwqueue.h nwtypes.h
CloseFileAndStartQueueJob	NWQMarkJobForService (Different parameters)	nwqueue.h nwtypes.h
CreateQueueJobAndFile	NWQCreateJob (Different parameters)	nwqueue.h nwtypes.h
CreateQueue	NWQCreate (Different parameters)	nwqueue.h nwtypes.h
DestroyQueue	NWQDestroy (Different parameters)	nwqueue.h nwtypes.h
DetachQueueServerFromQueue	NWQDetachServer (Different parameters)	nwqueue.h nwtypes.h
FinishServicingQueueJobAndFile	NWQEndJobService (Different parameters)	nwqueue.h nwtypes.h
GetQueueJobList	NWQScanJobNums (Different parameters)	nwqueue.h nwtypes.h
GetQueueJobsFileSize	NWQGetJobFileSize (Different parameters)	nwqueue.h nwtypes.h
ReadQueueCurrentStatus	NWQGetStatus NWQGetServers (Different parameters)	nwqueue.h nwtypes.h

ReadQueueJobEntry	NWQGetJobEntry (Different parameters)	nwqueue.h nwtypes.h
ReadQueueServerCurrentStatus	NWQGetServerStatus (Different parameters)	nwqueue.h nwtypes.h
RemoveJobFromQueue	NWQRemoveJob (Different parameters)	nwqueue.h nwtypes.h
RestoreQueueServerRights	NWQRestoreServerRights (Different parameters)	nwqueue.h nwtypes.h
ServiceQueueJobAndOpenFile	NWQBeginJobService (Different parameters)	nwqueue.h nwtypes.h
SetQueueCurrentStatus	NWQSetStatus (Different parameters)	nwqueue.h nwtypes.h
SetQueueServerCurrentStatus	NWQSetServerStatus (Different parameters)	nwqueue.h nwtypes.h

VAP Equivalents for SAP Functions

SAP Services functions are similar for VAPs and NLM™ applications. The following table shows VAP functions, their NLM counterparts and the associated file header. For more information on SAP Services, seeSAP.

VAP Function	NLM Function	NLM Header
AdvertiseService	AdvertiseService (NLM)	sap.h nwipxspx.h nwtypes.h
QueryServices	QueryServices (Different syntax)	sap.h nwipxspx.h

		nwtypes.h
ShutdownSAP	ShutdownAdvertising (Additional Parameter)	sap.h nwipxspx.h nwtypes.h

VAP Equivalents for Server Environment Functions

Several Server Environment functions are different between the two libraries. The following table shows VAP functions, theirNLM™ counterparts, and the associated header file. For more information on Server Environment Services, see Server Environment.

VAP Function	NLM Function	NLM Header
CheckConsloePrivileges	CheckConsolePrivileges	nwenvrn.h nwtypes.h
CheckNetWareVersion	CheckNetWareVersion	nwenvrn.h nwtypes.h
ClearConnectionNumber	ClearConnectionNumber	nwenvrn.h nwtypes.h
DisableFileServerLogin	DisableFileServerLogin	nwenvrn.h nwtypes.h
DisableTransactionTracking	DisableTransactionTracking	nwenvrn.h nwtypes.h
DownFileServer	DownFileServer	nwenvrn.h nwtypes.h
EnableFileServerLogin	EnableFileServerLogin	nwenvrn.h nwtypes.

		h
EnableTransactionTracking	EnableTransactionTracking	nwenvrn.h nwtypes.h
GetBinderyObjectDiskSpaceLeft	GetBinderyObjectDiskSpaceLeft	nwenvrn.h nwtypes.h
GetConnectionOpenFile	Not applicable for NLM	
GetConnectionsSemaphores	Not applicable for NLM	
GetConnectionsTaskInformation	Not applicable for NLM	
GetConnectionsUsageStats	Not applicable for NLM	
GetConnectionsUsingFile	Not applicable for NLM	
GetDiskCacheStats	Not applicable for NLM	
GetDiskChannelStats	Not applicable for NLM	
GetDiskUtilization	GetDiskUtilization	nwenvrn.h nwtypes.h
GetDriveMappingTable	Not applicable for NLM	
GetFileServerDateAndTime		
GetFileServerDescriptionStrings		
GetFileServerLANIOStats	Not applicable for NLM	
GetFileServerLoginStatus		
GetFileServerMiscInformation	Not applicable for NLM	
GetFileServerName		
GetFileSystemStats	Not applicable for NLM	
GetLanDriverConfigInfo	Not applicable for NLM	
GetLogicalRecordInformation	Not applicable for NLM	
GetLogicalRecordsByConnection	Not applicable for NLM	
GetPathFromDirectoryEntry	Not applicable for NLM	
GetPhysicalRecordLocksByFile	Not applicable for NLM	
GetPhysRecLocksByConnectAndFile	Not applicable for NLM	

GetSemaphoreInformation	Not applicable for NLM	
GetServerInformation	GetServerInformation (Different parameters)	nwenvrn.h nwtypes.h
SendConsoleBroadcast	SendConsoleBroadcast	nwenvrn.h nwtypes.h
SetFileServerDateAndTime	SetFileServerDateAndTime	nwenvrn.h nwtypes.h
TTSGetStats	Not applicable for NLM	

VAP Equivalents for Synchronization Functions

In general, the Synchronization Services functions are the same for VAPs and NLM™ applications. The following table shows VAP functions, their NLM counterparts and the associated header file. For more information on Synchronization Services, see NLM Synchronization.

VAP Function	NLM Function	NLM Header
Clear File	Clear File	nwsync.h nwtypes.h
ClearFileSet	ClearFileSet	nwsync.h nwtypes.h
ClearLogicalRecord	ClearLogicalRecord	nwsync.h nwtypes.h
ClearLogicalRecordSet	ClearLogicalRecordSet	nwsync.h nwtypes.h
ClearPhysicalRecord	ClearPhysicalRecord	nwsync.h nwtypes.h
ClearPhysicalRecordSet	ClearPhysicalRecordSet	nwsync.h nwtypes.h
CloseSemaphore	CloseSemaphore	nwsync.h nwtypes.h
ExamineSemaphore	ExamineSemaphore	nwsync.h nwtypes.h
GetLockMode	GetLockMode	nwsync.h

		nwtypes.h
LockFileSet	LockFileSet	nwsync.h nwtypes.h
LockLogicalRecordSet	LockLogicalRecordSet	nwsync.h nwtypes.h
LockPhysicalRecordSet	LockPhysicalRecordSet	nwsync.h nwtypes.h
LogFile	LogFile	nwsync.h nwtypes.h
LogLogicalRecord	LogLogicalRecord	nwsync.h nwtypes.h
LogPhysicalRecord	LogPhysicalRecord	nwsync.h nwtypes.h
OpenSemaphore	OpenSemaphore	nwsync.h nwtypes.h
ReleaseFile	ReleaseFile	nwsync.h nwtypes.h
ReleaseFileSet	ReleaseFileSet	nwsync.h nwtypes.h
ReleaseLogicalRecord	ReleaseLogicalRecord	nwsync.h nwtypes.h
ReleaseLogicalRecordSet	ReleaseLogicalRecordSet	nwsync.h nwtypes.h
ReleasePhysicalRecord	ReleasePhysicalRecord	nwsync.h nwtypes.h
ReleasePhysicalRecordSet	ReleasePhysicalRecordSet	nwsync.h nwtypes.h
SetLocKMode	SetLocKMode	nwsync.h nwtypes.h
SignalSemaphore	SignalSemaphore	nwsync.h nwtypes.h
WaitOnSemaphore	WaitOnSemaphore	nwsync.h nwtypes.h

VAP Equivalents for TTS Functions

In general, the TTS™ Services functions are the same for both VAPs and NLM™ applications. The following table shows VAP functions, their NLM counterparts, and the associated header files. For more information on TTS,

see TTS.

VAP Function	NLM Function	NLM Header
TTSAbortTransaction	TTSAbortTransaction	nwtts.h nwtypes.h
TTSBeginTransaction	TTSBeginTransaction	nwtts.h nwtypes.h
TTSEndTransaction	TTSEndTransaction	nwtts.h nwtypes.h
TTSGetApplicationThresholds	TTSGetApplicationThresholds	nwtts.h nwtypes.h
TTSGetWorkstationThresholds	TTSGetWorkstationThresholds	nwtts.h nwtypes.h
TTSIsAvailable	TTSIsAvailable	nwtts.h nwtypes.h
TTSSetApplicationThresholds	TTSSetApplicationThresholds	nwtts.h nwtypes.h
TTSSetWorkstationThresholds	TTSSetWorkstationThresholds	nwtts.h nwtypes.h
TTSTransactionStatus	TTSTransactionStatus	nwtts.h nwtypes.h

VAP Presentation Functions

NLM™ presentation at the server console offers a substantial gain over VAP console presentation. NLM applications can access a wide variety of screen and I/O management functions using the NetWare® API. These functions allow NLM applications to set up and maintain their own console screens.

Although almost all NLM applications use their own screen, if necessary they can use the server's system console screen and hook into the command line parser for that screen. NLM applications hook into the parser by specifying to the OS those commands they want to act upon. This process is similar to the use of keywords in a VAP header to register commands the VAP acts upon.

Whenever the OS receives an unrecognized command, it calls the command parsing function. If you specify an NLM command using the NetWare API function **RegisterConsoleCommand**, the NLM gets a chance to identify and act on that command.

> **NOTE:** The NLM can specify only those commands unique to itself. That is, the NLM cannot specify a command used by the OS or another

NLM.

VAP Processes and NLM Threads

Both VAPs and NLM™ applications use multiple processes to accomplish tasks. In an NLM, these processes are called **threads**. There are a few differences between VAP processes and NLM threads:

Initialization---VAPs use a VAP header that includes information for spawning all necessary processes and allocating enough memory for those processes. (You cannot increase VAP memory allocation after initialization.) In contrast, NLM applications can begin and end threads and allocate memory as needed at any point in the program.

Context and Grouping---Whereas all VAP processes share the same limited context, NLM applications use thread grouping to arrange threads into groups. This allows threads within a thread group to share context, such as the CWD, current screen, and current connection.

For more information on multiple execution threads in NLM applications, see Thread and NLM Code Development.

VAP Services

The following VAP Services can be replaced by the designated NLM™ functions:

VAP Function	NLM Function	NLM Header
AllocateSegment	malloc	malloc.h
CalculateAbsoluteAddress	Not necessary except to developers of LAN and device drivers.	
ChangeProcess	ThreadSwitch	process.h
ChangeSegmentToData	Not applicable for NLM	
ClearScreen	clrscr	conio.h nwtypes.h
ConsoleDisplay	ConsolePrintf	conio.h nwtypes.h
ConsoleError	ConsolePrintf	conio.h nwtypes.h
ConsoleMessage	ConsolePrintf	conio.h nwtypes.h
ConsoleQuery	Not applicable for NLM	

ConsoleRead	Not applicable for NLM	
CreateProcess	BeginThread or BeginThreadGroup	process.h
DeclareExtendedSegment	Not applicable for NLM	
DeclareSegmentAsData	Not applicable for NLM	
DelayProcess	delay	process.h
DoConsoleError	ConsolePrintf (This function does not print errors to the SYS$LOG.ERR file.)	conio.h nwtypes.h
GetInterruptVector	Not applicable for NLM	
GetProcessID	GetThreadID	process.h
GetScreenMode	GetCurrentScreen	conio.h nwtypes.h
GetVAPHeader	Not applicable for NLM	
InitializationComplete	Not applicable for NLM	
InString	Not applicable for NLM	
KillProcess	ExitThread	process.h
OutString	ConsolePrintf	conio.h nwtypes.h
PrintString	ConsolePrintf	conio.h nwtypes.h
ReadKeyboard	cgets (and others)	conio.h nwtypes.h
SegmentToPointer	Not applicable for NLM	
SetExternalProcessError	Not applicable for NLM	
SetHardwareInterruptVector	SetHardwareInterrupt	advanced.h
SetScreenMode	clrscr (and others)	conio.h nwtypes.h
ShellPassThroughEnable	Not applicable for NLM	
SleepProcess	SuspendThread	process.h
SpawnProcess	BeginThread	process.h
VAPAttachToFileServer	LoginToFileServer (Different parameters; does not attach to the server)	nwconn.h nwtypes.h
VAPGetConnectionID	GetConnectionInformation	nwconn.h nwtypes.h
VapGetFileServerName	GetFileServerName	nwconn.h

		nwenvrn.h nwtypes.h
WakeUpProcess	ResumeThread	process.h

VAP to NLM Conversion and Linking

The linking procedure is different for NLM™ applications than for VAPs. When you convert your VAP source code, remember that:

Instead of using a special header file as VAPs do, NLM applications are linked with an object module called PRELUDE.OBJ.

The WLINK command used to link NLM applications can accept a directive file for passing all options and arguments. (See Linker Directives and Options for a discussion of directive files.)

The directive file contains the list of files with functions you want to import and export. Your NLM can call (import) functions from other NLM applications for its own use. Likewise, your NLM can provide (export) functions to other NLM applications. Functions can be imported and exported only if specified in the module's directive file when linked.

For more information on linking an NLM, see NLM Linkers.

Whereas VAPs produce utilities only, there are four types of NLM applications, each defined by its function. The WATCOM linker (WLINK) converts an object file (.OBJ) to an executable file with an extension that corresponds to the function of the loadable module. NLM applications can be

Management utility and server application modules (.NLM)

LAN driver modules (.LAN)

Disk driver modules (.DSK)

Modules relating to file system name spaces (.NAM)

For more information on NLM types, see Types of NLM Applications.

VAP to NLM Conversion and Loading

You must load VAPs on a NetWare® 2.x server at boot time. In addition, you must load either all VAPs or no VAPs. In contrast, you can load and unload NLM™ applications dynamically as needed during server operation without bringing down the server or loading or unloading all other NLM applications.

Either a console command or another NLM can load an NLM. NLM applications can be unloaded by a console command or by themselves. However, an NLM cannot unload another NLM, at least not directly. For instance, if NLM1 loads NLM2, it can unload NLM2 only by issuing some sort of signal that NLM2 recognizes as an instruction to unload itself.

VAPs use the NetWare C Interface Library, which contains an interface for communicating with NetWare. The loading process links each VAP with the code required to handle server requests. This method of interface makes VAPs larger and requires more server memory.

By comparison, NLM applications use the dynamically linked NetWare API. This library is loaded once on the server. When NLM applications are loaded, the function calls they make to this library are resolved. This means that NLM applications are smaller than VAPs and take less memory.

For more information about the process of creating an executable NLM, see NLM Programming Overview.

Workstation Environment Services

The following table shows the VAP functions, their NLM™ counterparts and the associated header file.

VAP Function	NLM Function	NLM Header
EndOfJob	Not applicable for NLM	
GetConnectionID	GetConnectionInformation	nwconn.h nwtypes.h
GetDefaultConnectionID	GetDefaultConnectionID (Different return values)	nwconn.h
GetFileServerName	GetFileServerName	nwconn.h nwenvrn.h nwtypes.h
GetPreferredConnectionId	GetFileServerID (Different parameters)	nwconn.h
GetPrimaryConnectionId	GetFileServerID (Different parameters)	nwconn.h
IsConnectionIDInUse	Not applicable for NLM	
SetPreferredConnectionID	SetCurrentFileServerID	nwcntask.h
SetPrimaryConnectionID	SetCurrentFileServerID	nwcntask.h

Connection IDs for VAPs differ from server IDs for NLM applications. To better understand server IDs, see Connection.

NLM Development Tools

NLM Development Tools: Guides

NLM Development Tools: Concept Guide

Tools Overview

NLM Compilers

This information explains compilers and options that are specific toNLM™

development. This chapter is not intended to be a comprehensive guide for the compilers. For complete information about your compiler, see the documentation that came with it.

If you are going to link your NLM using NLMLINK, you must use a386* native mode compiler that generates PharLap* Easy Object Module Format (OMF).

WATCOM Compilers

Using the WATCOM Compilers

Passing Parameters

Using Pragmas

NLM Linkers

Two linkers are available for linking NLM™ applications. They are the WATCOM* linker (WLINK) and the Novell® linker (NLMLINK). Both support message files, which are used in enabling. WLINK provides debugging information for the WVIDEO debugger; NLMLINK does not.

Because the directives are sometimes different between the two linkers, descriptions of linker directives include WATCOM information.

Using NLMLINK

Using the WATCOM Linker

Specifying a Linker Directive File

Linker Directives and Options

Symbols Used by NLMLINK

Linker Definition Syntax

Linker Directives and Options

Imported and Exported Symbols

NLM Debuggers

Debugger Overview

Linking Debug Information with WLINK

Linking Debug Information with NLMLINK

Using the WVIDEO Debugger

Using the NetWare Internal Debugger

Setting Breakpoints

Specifying Expressions

Overview of Rdebug

Using NLMDebug

NLMDebug aids in debugging NLM™ applications and getting them ready for certification. This NLM gives you the capability to check the resources, semaphores, memory overwrites of your NLM and to debug NCP™ requests. You can also view CLIB remote connection information, perform function-count profiles and track the use of CLIB context.

Setting Up and Using NLMDebug

Changing Debug Settings

Watching CLIB Context

Using the NCP Debugger

Viewing CLIB Remote Connection Information

Using the Function-Call Profiler

Watching for File Opens

Using the Process Timer

Using Error Watch

NLMPACK and UNPACK Information

NLMPACK and UNPACK Information

NLMPACK and UNPACK Syntax

Memory Protection

In the NetWare® OS, NLM™ applications share the same data and code space area as the OS. (This sharing is an option for the NetWare 4.x OS.) This optimizes the speed that NLM applications run at, but does not provide any protection for the OS or other NLM applications. This is a

concern to people running mission-critical applications, since an erring NLM can bring down the server.

For developers, there is a limited amount of memory protection available for the NetWare 3.x OS through PROTECT.NLM, but for the end users there is no memory protection. The NetWare 4.x OS has memory protection as an option that is available for developers and for end users. NetWare 4.x memory protection is provided by DOMAIN.NLM.

The following forms of memory protection are discussed:

PROTECT.NLM---Used with NetWare 3.x servers. See PROTECT.NLM.

DOMAIN.NLM---Used with NetWare 4.x servers. See DOMAIN.NLM.

NLM Development Tools: Tasks

Breaking on a Connection Number

To have the debugger break on a specific connection number, do the following:

1. **From the NCP™ Debugger Setup Menu, select Break on Connection Number.**
2. **Enter the connection number you want NLMDebug to break on and press Enter.**

   NLMDebug breaks on the connection number entered.

   If you do not want the debugger to break on a connection number, enter 0 in the edit box.

Parent Topic: Using the NCP Debugger

Breaking on a Function Code

To have the debugger break on a specific function code, do the following:

1. **From the NCP™ Debugger Setup Menu, select Break on Function Code.**
2. **Enter the function code as a hex number and press Enter.**

   If you do not want to break on a function code, enter 0 in the edit box.

Parent Topic: Using the NCP Debugger

Breaking on a Subfunction Code

To have NLMDebug break on a specific subfunction code, do the following:

1. **From the NCP™ Debugger Setup Menu, select Break on Sub-Function Code.**
2. **Enter the subfunction code as a hexadecimal number and press Enter.**

If you do not want NLMDebug to break on a subfunction code, enter 0 in the edit box.

Parent Topic: Using the NCP Debugger

Breaking on an Error

To have NLMDebug break on errors, do the following:

1. **From the NCP™ Debugger Setup Menu, select Break on Error.**
2. **Type Y and press Enter.**

   The debugger breaks wherever an error is encountered in sending an NCP packet.

   If you do not want the debugger to break on errors, enter No in the edit box.

Parent Topic: Using the NCP Debugger

Breaking on Every Packet

To have the debugger break on every packet, do the following:

1. **From the NCP™ Debugger Setup Menu, select Break on Every Packet.**
2. **Type Y and press Enter.**

   The debugger breaks after each packet and only continues after you press the Spacebar.

   If you do not want the debugger to break on every packet, enter No in the edit box.

Parent Topic: Using the NCP Debugger

Changing Debug Settings

From the Debug Settings menu you can set NLMDebug to check for resources not freed, memory overwrites, semaphores not freed, and to report if a CLIB function is called without CLIB context. You can also set NLMDebug to ring a bell when information is written to the console and log errors to a file.

To set Debug options do the following:

1. **From the NLMDebug main menu select Debug Settings and press Enter.**

   The Debug Settings menu is displayed.

2. **Select each option and press Y or N to tailor NLMDebug to your needs.**

   If you want all errors logged to a file, enter a file name in the Log Errors To File edit box.

   For more information about options, see the following:

   Resource Checking

   Memory Checking

   Semaphore Checking

   Semaphore Monitoring

   Report No CLIB Context

   Ring the Bell on Error

   Auto-Save All Settings

   Log Output to Path

3. **Press Enter.**

4. **When you are done, press Esc.**

Parent Topic: Setting Up and Using NLMDebug

Delaying between Packets

To have the debugger break after each packet for a specified period of time, do the following:

1. **From the NCP™ Debugger Setup Menu, select Delay Between Packets.**

2. **Enter the delay time in seconds and press Enter.**

   The debugger breaks after each packet for the specified time before continuing.

   If you do not want the debugger to delay between packets, enter 0 in the edit box.

Parent Topic: Using the NCP Debugger

Loading NOVSERV.NLM

1. **The command syntax for loading NOVSERV.NLM is as follows:**

   ```
   LOAD NOVSERV server_name
   ```

   Where *server_name* is the name to be advertised as a communications server. If no name is specified, the default name is "NovLink." The name can be up to 47 characters and must follow the NetWare® naming rules for server names.

   Example:

   ```
   LOAD NOVSERV debug_server
   ```

Parent Topic: Setting Up the Server for Remote Debugging

Loading PARSERV.NLM

1. **The command syntax for loading PARSERV is as follows:**

   ```
   LOAD PARSERV port_number
   ```

   Where *port_number* is the number of the parallel port (1 through 3) to be used to communicate with the WVIDEO debugger. The default port number is 1.

   Example:

   ```
   LOAD PARSERV 1
   ```

   Remote debugging over the parallel port is faster than remote debugging over the serial port.

Parent Topic: Setting Up the Server for Remote Debugging

Loading SERSERV.NLM

1. **The command syntax for loading PARSERV is as follows:**

   ```
   LOAD SERSERV port_number . max_baud
   ```

   Where *port_number* is the number of the serial port (1, 2, 3, etc.) to be used to communicate with the WVIDEO debugger. The default number

is 1. The *max_baud* argument is maximum baud rate to start checking at to find an error-free transmission rate that can be used between the task and the debugger machine.

Example:

```
LOAD SERSERV 1.9600
```

The actual transmission rate can be lower than the one specified with the *max_baud* argument. Notice that a period (.) separates *port_number* from *max_baud*.

Parent Topic: Setting Up the Server for Remote Debugging

Using DOMAIN.NLM as a Development Tool

As a developer, you can catch many of the addressing errors in yourNLM™ applications by running them in the OS_PROTECTED domain. The advantage of developing on the NetWare 4.x operating system with DOMAIN.NLM is that you do not need to reboot your server when an NLM causes the operating system to abend.

To test NLM applications in the OS_PROTECTED domain, follow the steps below:

1. **Start with the developer option set to ON so the NLM will abend the server when it makes an addressing error.**
2. **Load your NLM for testing.**
3. **At an "abend" screen, enter the NetWare® Internal Debugger (by pressing Left-shift+Right-shift+Alt+Esc) to see the reason for the abend and to see which instruction caused the error. (You will have access to all of the debugger's commands.)**
4. **After you have determined the problem, toggle the developer option to "OFF" using the .T command.**
5. **Issue the G command to reexecute the instruction that caused the server to abend. The OS will then quarantine the NLM.**
6. **Unload the NLM from the system console by using the UNLOAD command. (You do not need to reboot the server.)**
7. **Reset the developer option to ON at the system console by entering the "Set developer option = on" command.**
8. **Correct the errors in your NLM and then reload it for additional testing.**

Parent Topic: DOMAIN.NLM

Using Error Watch

Use Error Watch to have NLMDebug watch for *errno* and *NetWareErrno* values and break when they are encountered. A stack walk to the function that returned the error is displayed.

To use Error Watch, do the following:

1. **From the NLMDebug main menu, select Error Watch and press Enter.**

   The Error Watch screen is displayed.

   You can have the NLMDebug watch for *errno* and\or *NetWareErrno* values and break when an error is encountered.

2. **Select Break on `errno': and type Y or N depending on whether or not you want the debugger to break when an *errno* value is returned.**

3. **Select Break on `NWErrno': and type Y or N depending on whether or not you want the debugger to break when a *NetWareErrno* value is returned.**

4. **Press F10 to run Error Watch.**

   A stack walk to the error is displayed on the console screen.

Parent Topic: Setting Up and Using NLMDebug

Using the Function-Call Profiler

Use the Function-Call Profiler to display the functions called by your NLM™ application and the number of times the functions are called.

To use Function-Call Profiler, do the following:

1. **From the NLMDebug main menu, select Function-Call Profiler and press Enter.**

   The Function-Call Profiler screen is displayed.

2. **Enter the name of the NLM that you want to profile in the Count Function Calls for: edit box.**

3. **Select Count Calls in this NLM, type Y or N and press Enter.**

   If you want to see a list of the functions used that are not CLib, you must have previously defined them by calling **NWBumpFunctionCount** for each function you want to see. Type Y if you want to see these functions, if not type N.

4. **Select Count Server-Library APIs, type Y or N and press Enter.**

   If you want to see a list of the functions used that reside in CLib type Y; otherwise type N.

5. **Select Sort Output by... and press Enter.**

   The Sort Output by selection box is displayed.

6. **Select the desired output and press Enter.**

   You can sort the output list in one of three ways: in alphabetical order, in ascending frequency of calls to a function or by descending frequency of calls to a function. You can change the way the output is sorted at any time. Press **Esc** to return to the Function-Call Profiler menu.

7. **Press F10 to start the profile.**

8. **Press the Spacebar to start and stop the profile until it is finished.**

Parent Topic: Setting Up and Using NLMDebug

Using the NCP Debugger

Use the NCP™ Debugger to watch request and reply packet activity for an NLM™ application. You can set the NCP Debugger to report activity for every packet, only when there are errors, or when specific functions and/or subfunctions are detected.

To use NCP Debugger, do the following:

1. **From the NLMDebug main menu select NCP Debugger and press Enter.**

   The NCP Debugger Setup Menu is displayed.

2. **Enter the server name your NLM is communicating with. If left blank, all servers are monitored.**

3. **Set the conditions you want for monitoring packets, then pressEnter.**

4. **Press F10.**

   The NCP Debugger Setup Menu closes and the Waiting for Beginning Request Packet and Waiting for Beginning Reply Packet windows display. Run your NLM and on this screen watch the packet activity based on the set conditions. To view the buffer information for either the Request Packet Activity or the Reply Packet Activity, highlight the screen you are interested in with the **PageUp** and **PageDown** keys. Then scroll through the information in the buffer of the highlighted

screen using the **Up-** and **Down-arrow** keys.

5. **Press F4 from the NCP Debugger window to display the NCP Debugger Setup Menu again.**

   You can make any changes to the NCP Debugger Setup at any time, even while your NLM is being monitored.

Related Topics

Breaking on a Connection Number

Breaking on a Function Code

Breaking on a Subfunction Code

Breaking on an Error

Delaying between Packets

Breaking on Every Packet

Viewing Packet Header Information

Viewing Stack Walk Information

Viewing NCP Error Status

Parent Topic: Setting Up and Using NLMDebug

Using the Process Timer

Use the NLM™ Process Timer to determine where your NLM needs to relinquish the CPU.

To use the NLM Process Timer, do the following:

1. **From the NLMDebug main menu, select Process Timer and press Enter.**

   The NLM Process Time screen is displayed.

2. **Enter the maximum time slice in milliseconds and press Enter.**

   The NLM Process Stack Trace screen is displayed. Whenever your NLM takes longer than the entered number of milliseconds to relinquish the CPU, a stack walk to the problem is displayed on this screen.

3. **To close the NLM Process Stack Trace screen, press Esc.**

Parent Topic: Setting Up and Using NLMDebug

Viewing NCP Error Status

1. **To view the NCP™ Error Status for a specific error, press F5 from the NCP Debugger main screen.**

   The Error Status screen opens, displaying information about the detected error.

   The Return Code field displays the error string associated with the error.

   The Connect Status information field tells you whether the connection is OK or Bad.

   The Reply ECB Status field tells you whether there are buffer overflows or not.

Parent Topic: Using the NCP Debugger

Viewing Packet Header Information

1. **To view the packet header information, press F3 from the NCP™ Debugger main screen.**

   The Packet Header Information screen is displayed.

   The following describes the fields displayed on this screen.

   Checksum displays the packet checksum.

   Packet Length displays the entire length of the packet, including the length of the header and the length of the data.

   Transport Control displays the packet transport control. TheIPX™ protocol always sets this field to zero before sending the packet.

   Destination Socket displays the destination socket address. The following sockets are reserved for use by theNetWare® OS:

Table auto. NetWare Reserved Sockets

451	File service packet
452	Service advertising packet
453	Routing information packet
455	NetBIOS packet
456	Diagnostic packet

Therefore, NetWare servers accept requests addressed to socket 0x0451.

From displays the physical address of the packet source node.

Source Socket displays the socket address of the packet source node.

NCP Type displays the packet NCP type.

Sequence Number displays the packet sequence number.

Low Slot displays the low slot connection number.

Task displays the client task making the service request.

High Slot displays the high slot connection number.

Return Code displays the packet return code as a hex value.

Connection Status displays the packet connection status.

Parent Topic: Using the NCP Debugger

Viewing Stack Walk Information

1. **To view the stack walk information, press F2 from the NCP™ Debugger main screen.**

   The Stack Walk Information screen opens and displays the stack walk information for the function generating the received packet.

Parent Topic: Using the NCP Debugger

Watching CLIB Context

Use CLIB context to watch the relationship between threads and thread groups in your NLM™ application in real time.

To use CLib Context, do the following:

1. **From the NLMDebug main menu, select CLib Context and press Enter.**
2. **Enter the name of the NLM whose context you want NLMDebug to inspect.**
3. **Press F10.**

   NLM control structure information is displayed.

   If the NLM is not currently loaded, press **Alt+Esc** to switch to the

system console screen and type `LOAD <nlm name>` at the system console command prompt. Return to NLMDebug using **Alt+Esc**.

4. **Select options as follows:**

   To view information about fields in the screen, select the fields for which you want more information and press **Enter**.

   The function names that manipulate the information in the selected field are also displayed.

   To view the memory addresses of the thread groups belonging to the NLM, select Thread Groups and press **Enter**.

   To view information for a specific thread group, select the thread group for which you want information and press **Enter**.

   To display the memory address of all threads governed by a thread group, select Threads and press **Enter**.

For more information about context, thread groups, and threads, seeNLM Code Development.

Parent Topic: Setting Up and Using NLMDebug

NLM Development Tools: Concepts

Additional Debugger Commands

Loading DOMAIN.NLM adds additional commands to theNetWare® Internal Debugger. These commands are listed in NLM Debuggers. You can also display a help screen that lists the commands by entering "/h" while in the NetWare Internal Debugger.

Parent Topic: DOMAIN.NLM

Allow Invalid Pointers

(NetWare® 4.x only) This parameter determines whether invalid pointers are allowed to cause a nonexistent page to be mapped in with only one notification. A nonexistent page is one whose address is above the physical memory of the server.

If set to ON, the OS maps the nonexistent page into the page table and sends a notification to the system console. The OS does not unmap the page after it is accessed so subsequent accesses to the page do not cause a notification. This means that in addition to the invalid pointer that caused the page to be mapped in, other invalid pointers that access the same page will also not cause a notification.

If set to OFF, an attempt to read a nonexistent memory page results in a read page fault. If the "developer option" flag is set to OFF, theNLM™ application is quarantined. If the "developer option" is set to ON, the server abends and the NetWare Internal Debugger screen appears.

Default Setting: ON

Recommended Setting: OFF

Parent Topic: Setting Server Parameters

Auto-Save All Settings

When Auto-Save All Settings is set to Yes, NLMDebug saves all settings used in the session.

Parent Topic: Changing Debug Settings

AUTOUNLOAD

```
AUTOUNLOAD
```

This directive sets a flag in the flags field of the NLM™ header indicating that this NLM is to be unloaded when none of its entry points are in use.

The flags are defined as shown in the Current Features table (see FLAG_ON, FLAG_OFF).

Parent Topic: Linker Directives and Options

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

||	Logical OR	11

Parent Topic: Specifying Expressions

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
```

Parent Topic: Debugger Commands

CHECK

```
CHECK check procedure name
```

This directive specifies the name of a function in the NLM™ application to be executed when the console operator attempts to unload the NLM using the **UNLOAD** console command. Do not confuse this operation with registering a function by calling **AtUnload** or **atexit**, as explained in Following Exit Steps.

This function returns 0 if the NLM can be unloaded. If a nonzero value is returned, the NLM does not want to be unloaded.

WLINK: OPTION CHECK

Parent Topic: Linker Directives and Options

CODESTART

```
CODESTART address
```

This directive specifies an offset to be added to each code symbol offset in the map file. This directive allows the developer to create a map file that compares well with the values displayed by the debugger.

See also MAP and FULLMAP.

Parent Topic: Linker Directives and Options

Commands Available with DOMAIN.NLM

The following table lists commands added when DOMAIN.NLM is loaded.

Command	Description
/a filename	Write used symbols to DOS file named filename.
/c	Close screen to screen dump file (in DOS).
/d	Displays the names of all domains and lists the NLM™ applications that are running in them. This is the same as using the "DOMAIN" command at the system console.
/d=domain_name	Sets the domain to the specified domain. The currently available domains are OS and OS_PROTECTED. This command is similar to using the "DOMAIN=OS" or "DOMAIN=OS_PROTECTED" commands at the system console.
/f	Displays outstanding RPC stack frames.
/g	Displays the Global Descriptor Table
/h	Displays the help screen for the DOMAIN.NLM

	debugger commands.
/i	Displays the Interrupt Descriptor Table.
/m filename	Writes symbols missing RPC definitions to a DOS file **filename**.
/o filename	Opens a screen dump file (in DOS) with the name of filename.
/o	Dumps the current screen to the screen dump file specified with the "/o filename" command.
/q symbol	Displays the RPC data for symbol.
/r	Displays the RPC Pseudo Registers.
/s filename	Writes the symbol lists to the DOS file filename.
/t	Displays the current RPC Stack Frame.
/u	Disassembles the current RPC layer.
/x+	Turns on all RPC Interpreter Single Step.
/y-	Turns off IBT break points.
/y+	Turns on IBT break points.

Parent Topic: Debugger Commands

Compilers for NLM Applications

The WATCOM* C/386 and C/C++[32] compilers are cross-compilers that run under DOS or OS/2* 2.x and produce object files for other operating systems, such as the NetWare® OS. They translate your standard ANSI C program from source files and header files into 32-bit machine language instructions.

The WATCOM C/386 and C/C++[32] compilers are command-line oriented and allow you to specify a wide range of parameters, such as:

Whether to include debug information in the object file

Object filename (if different from the default)

Pathnames for include files

Amount and type of optimization to perform

In addition, the compiler includes a set of `#pragma` directives you can use to customize the code generation process.

The compiler also provides switches for specifying whether to use stack-based or register-based parameter passing. The NetWare API expects stack-based parameter passing. However, you can write "mixed mode"

NLM applications that use register-based parameter passing except when calling the NetWare API. Using register-based parameter passing generates smaller, faster code.

For more specific information about the WATCOM C/386 and C/C++[32] compilers, and about other compilers, seeNLM Compilers.

Parent Topic: NLM Development Tools Overview

COPYRIGHT

```
COPYRIGHT "copyright string"
```

The copyright string is displayed on the console screen when theNLM™ is loaded. If this option is not used, no copyright information is displayed.

If this directive is used, but no string is specified, NLMLINK uses the Novell® copyright notice.

WLINK: OPTION COPYRIGHT

Parent Topic: Linker Directives and Options

CUSTOM

```
CUSTOM custom data filename
```

This directive allows the developer to specify a custom data file for the use of the NLM™.

WLINK: OPTION CUSTOM

Parent Topic: Linker Directives and Options

DATASTART

This directive causes the .MAP file listing of data symbols to show locations with a fixed offset (it has no effect on the .NLM file). This command, with CODESTART, is used by developers to make map information correspond to addresses where the OS is loaded, avoiding computation during debugging.

Parent Topic: Linker Directives and Options

DATE

```
DATE month, day, year
```

This directive dates the NLM™ application. The *month, day,* and *year* parameters are entered with any whitespace character separating them.

The year must be expressed as the four-digit number. NLMLINK returns an error if the year is less than 1900 or greater than 3000. The month must be between 1 and 12. The day must be between 1 and 31.

Parent Topic: Linker Directives and Options

DEBUG

```
DEBUG
```

This directive instructs the linker to generate debugging information in the executable file. NLMLINK creates debugging records for the NetWare® Internal Debugger. The debug records contain all internal data and procedure names.

WLINK: DEBUG

Also generates debug information for the WVIDEO debugger.

Parent Topic: Linker Directives and Options

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 commands in the NetWare Internal Debugger:

HE---Help on expressions

HB---Help on breakpoints

H---General Help

.H---Help with the supplementary commands

/h---Help with domain commands

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

Commands Available with DOMAIN.NLM

Supplementary Commands

Breakpoint Commands

General Debugger Commands

SFT III Debugger Commands

Parent Topic: Using the NetWare Internal Debugger

Debugger Overview

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

The WATCOM* linker (WLINK) and the Novell® linker (NLMLINK) can be used to include debugging information in an executable file. Additionally, the linkers can be used to generate a map file, which is a memory map of your program.

The WATCOM Visual Interactive Debugging Execution Overseer (WVIDEO) is a source-level debugger that is used remotely from another DOS machine. The WVIDEO debugger supports interaction with a keyboard or mouse.

The NetWare® 3.x and 4.x OS's have a built-in command line debugger that performs symbolic debugging.

The Rdebug tool is a powerful Windows 3.x test and development tool that can be used for NLM software development, debugging, and integration.

Related Topics

Linking Debug Information with WLINK

Linking Debug Information with NLMLINK

Using the WVIDEO Debugger

Using the NetWare Internal Debugger

Setting Breakpoints

Specifying Expressions

Debuggers for NLM Applications

The tools that are available for debugging your NLM™ applications are:

WATCOM* Visual Interactive Debugging Execution Overseer (WVIDEO)

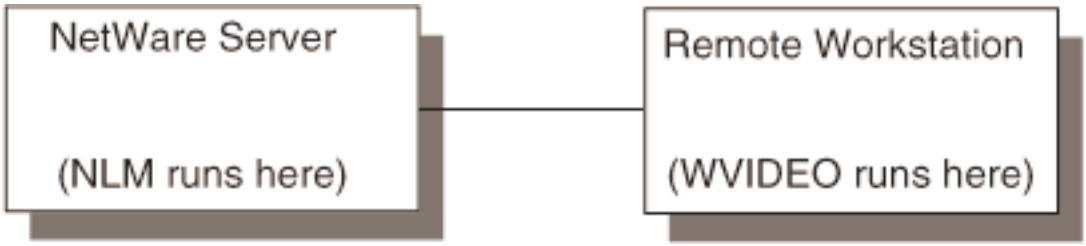
NetWare® Internal Debugger

WVIDEO Debugger

The WATCOM VIDEO debugger (WVIDEO) is a source-level debugger with a window-based user interface. The debugger can be used with a keyboard or mouse.

WVIDEO requires the use of remote debugging, where you run the test application from one workstation and the debugger from another, as shown in the following figure.

Figure 17. WVIDEO Remote Debugging



For an introduction to using the WATCOM VIDEO debugger specifically for debugging NLM applications, see NLM Debuggers.

For complete information on the WATCOM VIDEO debugger screens and usage, see the WATCOM manuals.

NetWare Internal Debugger

The NetWare Internal Debugger is an assembly language debugger that is built into the NetWare 3.x and 4.x OS's. This debugger is a command-line debugger that does not display source code. However, you can use the

WATCOM utility WDISASM with the "/s /l" options to create a list file containing C source interspersed throughout the assembly code. This is very helpful for debugging in 80386 assembly. 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 NLM applications. It includes a set of supplementary commands that are customized for NLM applications, 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 NLM applications concurrently.

To access the NetWare Internal Debugger, use one of these three methods:

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.

For information about using the NetWare Internal Debugger, see NLM Debuggers.

Parent Topic:

NLM Development Tools Overview

DESCRIPTION

```
DESCRIPTION "NLM description string"
```

The description string is displayed on the console screen when theNLM™ application is loaded. Typically, the description contains the name of the NLM. The description string can be up to 127 characters.

This directive is required. If no description is specified, an error occurs. Further, this directive should not be repeated.

Parent Topic: Linker Directives and Options

Developer Option

If set to ON, the settings for the following developer flags are enabled:

Read fault emulation

Read fault notification

Write fault emulation

Write fault notification

This means that errors cause the server to abend and call the debugger.

If set to OFF, the OS quarantines NLM™ applications that have unhandled exceptions.

Default Setting: ON

Recommended Setting: ON (if you want the server to abend or to call the debugger when an exception happens.)

OFF (if you want the offending NLM to be quarantined, and to have the server keep running.)

> **NOTE:** The Developer Option can be toggled within the NetWare Internal Debugger by using the **.T** command.

Parent Topic: Setting Server Parameters

Displaying Domain Information

To verify what domain is the current domain, type the following command:

```
DOMAIN
```

If the loading sequence in Loading Domain NLM is followed, the current domain is OS_PROTECTED, and the screen displays output similar to the following:

```
Domain "OS_PROTECTED" in ring 3 is the current domain.
Domain "OS" in ring 0.
   NLM NetWare C NLM Runtime Library V4.0
   NLM NetWare STREAMS
   NLM NetWare OS Loader
   NLM NetWare v4.0 ISA Device Driver
   NLM NetWare Bindery Name Service
   NLM NetWare 386 Domain Support NLM
   NLM NetWare Server Operating System
```

You can switch to the OS domain by typing the following command:

```
DOMAIN=OS
```

If no other modules have been loaded, the output is something like the following:

```
Domain "OS" in ring 0 is the current domain.
   NLM NetWare C NLM Runtime Library V4.0
   NLM NetWare STREAMS
   NLM NetWare OS Loader
   NLM NetWare v4.0 ISA Device Driver
   NLM NetWare Bindery Name Service
   NLM NetWare 386 Domain Support NLM
   NLM NetWare Server Operating System
Domain "OS_PROTECTED" in ring 3.
```

As more modules are loaded, their names are displayed under the domain that they were loaded in.

Parent Topic: DOMAIN.NLM

DOMAIN.NLM

The NetWare® 4.x OS provides memory protection in the form of a new page-mapping system and with DOMAIN.NLM. The new page-mapping system marks certain pages, such as the zero page where DOS resides, off-limits to NLM™ applications. NLM applications accessing these protected areas generate page faults.

DOMAIN.NLM is used to create domains, which are areas of memory that NLM applications run in. The version of DOMAIN.NLM that ships with the NetWare 4.x OS provides two domains: OS and OS_PROTECTED. The operating system is not protected from NLM applications running in the OS domain, but it is protected from NLM applications running in the OS_PROTECTED domain. The OS domain runs in ring 0, and the OS_PROTECTED domain can be run in rings 1, 2, or 3. Ring 3 is the default.

DOMAIN.NLM is provided as a tool for developers and system administrators, but its use is not mandatory. A developer can use DOMAIN.NLM to create the OS_PROTECTED domain and develop NLM applications in it, catching read and write requests to invalid memory locations. A system administrator can use DOMAIN.NLM to create the OS_PROTECTED domain and run new NLM applications in that domain, until the administrator feels confident that the NLM is well behaved. Then the administrator can move the NLM into the OS domain for faster program execution.

Depending upon the switches that are set, a read or write to a bad address either causes the server to abend, or DOMAIN.NLM stops (quarantines) the offending NLM without bringing down the server. Then the NLM can be

unloaded and restarted. These options are discussed later.

In the OS domain there is no memory protection. Running NLM applications in the OS_PROTECTED domain protects the operating system from modules trying to access invalid memory addresses, such as the space used by the OS.

You should load your NLM applications in the OS_PROTECTED domain as you develop and test them. This way you can catch more memory violations than if you test in the OS domain only.

Another advantage to testing in the OS_PROTECTED domain is that in most cases you do not need to reboot your server when an NLM violates memory boundaries. Instead, the OS quarantines the NLM and you can unload it.

For more information about using DOMAIN.NLM as a development tool, see Memory Protection.

The commands listed in the following table are only available when DOMAIN.NLM is loaded:

Domain Command	Description
domain	Displays information about the NLM applications that are loaded and shows what domain they are loaded in. It also shows which domain is the current domain.
domain= *domain_name*	Sets the current domain to the specified domain. Currently, there are two domains: OS and OS_PROTECTED.
domain ring 1-3	Sets the protection ring that the OS_PROTECTED domain runs in. Ring 3 is the default for startup.
domain help	Displays a help screen, showing available commands for DOMAIN.NLM.
domain show [off | on]	Determines whether RPC symbols are displayed as the exporting NLM is loaded. The command without the options show whether the option is toggled on or off.

Related Topics

Loading Domain NLM

Memory Protection Commands

Displaying Domain Information

Additional Debugger Commands

Setting Server Parameters

Using DOMAIN.NLM as a Development Tool

EXIT

```
EXIT exit procedure name
```

This directive specifies the name of a symbol in the NLM™ application where execution should stop. This procedure makes sure that all resources have been released and all threads terminated before the NLM unloads.

If this option is not used, _Stop is the default.

WLINK: OPTION EXIT

Parent Topic: Linker Directives and Options

EXPORT

```
EXPORT symbol list

EXPORT @symbol list file
```

> **IMPORTANT:** This describes how EXPORT was used in NetWare® 3.x and 4.0. For current information, see Imported and Exported Symbols.

EXPORT is followed by an export list. EXPORT must appear in the left-most column. Symbols must not appear in the first column. Symbols can be indented with a tab or spaces, and can be separated by commas or spaces. A file containing a list of symbols can be indicated by @ followed by the filename.

WLINK: EXPORT

Parent Topic: Imported and Exported Symbols

FLAG_ON, FLAG_OFF

```
FLAG_ON flag number

FLAG_OFF flag number
```

These directives were added to allow for setting bits in the header without

requiring the linker to add new directives. The feature number is a decimal number that is used to form a bit mask. See the following table for a list of current flags and their flag numbers.

Table auto. Current Features

Flag Number	Mask Bit	Flag/Linker Directive	Description
1	0x00000001	REENTRANT	The module is reentrant. That is, if the NLM™ application is loaded twice, the code in the server's memory is reused.
2	0x00000002	MULTIPLE	The module can be loaded more than once by a **LOAD** console command.
4	0x00000004	SYNCHRONIZE	The load process goes to sleep until the NLM calls **SynchronizeStart**. This prevents any other console commands (particularly LOAD) from being processed while the NLM is being loaded.
8	0x00000008	PSEUDOPREEMPTION	The OS forces the NLM to relinquish control if the NLM does not do so on its own often enough.
16	0x00000010	OS_DOMAIN	The module must be loaded in the OS domain.
64	0x00000040	AUTOUNLOAD	Causes the module to be automatically unloaded when none of its entry points are in use.

FLAG_ON turns the bits on; FLAG_OFF turns the bits off.

FLAG_ON and FLAG_OFF can appear more than once in a .DEF file. The result is cumulative.

The following directives are also used to set feature flags: AUTOUNLOAD, MULTIPLE, OS_DOMAIN, PSEUDOPREEMPTION, REENTRANT, and SYNCHRONIZE.

Parent Topic: Linker Directives and Options

FORMAT (W)

See NLMLINK directives DESCRIPTION and TYPE. This directive specifies the format (NLM, LAN, DSK, NAM, and so on) of the executable file to be generated. This directive also specifies the name to be displayed when the NLM™ application is loaded.

Parent Topic: Linker Directives and Options

FULLMAP

```
FULLMAP map filename
```

This directive is the same as MAP, except that it adds relocation and fixup data to the map file that is not included by MAP (see MAP).

Parent Topic: Linker Directives and Options

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 NLM™ applications 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:

```
?
```

Parent Topic: Debugger Commands

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.

Parent Topic: Specifying Expressions

HELP

```
HELP help file path
```

The help file path specifies the path to an internationalized help file that contains the default help screens for the NLM™ application.

WLINK: OPTION HELP

Parent Topic: Linker Directives and Options

IMPORT

```
IMPORT symbol list

IMPORT @symbol list file
```

> **IMPORTANT:** This describes how IMPORT was used in NetWare® 3.x and 4.0. For current information, see Imported and Exported Symbols.

IMPORT is followed by an import list. IMPORT must appear in the left-most column. Symbols must not appear in the first column. Symbols may be indented with a tab or spaces, and can be separated by commas or spaces. A file containing a list of symbols can be indicated by @ followed by the filename.

WLINK: IMPORT

Parent Topic: Imported and Exported Symbols

Imported and Exported Symbols

Imported and exported symbols are used by the loader to match functions called within the NLM™ that are exported by a different NLM. For information about how imported and exported symbols were handled in NetWare® 3.x and 4.0, see IMPORT and EXPORT.

To enable multiple compilers and libraries to be used (that is, to allow for multiple versions of a function to exist in the server symbol table simultaneously), we have modified the import and export lists to include prefixes. The prefix is a code string representing the publisher of the library. See below.

Prefixes for Imports and Exports: When the directive IMPORT or EXPORT is encountered, the *currentPrefix* is set to NULL. When a string in parentheses occurs after the IMPORT or EXPORT directive, that string becomes the *currentPrefix*. For example, a developer might specify the following:

```
IMPORT symbol1
   (Prefix)
   symbol2
   symbol3
```

The prefix is appended to the beginning of the symbol name with the @ character separating the prefix from the name. The names of the prefixed symbols are represented as Prefix@symbol2 and Prefix@symbol3 in the NLM.

Prefix represents a library publisher. For example, the prefix representing libraries published by Novell might be NOVL. If no prefix is specified, no prefix or @ character is attached to the name of the function.

IMPORTANT: The prefix must also be appended to appropriate debug record names.

NOTE: New prefixes are registered by calling 1-800-NETWARE (1-800-638-9273).

Parent Topic: Linker Directives and Options

INPUT

```
INPUT object list

INPUT @object list file
```

This directive lists the object files that are to be linked. If the object files are listed within a file, the object files should appear in standard list format (see Linker Definition Syntax). If no extension is given, then either .OBJ or the default for the environment should be assumed.

WLINK: FILE

Parent Topic: Linker Directives and Options

LIBRARY (W)

Specifies a user-defined library of functions.

Parent Topic: Linker Directives and Options

Linker Definition Syntax

Any word beginning in the left-most column is considered to be a directive (unless it begins with #, in which case it is considered to be a comment).

A directive can be followed by a list of parameters. These parameters can be separated by commas or any whitespace character (such as spaces, tabs, and carriage returns), but they must not begin in the left-most column. A file containing a list of parameters is acceptable in most cases. The filename is preceded by @. The parameters in this file must not begin in the left-most column.

For example:

```
INPUT input1.obj
      input2.obj,
      @objlist.lst
```

NOTE: If you use commas, they do not have to be used consistently.

Parent Topic: Linker Directives and Options

Linker Directives and Options

Linker directives and options tell the linker how to create your program. You can use directives and options when generating NLM™ applications. Directives and options can be given at the command line, or they can be placed in a directive file.

Linker Directives: Directives appear at the left-most column of the file. Parameters for directives can appear anywhere but the left-most column. Parameters can be separated by commas, whitespace, carriage returns or tabs (which must not start in the left-most column). NLMLINK provides the following directives:

See the following topics for general information about NLMLINK:

Symbols Used by NLMLINK

Linker Definition Syntax

Imported and Exported Symbols

NOTE: Options for WLINK are preceded by the directive OPTION. NLMLINK does not use the OPTION directive before its options.

The following topics describe the directives and options for NLMLINK with comparisons to WLINK (options exclusive to WLINK are marked with "(W)").

NOTE: Do not rely on this information as a complete reference for all WLINK directives. Check the WLINK documentation for a complete list of directives.

AUTOUNLOAD

OPTION CASEEXACT (W)

CHECK

CODESTART

COPYRIGHT

CUSTOM

DATASTART

DATE

DEBUG

DESCRIPTION

OPTION DOSSEG (W)

EXIT

EXPORT

See Imported and Exported Symbols for information about this directive.

FLAG_ON, FLAG_OFF

FORMAT (W)

FULLMAP

HELP

IMPORT

See Imported and Exported Symbols for information about this directive.

INPUT

LIBRARY (W)

MAP

MESSAGES

MODTRACE (W)

MODULE

MULTIPLE

NAMELEN

OPTION NODEFAULTLIBS (W)

OS_DOMAIN

OUTPUT

PATH

PSEUDOPREEMPTION

OPTION QUIET (W)

REENTRANT

SCREENNAME

SHARELIB

STACK, STACKSIZE

STAMPEDDATA

START

OPTION SYMFILE (W)

OPTION SYMTRACE (W)

SYNCHRONIZE

THREADNAME

OPTION UNDEFSOK (W)

TYPE

VERBOSE

VERSION

XDCDATA

Linkers for NLM Applications

Because NLM™ applications use dynamically-linked libraries, they do not resolve external symbols (global variables and function names) at link time. Rather, they produce an executable file that contains references to external symbols that are resolved, by the NetWare® loader, at load time. This differs from operating systems such as DOS, in which external symbols are resolved at link time, or OS/2*, in which external symbols are resolved the first time the call is made in the program.

NLMLINK

NLMLINK is a linker that Novell® developed specifically for linking NLM applications. Novell supplies these versions of the linker: NLMLINKP, NLMLINKR and NLMLINKX. NLMLINKP runs in protected mode and NLMLINKR runs in real mode.

WLINK

WLINK is the WATCOM* linker. One of the executable formats it produces is that of NLM applications. You must use WLINK if you are going to use the WATCOM debugger WVIDEO, or if you require the need for linking static libraries.

For more information about NLMLINK and WLINK, see NLM Linkers.

Parent Topic:

NLM Development Tools Overview

Linking Debug Information with NLMLINK

You can specify the DEBUG directive in a linker directive file to generate debugging information in the executable file. DEBUG is available with NLMLINK. It generates debugging information for the NetWare® Internal Debugger.

To generate a map file, specify the MAP option with the OPTION directive. The map file specifies the relative location of all global symbols in your program and contains the size of your program.

For more information on how to use these directives and the MAP option, see NLM Linkers.

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.

To generate a map file, specify the MAP option with the OPTION directive. The map file specifies the relative location of all global symbols in your program and contains the size of your program.

Additionally, with WLINK, you can use the MODTRACE (W) directive to include in the map file a list of all modules that reference the symbols defined in the specified modules. Use the OPTION SYMTRACE (W) directive to list all the modules that reference the specified symbols. For more information on how to use these directives and the MAP option, see the WATCOM* manuals.

Loading Domain NLM

DOMAIN.NLM **must** be the first NLM™ application loaded. It is loaded with the following command:

```
LOAD DOMAIN
```

When DOMAIN.NLM loads, it creates the OS and the OS_PROTECTED domains. By default, the OS domain runs in ring 0, and the OS_PROTECTED domain runs in ring 3. The OS domain is the current domain.

All NLM applications that are loaded are loaded in the domain that is the current domain when they are loaded. You should load the NetWare API files in the OS domain and your NLM applications in the OS_PROTECTED domain. You might want to place the following commands in your AUTOEXEC.NCF file:

```
LOAD DOMAIN
; the current domain is OS
LOAD CLIB
; now switch to the OS_PROTECTED domain
domain = os_protected
LOAD MYNLM
```

NOTE: CLIB always loads in the OS domain, no matter which domain is the current domain.

Parent Topic: DOMAIN.NLM

Log Output to Path

If you want to save stack traces and other information produced by NLMDebug, enter a path where you want the information saved.

Parent Topic: Changing Debug Settings

MAP

```
MAP map file name
```

This directive sets an internal flag requiring the linker to create a map file with the specified name. If no filename is specified, the module name specified with the OUTPUT directive is used, but given a .MAP extension.

The FULLMAP directive produces a map file that is identical to that produced by MAP, except that it includes relocation fixup information.

WLINK: OPTION MAP

Parent Topic: Linker Directives and Options

Memory Checking

When the Memory Checking option is set to Yes, NLMDebug checks for memory overwrites on all calls to memory functions. A stack walk to the location of the memory overwrites is displayed on the console screen. If you want to force checking at a certain point in your code, call **free** (NULL).

Parent Topic: Changing Debug Settings

Memory Protection Commands

There are two types of commands associated with memory protection: those that are available after DOMAIN.NLM is loaded and those available regardless of whether or not DOMAIN.NLM is loaded.

Parent Topic: DOMAIN.NLM

MESSAGES

```
MESSAGES message file path
```

This directive specifies the file path to an internationalized message file that contains the default messages for the NLM™ application.

WLINK: OPTION MESSAGES

Parent Topic: Linker Directives and Options

MODTRACE (W)

Instructs the linker to print a list of all modules that reference symbols defined in the specified modules.

Parent Topic: Linker Directives and Options

MODULE

```
MODULE autoload NLM list
```

This directive specifies NLM™ applications that must be loaded before this NLM is loaded. These modules will be 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.

WLINK: MODULE

Parent Topic: Linker Directives and Options

MULTIPLE

```
MULTIPLE
```

This directive sets a flag in the flags field of 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.

The flags are defined as shown in the Current Features table (see FLAG_ON, FLAG_OFF).

WLINK: OPTION MULTILOAD

Parent Topic: Linker Directives and Options

NAMELEN

```
NAMELEN number
```

This directive sets a maximum name length for comparisons when resolving entry points, where *number* is the maximum name length to be used. The default value for *number* is 31.

This directive is useful when the maximum name length for assemblers or compilers differ. For example, an assembler might limit names to 31 characters in object files, but a C compiler might allow 32 characters. This means that only 31 characters should be significant if you are linking between the assembly and C object files.

WLINK: OPTION NAMELEN

Parent Topic: Linker Directives and Options

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 NLM applications that run on the NetWare® 4.x OS.

When the NetWare 4.x OS loads an NLM, the OS checks to see if the NLM is in compressed format. If the NLM has been compressed, the OS automatically decompresses it as the OS loads the NLM into memory.

> **NOTE:** You should not compress NLM applications that will be loaded on the NetWare 3.x OS because the 3.x OS cannot uncompress them

For more information about NLMPACK, see NLMPACK and UNPACK.

Parent Topic: NLM Development Tools Overview

NLM Development Tools Overview

The following topics provide an overview of the tools that ship with the NetWare® SDK. For a more in-depth look at the tools, see other topics listed in NLM Development Tools: Guides.

The following types of tools ship with the NetWare SDK:

Compilers for NLM Applications

Linkers for NLM Applications

NLM Make Utilities

Debuggers for NLM Applications

NLM Memory Protection Tools

NLM Execution Profilers

NLM Message Tools for Internationalization

NLM Compression Tools

NLM Testing Tools

NOTE: The WATCOM* tools can be purchased with this SDK or separately.

NLM Execution Profilers

The following execution profilers are available with this SDK:

Procedure Coverage Logger (PCL)

WATCOM* execution profiler (WPROF)

PCL

PCL.NLM is one of the tools provided in theNLM™ Testing Tools SDK. It does the following:

Dynamically monitors which public symbols have been called, including initialization and cleanup routines

Dynamically tracks how often selected sections are called, helping you identify critical paths for optimizing execution speed

Dynamically monitors all NetWare resources used by your product

Generates ASCII text reports that can be saved to either a NetWare® or a DOS partition on the file server

Simultaneously monitors up to 13 NLM applications

For more information about PCL, see the *PCL User's Manual* that ships with the NLM Testing Tools SDK.

WPROF

WPROF is a performance development tool that allows you to identify heavily executed regions of code and concentrate efforts on improving the productivity of these regions. WPROF is used to display execution information obtained by WSAMPLE.NLM. (WPROF and WSAMPLE.NLM ship with the WATCOM compiler package.)

Analysis using WPROF is performed in two stages:

In the first phase of the analysis process, WSAMPLE.NLM runs on the server and sets up a periodic interrupt. It then records the location of the instruction pointer each time the program's execution is interrupted. These samples of the execution provide a statistical measure of the heavily executed portions of the program.

In the second phase, WPROF runs on a workstation and displays, in graphical form, the execution samples saved by WSAMPLE.NLM. The interactive nature of WPROF facilitates analysis of the program's execution, and enables you to experiment with the effects of different coding strategies in order to achieve optimal code performance.

For more information about the execution sampler and profiler, see the WATCOM documentation.

Parent Topic:

NLM Development Tools Overview

NLM Make Utilities

The WATCOM* make utility is WMAKE. The makefiles associated with the examples that ship with this SDK 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 SDK ships with the QMK386 utility. QMK386 is located in the TOOLS directory.

To display the QMK386 options, type:

```
QMK386 ?
```

QMK386 gives you the option of creating makefiles that use WLINK or makefiles that use NLMLINK. You can also customize the makefiles after QMK386 creates them.

Parent Topic: NLM Development Tools Overview

NLM Memory Protection Tools

Two tools are available to help you discover addressing errors in the NLM™ applications you are developing. These tools are:

PROTECT.NLM

DOMAIN.NLM

Parent Topic:

NLM Development Tools Overview

NLM Message Tools for Internationalization

The NetWare® OS has a world market, and so it has been "internationalized" for use with multiple languages. The term used to describe writing software that is easily adaptable to different locales is **enabling**. Enabled software can be adapted without having to recompile and relink the source code.

You should also consider the world market for yourNLM™ applications. You should not limit your NLM applications to one language; you should enable them for multiple languages. For this reason, we ship the Message Tools with this SDK.

The Message Tools are a collection of programs that can be used to extract the messages (text strings that will be displayed) from your source code, so you can send these messages to a translation house. The end results of the Message Tools are message files for multiple languages. Changing the message file changes your module's display for a particular language.

Parent Topic: NLM Development Tools Overview

NLM Testing Tools

The NLM™ Testing Tools SDK, which ships with this SDK, provides the following:

Information about the Novell NLM Certification program

NLM certification tools

Testing and programming hints

Tools for automating testing

For more information about these tools, see the manuals in the NLM Testing Tools SDK.

Parent Topic: NLM Development Tools Overview

NLMLINK Version History

The following information identifies some of the new features added with each version of NLMLINK.

Version 2.82

The description string processes new links in the string. For example, "First Line\nSecond Line" is printed on two lines on the console when theNLM™ application is loaded.

Added support for double quotes within a quoted string using escape quote (\""). For example, "\"A quoted string in quotes.\""

Version 2.81

Changed some WARNING messages to INFORMATION messages so that they would be easier to understand and cause less distraction.

Added checking for zero-length files and improved the error messages associated with them.

Cleaned up several obscure error messages that gave incomplete or misleading information.

Version 2.79

Processes object files produced with theWATCOM* /d2 switch.

Processes object files produced by the Borland* 386 compiler. This is a side effect of fixing several problems with unrecognized object records.

Allows import/export prefix in symbols lists.

Added these new parameters: AUTOUNLOAD, DATASTART, FLAG_ON, FLAG_OFF, NAMELEN, PATH, STAMPEDDATA.

Parent Topic:

Using NLMLINK

NLMPACK and UNPACK

NLMPACK is a compression utility designed to minimize the disk space needed for NLM™ applications that will run on the NetWare® 4.x OS. When the NetWare 4.x OS loads an NLM, it checks in the NLM header to see if the NLM has been compressed. It if has, the OS decompresses it as it is loaded.

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.

> **CAUTION: NLM applications compressed with NLMPACK cannot be decompressed by the NetWare 3.x OS. NLM compression should only be used for NLM applications that will run on the NetWare 4.x OS.**

For the syntax for NLMPACK and UNPACK, see NLMPACK and UNPACK Syntax.

NLMPACK and UNPACK Syntax

Usage of NLMPACK is as follows:

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

source name is the name of the NLM™ application 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*.**

Parent Topic: NLMPACK and UNPACK Information

OPTION CASEEXACT (W)

Tells the linker to respect case when resolving references to global symbols.

Parent Topic: Linker Directives and Options

OPTION DOSSEG (W)

Tells the linker to order segments in a special way.

Parent Topic: Linker Directives and Options

OPTION NODEFAULTLIBS (W)

Instructs the linker to ignore default libraries. This is used because you do not link your object files to any libraries. Instead, the loader resolves addresses with symbols provided by libraryNLM™ applications such as CLIB.NLM.

Parent Topic: Linker Directives and Options

OPTION QUIET (W)

Instructs the linker to suppress informational messages.

Parent Topic: Linker Directives and Options

OPTION SYMFILE (W)

Provides a method for specifying an alternate file for debugging information.

Parent Topic: Linker Directives and Options

OPTION SYMTRACE (W)

Instructs the linker to print a list of all modules that reference the specified

symbols.

Parent Topic: Linker Directives and Options

OPTION UNDEFSOK (W)

Tells the linker to generate an executable file even if undefined symbols are present.

Parent Topic: Linker Directives and Options

OS_DOMAIN

```
OS_DOMAIN
```

This directive sets a flag in the flags field of the NLM™ header indicating that this NLM must be loaded in the OS domain.

The flags are defined as shown in the Current Features table (see FLAG_ON, FLAG_OFF).

Parent Topic: Linker Directives and Options

OUTPUT

```
OUTPUT target file name
```

Provides the name of the output file to the linker. The name is restricted to an 8-character name with a 3-character extension. If no extension is specified, the linker will create an extension according to the NLM™ type that is specified (see TYPE).

> **NOTE:** If the module name does not match the module type, the OS displays a warning message when the module is loaded.

WLINK: OPTION NAME

Parent Topic: Linker Directives and Options

Overview of Rdebug

The Rdebug tool is a powerful Windows* 3.x test and development tool for the NetWare® server. This tool can be used for NLM™ software

development, debugging, and integration.

Rdebug has the following features and benefits:

Windows 3.x human interface.

Source-level debugging. You can set and display break points directly in the source code, browse through the program modules, and examine the function call chain.

Watch expressions. You can select specific program variables to display and watch the values of these variables change as you step through your program.

Breakpoints. You can define a breakpoint, a conditional breakpoint, or a passpoint on a source code line or an assembly instruction.

Stepping. You can execute your program as single assembly language steps, high-level language statement steps, or return steps.

Register access. You can examine and modify processor registers.

Memory access. You can display memory using various type formats including assembler mnemonics.

Symbolic support. You can use symbolics to debug all programs written in either assembly or C languages. You can also display and modify program memory using program symbolics.

Loading NB09. Rdebug supports NB09 as the debug format.

For complete information on the Redebug tool, see the Windows help for Rdebug.

Passing Parameters

The WATCOM* C/386 and C/C++[32] compilers allow you to specify either stack-based or register-based parameter passing.

How parameters are passed to functions is determined by the size (in bytes) of the parameter and where in the parameter list the parameter appears. Depending on the size, parameters are either passed in registers or on the stack. CLIB uses stack-based parameter passing for all of the functions. This means you need to specify the /3s command line switch when building NLM™ applications with the WATCOM compiler. Another option is to #include , then use the default parameter passing mechanism.

The WATCOM compiler no longer supports the predefined constant _NOVELL_. If you have code that depends upon this constant, you can use the WCC386 environment variable to predefine it (for example, c:\< set wcc386 =/3s /d_NOVELL_#1)

Parameters such as structures are almost always passed on the stack, since they are generally too large to fit in registers.

Because parameters are processed from left to right, the first few parameters are likely to be passed in registers (if they can fit), and if the parameter list contains many parameters, the last few parameters are likely to be passed on the stack. The NetWare® API uses stack-based parameter passing. Bytes, words, double words, and pointers are all pushed onto the stack as 4-byte parameters.

It is possible to write an application that uses register-based parameter passing internally and stack-based parameter passing when it calls the NetWare API.

Parent Topic: WATCOM Compilers

PATH

```
PATH Apath[;[Apath]] . . .
```

This directive identifies search paths for files used with the INPUT, CUSTOM, HELP, MESSAGES, SHARELIB, XDCDATA, IMPORT, and EXPORT directives. The *Apath* parameter is a string suitable for prepending to any filename to create a complete DOS path. Therefore, *Apath* must end with "\". The current directory is not searched unless it is specified as *Apath*, which can be done by including a NULL path (terminated by a semicolon), or the specifier ".\". For example,

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

and

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

both search the local directory and then search the obj directory one level up.

The default value for PATH is the current directory. PATH can be repeated and is in effect until PATH occurs again.

Any file that is specified with a path component only searches that specified path. For example, if a path is specified as follows:

```
PATH ..\english\
```

The directive

```
MESSAGES ..\french\msgfile
```

does not search ..\english for msgfile.

To restore the default (search the current directory), simply use PATH with

no path specified. That is,

```
PATH
```

is equivalent to

```
PATH.\
```

NOTE: Because the path is prepended to filenames, the terminating backslash is required.

WLINK: PATH

Parent Topic: Linker Directives and Options

Placement of Debugging Files

For remote debugging of NLM applications, there are certain files that must be on the machines:

NOVSERV.NLM, PARSERV.NLM, or SERSERV.NLM

The NLM™ application to be debugged

The data files for the NLM

The following files must be present on the remote debugging machine:

WVIDEO.EXE (if using the DOS or OS/2* OS)

NOV.TRP (if debugging over a NetWare® network)

PAR.TRP (if debugging over a parallel port)

SER.TRP (if debugging over a serial port)

NOV.DLL (if using OS/2)

The NLM source code

Parent Topic: Using the WVIDEO Debugger

PROTECT.NLM

A limited amout of memory protection is available for NetWare® 3.x servers through the use of PROTECT.NLM. PROTECT.NLM is a menu-driven debugging tool that you can use during NLM™ development.

PROTECT.NLM is used with the NetWare® 3.x OS. It provides a limited amount of memory protection to help catch instructions that access invalid

memory. PROTECT.NLM can be used by developers, but it is not used by end users.

From the PROTECT.NLM main menu, you can toggle memory protection on or off. You can also add NLM applications to the protection session and can remove NLM applications, one at a time, from the protection session.

PROTECT.NLM also features an error report screen for displaying memory faults.

You can use PROTECT.NLM as a debugging tool to detect and report programming errors that result when an NLM attempts to access memory locations to which it does not have any rights. You can also use it as a certification tool to test released or pre-released products.

PROTECT.NLM is documented in the *NLM Certification* manual that is part of the NLM Testing Tools SDK.

PSEUDOPREEMPTION

```
PSEUDOPREEMPTION
```

This directive sets a flag in the flags field of the NLM™ header indicating that this NLM will be forced to relinquish control by the NetWare® OS 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 Time**. When the time limit is exceeded, the NLM is forced to relinquish control on the next file read or write system call.

The flags are defined as shown in the Current Features table (see FLAG_ON, FLAG_OFF).

WLINK: OPTION PSEUDOPREEMPTION

Parent Topic: Linker Directives and Options

Read Fault Emulation

This parameter determines the action taken by the OS when a page fault occurs (a process attempted to read a memory location not mapped to its domain).

If set to ON, the OS temporarily maps a memory page to the current domain, and the OS reads the memory location. The page is then mapped out of the domain.

If set to OFF, an attempt to read a nonpresent memory page results in a read

page fault. If the "developer option" flag is set to OFF, the NLM™ application is quarantined. If the "developer option" is set to ON, the server will abend and call the debugger.

Default Setting: OFF

Recommended Setting: ON

Parent Topic: Setting Server Parameters

Read Fault Notification

If the "read fault emulation" flag is set to ON, the OS sends a message to the system console screen and to a log file, recording the details of the read page fault.

Default Setting: ON

Recommended Setting: ON

Parent Topic: Setting Server Parameters

REENTRANT

```
REENTRANT
```

This directive 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 in memory is reentered. In this case, one copy of the NLM is in memory.

The flags are defined as shown in the Current Features table (see FLAG_ON, FLAG_OFF).

WLINK: OPTION REENTRANT

Parent Topic: Linker Directives and Options

Registers and Flags

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

Table auto. 80386 Registers

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.

Table auto. Flags Register Bits

Flag Bit	Name
FLCF	Carry flag
FLAF	Auxiliary carry flag
FLZF	Zero flag
FLSF	Sign flag
FLIF	Interrupt flag
FLTF	Trap flag
FLPF	Parity flag
FLDF	Direction flag
FLOF	Overflow flag

Parent Topic: Specifying Expressions

Report No CLIB Context

When the Report No CLib Context option is set to Yes, if a CLIB function that requires CLIB context is called without context, the error is reported on the console screen.

Parent Topic: Changing Debug Settings

Resource Checking

When the Resource Checking option is set to Yes, NLMDebug checks for various resources allocated by your NLM™ application that are not subsequently freed. A stack walk to the location of the allocated resources not freed is displayed on the console screen.

Parent Topic: Changing Debug Settings

Ring the Bell on Error

When Ring the Bell on Error is set to Yes, a bell rings whenever an error is detected according to the conditions set in CLib Debug Options and an error message is printed to the console screen.

Parent Topic: Changing Debug Settings

Running WVIDEO

Before running the WVIDEO program on the debugging machine, the debug server must be loaded on the task machine. When WVIDEO is invoked, it will look for the debug server.

The format of the WVIDEO command is as follows:

```
WVIDEO [options][drive][path]filename[cmd_line]
```

options

These are valid WVIDEO options, each preceded by a slash (/) or a dash (-). You can specify options in any order. For a complete listing of options when your remote debugger is using DOS, see the WATCOM* manuals.

The main options used for debugging NLM™ applications are:

/TRap=*trap_file[;trap_parm]*

where *trap_file* specifies the trap file, and *trap_param* specifies the parameters for the trap file. The parameters are different for each trap file.

/TRap=*NOV;debugging_server*

where *debugging_server* is the name given when NOVSERV.NLM was loaded on the task machine.

/TRap=*SER;port.BAUD*

where *port* is the name of the serial port and *BAUD* is the transmission baud rate. The parameters *port* and *BAUD* are separated by a period

(.).

/TRap=*PAR;port*

where *port* is the name of the parallel port that connects the debugging machine and the task machine.

drive

Specifies a drive where the NLM can be found. If you do not specify the drive, the default drive is assumed.

path

Specifies the path where the NLM can be found. If you do not specify the path, the current directory is assumed.

filename

Specifies the name of the NLM file to be loaded into memory. You can include the file extension, if applicable. If you omit the extension, WVIDEO attempts to load the file using, in succession, these extensions: .NLM, .DSK, .LAN, and .NAM.

cmd_line

Specifies the command line that is passed to the application.

The following is an example of loading using WVIDEO over a Novell® network:

```
WVIDEO /trap=nov;rmt_debug my_nlm
```

Where *nov* is the name of the trap file, *rmt_debug* is the name of the debugging server created when loading NOVSERV.NLM, and *my_nlm* is the name of the NLM that is to be debugged. If the NLM needs any arguments when loading, they would be entered after *my_nlm*.

For a complete guide to using the WVIDEO debugger, see the WATCOM manuals.

Parent Topic: Using the WVIDEO Debugger

SCREENNAME

```
SCREENNAME "initial screen name"
```

This value determines the name of the first screen of an NLM™ application, which is created when the NLM is loaded. 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 is restricted to no more than 71 characters.

If this directive is not used, or if NONE is specified, there is no initial screen.

In this case, the developer must call **CreateScreen** to create a screen for the NLM.

WLINK: OPTION SCREENNAME

Parent Topic: Linker Directives and Options

Semaphore Checking

When the Semaphore Checking option is set to Yes, NLMDebug checks for any semaphores allocated by your NLM™ application that are not subsequently freed. A stack walk to the location of the allocated semaphores that are not freed is displayed on the console screen. (Semaphores allocated through CLIB that are not freed no longer abend the server.)

Parent Topic: Changing Debug Settings

Semaphore Monitoring

When the Semaphore Monitoring option is set to Yes, NLMDebug reports all thread errors on the console.

Parent Topic: Changing Debug Settings

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.

Parent Topic: NLM Debuggers

Setting Server Parameters

To aid you in testing NLM applications that contain errors that could otherwise bring down the server, you can set parameters to allow the OS to stop the offending NLM. Then you can unload the NLM and restart it while the OS keeps running. DOMAIN.NLM lets you do this.

The following parameters should be set with the **SET** command if you want the OS to quarantine modules that access memory they shouldn't.

Allow Invalid Pointers = OFF

Read fault emulation = OFF

Read fault notification = ON

Write fault emulation = OFF

Write fault notification = OFF

Developer option = OFF

These settings will be explained in detail in this section.

The NetWare 4.x OS has a number of parameters that can be set to configure the operating system various protection levels. The parameters that deal with memory protection are given below, along with default settings, recommended settings on NLM test file servers, and brief explanations of each parameter.

To change a file server parameter from its default value, you can execute the **SET** command at the file server console, or enter a **SET** command in the file server's AUTOEXEC.NCF file. Some parameters can also be set in the file server's STARTUP.NCF file. Documentation for the **SET** command and all the file server parameters is given in NetWare 4.x OS's *Utilities Reference* manual.

Related Topics

Allow Invalid Pointers

Read Fault Emulation

Read Fault Notification

Write Fault Emulation

Write Fault Notification

Developer Option

Parent Topic: DOMAIN.NLM

Setting Up and Using NLMDebug

To use NLMDebug, copy nlmdebug.nlm to the server where the NLM™ application that you want to debug is loaded. At the server prompt type `LOAD NLMDEBUG`. The NLMDebug main screen opens.

NOTE: Make sure that CLIB.NLM v.4.11 is loaded.

The Main Menu Options screen displays. Select the debug option you want by using the **Up-** and **Down-arrow** keys to highlight the option and press **Enter**.

Related Topics

Changing Debug Settings

Watching CLIB Context

Using the NCP Debugger

Viewing CLIB Remote Connection Information

Using the Function-Call Profiler

Watching for File Opens

Using the Process Timer

Using Error Watch

Setting Up the Debugger Machine for Remote Debugging

Before the debugging machine can be used for remote debugging, it first must be connected to the server.

If the debugging machine is using debugging through a parallel or serial port, there are special wiring considerations between the two computers.

See the WATCOM* manuals for more complete information.

If you will be debugging over a NetWare® network, your debugging machine must have the necessary hardware to access the network. The server and the remote workstation must also be on the same network segment. WVIDEO does not work with NOVSERV.NLM if the server and remote workstation are on different network segments.

Parent Topic: Using the WVIDEO Debugger

Setting Up the Server for Remote Debugging

Before remote debugging can begin, the task server must be set up to be a debugging server by loading a communications-server NLM™ application. The following servers (NLM applications) are provided, by WATCOM*, for use with WVIDEO under the NetWare® 3.x and 4.x OS's.

NOVSERV.NLM, for remote debugging over the NetWare network. See Loading NOVSERV.NLM.

PARSERV.NLM, for remote debugging over the parallel port. See Loading PARSERV.NLM.

SERSERV.NLM, for remote debugging over the serial port. See Loading SERSERV.NLM.

Parent Topic: Using the WVIDEO Debugger

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.

Parent Topic: Debugger Commands

SHARELIB

```
SHARELIB path to library NLM
```

This directive specifies the path to an NLM™ application to be loaded as a shared NLM. Shared NLM applications (for example, math libraries) contain global code and global data that are mapped into all memory protection domains. This method of loading exported functions can be used to avoid ring transitions when calling exported functions in other domains.

WLINK: OPTION SHARELIB

Parent Topic: Linker Directives and Options

Specifying a Linker Directive File

A linker directive 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 directive file can also include other linker directive files.

The @ directive instructs the linker to process the contents of the specified directive file. The format of the @ directive is as follows:

```
@directive_file
```

If you do not specify a file extension, the linker assumes a filename extension of .LNK.

The following is a sample linker directive file:

```
form nov 'NLM Linker Directive File'
name     test
debug    lines
file     prelude, test
import   @clib.imp
```

For information about PRELUDE.OBJ, see NLM Startup.

The IMPORT directive shown in the example above enables your NLM™ application to import (call) functions in other NLM applications. 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 directive file (as shown above with "@clib.imp").

Place all the function names in an import file (.IMP) and specify that file as the IMPORT entry in the directive file.

For example, part of the NetWare® API is CLIB.NLM. It runs in memory, and all the NLM applications loaded on the same server can import its

and all the NLM applications loaded on the same server can import its functions. To import a NetWare API function from CLIB.NLM, an NLM directive 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.

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

See the following sections for the various types of operators:

Grouping Operators

Unary Operators

Binary Operators

Ternary Operators

Registers and Flags

STACK, STACKSIZE

```
STACK stack size

STACKSIZE stack size
```

Both directives specify the stack size for the NLM™ application (in bytes, for NLMLINK). The minimum stack size is 2 KB. Over 4 KB is recommended. If no size is specified, the default is 8,192 bytes.

WLINK: OPTION STACK

Parent Topic: Linker Directives and Options

STAMPEDDATA

```
STAMPEDDATA "stamp" datafile
```

This directive causes NLMLINK to create a custom data structure in which the *dataName* is given by *stamp* and adds the contents of the *datafile* to the

NLM™ application.

For compatibility with earlier versions of NLMLINK, both *stamp* and *datafile* are required. However, *datafile* can have zero length.

Parent Topic: Linker Directives and Options

START

```
START start procedure name
```

This directive specifies the name of a symbol in the NLM™ application where execution should start. This procedure tracks the state of the NLM and helps with the final cleanup function.

If this option is not used, _Prelude is the default.

WLINK: OPTION START

Parent Topic: Linker Directives and Options

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

	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.

Parent Topic: Debugger Commands

Symbols Used by NLMLINK

The following symbols are used by NLMLINK:

@	Instructs the linker to process the contents of the specified directive file (for a definition of a directive file, see Linker Definition Syntax). **Note:**The linker should accept an empty file.
#	Marks the beginning of a comment, which continues to

	the end of the line.
Whitespace characters	Includes spaces, commas, tabs, carriage-returns, and line-feeds.

Parent Topic: Linker Directives and Options

SYNCHRONIZE

```
SYNCHRONIZE
```

This directive 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**. This prevents other console commands (particularly **LOAD**) from being processed while the NLM is loading.

The flags are defined as shown in the Current Features table (see FLAG_ON, FLAG_OFF).

WLINK: OPTION SYNCHRONIZE

Parent Topic: Linker Directives and Options

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 )
```

Parent Topic: Specifying Expressions

THREADNAME

```
THREADNAME "initial thread name"
```

This directive is used to name the threads of the NLM™ application. The first 12 characters are used to create names for threads. For example, if the name was Process, then threads created in the NLM would be named "Process 1," "Process 2," "Process 3," and so on. Thread names can be displayed by using the .P option in the NetWare® Internal Debugger.

The thread name is restricted to no more than 17 characters.

WLINK: OPTION THREADNAME

Parent Topic: Linker Directives and Options

TYPE

```
TYPE number
```

This directive is used to provide the type for the NLM™ application. If no type is given, the default value is 0. NLM types are listed in the following table.

Table auto. NLM Types and Corresponding Extensions

Number	Extension	Description
0	.NLM	Generic NLM (default value)
1	.LAN	LAN driver
2	.DSK	Disk driver
3	.NAM	Name space support module
4	.NLM	Utility or support program
5	.MSL	Mirrored Server Link
6	.NLM	OS NLM
7	.NLM	Paged high OS NLM
8	.HAM	Host Adapter Module (works with Custom Device Module)
9	.CDM	Custom Device Module (works with Host Adapter Module)
10	.NLM	OS Reserved
11	.NLM	OS Reserved
12	.NLM	OS Reserved

This directive should not be repeated.

Parent Topic: Linker Directives and Options

Unary Operators

The unary operators have precedence 1.

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

Parent Topic: Specifying Expressions

Understanding the Open File Information Screen

Information pertaining to the open file is displayed on the Open File Information screen, as follows.

Field	Description
Thread ID	Displays the thread level context for the open file.
Opening NLM	Displays the NLM that opened the file.
OS Handle	Displays the NetWare handle for the open file.
Position in File	Displays the position where the file was opened, 0 (zero) being the beginning.
Access Rights	
Connection	Displays which server connection opened the file.
Task	Displays the task number the connection was opened with.
Directory Number	Displays the name of the file opened.
Volume Number	Displays the volume on which the open file is located.
Open Count	Displays the number of times the file was opened.
Dup Count	Displays the number of times the **Dup** function was called.
Open Type	Displays the file type. File types include: LOCAL FILE

	LOCAL QUEUE REMOTE FILE DOS FILE EXTENDED ATTRIBUTE STREAM FILE BSD SOCKET CONSOLE PRINT QUEUE ASYNC IO REMOTE_EA
Task Number Allocated?	Displays Yes or No, was another task number allocated?
Opened *stdin*	Displays Yes or No, was a *stdin* opened?

Parent Topic: Watching for File Opens

Using NLMLINK

NLMLINK is a linker that was created specifically for linking NLM™ applications. Novell® supplies the linker NLMLINK in three versions: NLMLINKR (DOS real mode and 640 KB limit), NLMLINKX (DOS protected mode and requires DOS4GW), and NLMLINK2 (OS/2* version).

The current version is identified as "2.82 X2R" in the startup banner.

These three executables replace the older NLMLINKP and NLMLINKR. NLMLINKP is no longer supported. NLMLINKX is more compatible with memory managers. DOS4GW is distributed with the WATCOM* compiler (version 9.0 and above) and should be in the search path before running NLMLINKX.

All three executables are created from the same source code, so there should be no functional differences except those imposed by the execution environment.

To see the directives available for use with NLMLINK, enter the following at the command line:

```
NLMLINKP
```

The usage for NLMLINK is as follows:

```
NLMLINKP <def file name>
```

Where **<***def file name***>** is the name of the definition file holding the options for NLMLINK.

Related Topics

NLMLINK Version History

Using Pragmas

The WATCOM* C/386 and C/C++[32] compilers also include a powerful set of compiler directives known as pragmas. Pragmas allow you to specify certain compiler options, default libraries, the way structures are stored in memory, and so on. For example, auxiliary pragmas can be used to customize the code generation process as follows:

Give an alternate name to a public symbol (for example, a symbol called **MyFunc** in the C source code could generate a reference to **__MyFunc** in the object file).

Define the meaning of the cdecl, Pascal, and FORTRAN keywords.

Define the calling information for a function or for a group of functions.

Calling information includes specifying such things as:

Whether parameters are in registers or on a stack

Whether the parameters are in reverse order on the stack

The register where a particular parameter is located

The registers where the result is returned

Whether the caller or the function removes the parameters from the stack

The actual sequence of bytes to perform a function

Some versions of the compiler also allow inline assembly in pragmas.

For more information about pragmas, see the WATCOM manuals.

Parent Topic: WATCOM Compilers

Using the NetWare Internal Debugger

The NetWare® Internal Debugger is an assembly language debugger that is always present within the NetWare 3.x and 4.x OS's.

This debugger is a command line debugger that does not display source code. However, you can use the WATCOM* utility WDISASM with the "/s /l" options to create a list file containing C source interspersed throughout the assembly code. This is very helpful for debugging in 386 assembly. 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 NLM™ applications. It includes a set of supplementary commands that are customized for NLM applications, such as the **.A** (display abend or break reason) and **.P** (display all process names and addresses) commands. 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 NLM applications concurrently.

> **NOTE:** The SDK includes debug versions of IPXS.NLM, SPXS.NLM, TLI.NLM, STREAMS.NLM, and CLIB.NLM respectively named IPXSDEB.NLM, SPXSDEB.NLM, TLIDEB.NLM, STREAMSD.NLM and CLIBDEB.NLM. Debug records are linked in with each of these NLM applications, allowing better visibility to developers using the internal debugger.

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

Press **Left-shift+Right-shift+Alt+Esc** to enter the internal debugger. This keystroke combination activates the internal debugger and halts the server.

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

Enter an **INT 3** instruction in an assembly language program.

Call the NetWare API function **Breakpoint** in a C language program.

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 3.x and 4.x run 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 OS. 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 (for NLMLINK and WLINK) directive. For detailed information about this directive, see NLM Linkers.

The internal debugger is case-sensitive to symbols. Keep this in mind when specifying any symbol references.

All numbers are entered and displayed in hexadecimal format.

Bytes, words, double-words, and pointers are pushed onto the stack as 4-byte parameters.

Related Topics

Debugger Commands

Using the WATCOM Compilers

To compile the NLM™ source and create an object file, invoke the WATCOM* C/386 and C/C++[32] compilers at the workstation as follows:

```
WCC386 [options] [drive] [path] filename [options]
```

options

List of valid C compiler options, each preceded by a slash (/) or a dash (-). You can specify options in any order.

drive

Optional drive specification (for example, A: or B:). If you do not specify the drive, the default drive is assumed.

path

Optional path specification (for example, \PROGRAMS\SRC\). If you do not specify the path, the current directory is assumed.

filename

Name of the file to be compiled. You can include the filename extension, if applicable. If you omit the extension, a filename extension of .C is assumed. If you specify the period (.) but not the extension, the file is assumed to have no filename extension.

options

To display a summary of the WATCOM C/386 and C/C++[32] compiler options, enter the name of the compiler with no arguments. For an explanation of all of the compiler options, see the WATCOM manuals.

Starting with the WATCOM C/C++[32] compiler and tools, you can compile and link an NLM with the following command:

```
WCL386 /BT=NETWARE /L=NETWARE filename
```

filename

The name of the source file, including the .C filename extension.

For a complete listing of the options and syntax for WCL386 see the WATCOM manuals.

NOTE: Before using WCL386, you must define the INC386 and LIB386 environment variables to point to the libraries and the Novell headers that WATCOM provides.

NOTE: WATCOM now has a version of WVIDEO that runs on NetWare 4.x. This version is available with the latest WATCOM C/C++ compiler or from WATCOM's BBS.

Parent Topic: WATCOM Compilers

Using the WATCOM Linker

There are three ways you can use WLINK to link your NLM™ applications:

Interactively

Listing options and directives on the command line

Referencing a directive file

To use WLINK interactively, type:

```
WLINK
```

You will be prompted for input. Enter the directives and options, then press **Ctrl+Z**.

You can use WLINK by listing all directives and options at the command line. With DOS, you are limited to 127 characters, so this method has limited use. To do this, enter the following:

```
WLINK <list of directives and options>
```

Another way to use WLINK is to list all of the directives and options in a directive file and reference the file as follows:

```
WLINK @mynlm
```

You can view linker directives specific to NLM applications by entering the following command:

```
WLINK ? NOV
```

Using the WVIDEO Debugger

WVIDEO is a source-level debugger that uses remote debugging. Remote debugging involves using two computers to debug your application, as follows:

Run the NLM™ application on a NetWare® server, called the task server.

Run WVIDEO on the other PC, called the debugger machine.

Remote debugging consists of two stages:

Setting up the server for remote debugging.

Running WVIDEO on the debugger machine.

Related Topics

Setting Up the Server for Remote Debugging

Setting Up the Debugger Machine for Remote Debugging

Placement of Debugging Files

Running WVIDEO

VERBOSE

```
VERBOSE
```

This directive causes the linker to display more information while linking.

WLINK: OPTION VERBOSE

Parent Topic: Linker Directives and Options

VERSION

```
VERSION major version, minor version, revision
```

This directive communicates the version number of theNLM™ application. This number is displayed on the console screen when the NLM loads. The numbers must be separated by whitespace characters. 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.

Version information is required. The linker displays an error message if the VERSION directive is not used.

WLINK: OPTION VERSION

Parent Topic: Linker Directives and Options

Viewing CLIB Remote Connection Information

Use the CLib Remote Connection option to view in real time all remote connections to the server on which NLMDEBUG is running. The information provided by this option includes the ID of the server to which there is a connection, number of remote connections to each server, the connection number as found in the session list, and the connection type---attached, logged-in, or cached. Specific information for each connection is also available.

To view CLIB remote connection information, select CLib Remote Connections from the NLMDebug main menu and press **Enter**.

The Remote CLib Connections screen opens, displaying all remote connections to the server on which you are running NLMDEBUG. The IDs of all remote servers, the number of connections to each server, and the connection types are displayed.

Your server can have more than one type of connection to a specific remote server. A logged-in connection is one created by **LoginToFileServer**. An attached connection is one created by **AttachToFileServer**. A cached connection is one created by **ReturnConnection**. A cached connection is a connection that has been returned but not completely destroyed. Caching connections is important for speed in NetWare.

To view the connection information for a specific server, from the Remote CLib Connections screen select the server for which you want connection information and press **Enter**.

The Connection Information screen opens, displaying all remote connections to the server.

The connection numbers as found in the session list are displayed as well as the connection types.

To view detailed information for a specific connection, from the Connection Information screen select the connection for which you want detailed information and press **Enter**.

The Detailed Connection Information screen opens, displaying information for that connection.

The Slot and Status fields display the connection number and connection type, respectively. The Net field displays the network address, the node, and the socket. The NLMID field displays the ID of the NLM that gained the connection. You can also view information about the authentication state, whether or not the connection is licensed and whether or not the connection is an NDS™ connection.

Parent Topic: Setting Up and Using NLMDebug

Watching for File Opens

Use the File Opens option to have NLMDebug halt every time someone issues a CLIB open command or whenever a specified NLM™ application opens a file.

To use File Opens, select File Opens from the NLMDebug main menu and press **Enter**.

The File Opens Setup Menu is displayed.

To have NLMDebug halt whenever a CLib open command is detected, do the following:

1. **From the File Opens Setup Menu select Halt Processing on Every CLib Open and type Y.**

2. **Press F10.**

   NLMDebug runs until a CLIB open command is detected. NLMDebug halts and the Open File Information screen is displayed on the console.

To have NLMDebug halt when a specific NLM is opened, do the following:

1. **From the File Opens Setup Menu select Halt & Show Info. This NLM Opens: and enter the name of the NLM you want watched.**

2. **Press F10.**

   NLMDebug runs until the specified NLM is opened. NLMDebug halts and the Open File Information screen is displayed on the console.

For more about the Open File Information Screen, see Understanding the Open File Information Screen.

Parent Topic: Setting Up and Using NLMDebug

WATCOM Compilers

The WATCOM* C/386 and C/C++[32] compilers are cross-compilers that run under DOS or OS/2* 2.x, yet produce object files for other operating systems. They generate 32-bit protected mode code. They are command-line oriented, with 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

For detailed information about this compiler, see the WATCOM manuals.

Related Topics

Using the WATCOM Compilers

Passing Parameters

Using Pragmas

Write Fault Emulation

This parameter determines the action taken when a write page fault occurs. If set to ON, the OS temporarily maps the requested memory page to the current domain and the OS executes the write instruction. Then the OS unmaps the page from the current domain.

If set to OFF, an attempt to write to a nonexistent memory page results in a write page fault. If the "developer option" flag is set to OFF, theNLM™ application is quarantined. If the "developer option" flag is set to ON, the server abends and calls the debugger.

Default Setting: OFF

Recommended Setting: OFF

Parent Topic: Setting Server Parameters

Write Fault Notification

If the "write fault emulation" flag is set to ON, the OS sends a message to the system console screen and to a log file, recording the details of the write page fault.

Default Setting: ON

Recommended Setting: ON

Parent Topic: Setting Server Parameters

XDCDATA

```
XDCDATA path to RPC file
```

This directive specifies a path to a file containing Remote Procedure Call (RPC) descriptions for functions in theNLM™ application. RPC descriptions for functions make it possible for functions to be exported across memory protection domain boundaries.

The RPC compiler, which produces this file, has not yet been released.

WLINK: OPTION XDCDATA

Parent Topic: Linker Directives and Options

Advanced

Advanced: Guides

Advanced: Concept Guide

Advanced functionality allows you to take advantage of some advanced features provided by the NetWare® OS.

General Information

Advanced Function List

Advanced Service Areas

Dynamic Array Functions

Event Reporting and Management Functions

File I/O Functions

Using SynchronizeStart(): Example

Dynamic Linkage of Exported Symbols

Additional Links

Advanced: Functions

Advanced: Structures

Advanced: Concepts

Advanced Function List

AllocateDynArrayEntry	Allocates an entry in a dynamic array.
AllocateGivenDynArrayEntry	Allocates an entry in a dynamic array at a given element index.
AllocateResourceTag	Allocates a resource tag for a particular resource.
AsyncRead	Allows a file to be read directly from cache memory.
AsyncRelease	Releases the cache buffer memory allocated by a call to **AsyncRead**.
CancelNoSleepAESProcessEvent	Cancels a scheduled AES (Asynchronous Event Scheduler) event.
CancelSleepAESProcessEvent	Cancels a scheduled AES event.
DeallocateDynArrayEntry	Frees the dynamic array entry at a specified index.
GetFileHoleMap	Returns a block allocation map for a file.
GetSetableParameterValue	Obtains the value of a server parameter.
GetThreadDataAreaPtr	Returns the thread switch data area pointer for the current thread.
gwrite	Writes multiple buffers to a file.
ImportSymbol	Returns a pointer to an exported symbol.
NWAddSearchPathAtEnd	Adds a search path to the end of the search path list that the OS uses to determine the location of NLM applications.
NWGarbageCollect	Unfragments freed server memory.
NWDeleteSearchPath	Deletes a search path from the search path list that the OS uses to determine the location of NLM applications.
NWGetSearchPathElement	Returns a search paath from the search path list that the OS uses to determine the location of NLM applications.

NWInsertSearchPath	Inserts a search path into the search path list that the OS uses to determine the location of NLM applications.
qread	Performs a low-overhead read operation.
qwrite	Performs a low-overhead write operation.
RegisterConsoleCommand	Registers a console command parsing function.
RegisterForEvent	Registers to be notified when a particular event occurs.
SaveThreadDataAreaPtr	Sets the thread switch data area pointer for the current thread.
ScanSetableParameters	Returns information about server parameters.
ScheduleNoSleepAESProcessEvent	Defines a procedure to be called by the asynchronous event scheduler (AES) after a specified delay.
ScheduleSleepAESProcessEvent	Defines a procedure to be called by the AES after a specified delay.
SetSetableParameterValue	Sets the value of a server parameter.
SynchronizeStart	Restarts the NLM™ start-up process when using synchronization mode.
UnimportSymbol	Eliminates the dependency of an NLM on a specified external symbol.
UnRegisterConsoleCommand	Cancels a registered console command parsing function.
UnregisterForEvent	Cancels a registration for event notification.

Dynamic Array Functions

The dynamic array functions, listed below, assist the developer in handling arrays that grow dynamically.

AllocateDynArrayEntry

AllocateGivenDynArrayEntry

DeallocateDynArrayEntry

These functions perform some of the housework necessary when expanding in-memory tables.

An example of this kind of table is a connection table that might be used in an NLM™ application. More specifically, a database server might maintain a 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.

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. The DAB is 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.

There are two methods for generating indexes:

One method is to call **AllocateDynArrayEntry**, which generates the index for a new entry.

The other method is to specify which index to use when allocating a new entry in a dynamic array by calling **AllocateGivenDynArrayEntry**.

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** 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**.

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**.

Dynamic Linkage of Exported Symbols

ImportSymbol and **UnimportSymbol** 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.

Beginning with version 4.0 of the SDK, a header file (NEWIN400.H for

NetWare 4.0) is provided that typedefs all functions that are new to that version of CLIB.NLM as compared to the previous version. This can be of some aid in creating variables to which to assign the return value of **ImportSymbol**.

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

RegisterConsoleCommand

SaveThreadDataAreaPtr

SynchronizeStart

UnRegisterConsoleCommand

NLM applications that manage events can use event management functions to allocate resource tags and process events. Event management functions are listed below:

AllocateResourceTag

CancelNoSleepAESProcessEvent

CancelSleepAESProcessEvent

ScheduleNoSleepAESProcessEvent

ScheduleSleepAESProcessEvent

File I/O Functions

NLM applications that need faster access to files and information about sparse files can use File I/O functions.

AsyncRead

AsyncRelease

GetFileHoleMap

gwrite

qread

qwrite

Advanced: Functions

AllocateDynArrayEntry

Allocates an entry in a dynamic array
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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	
5	ENOMEM	Not enough memory.

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, DeallocateDynArrayEntry

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
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwdynarr.h>

int AllocateGivenDynArrayEntry (
   T_DYNARRAY_BLOCK   *dabP,
   int                ndx);
```

Parameters

dabP

(IN) Specifies 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:

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** for more detailed information about this macro.

See Also

AllocateDynArrayEntry, **DeallocateDynArrayEntry**

AllocateResourceTag

Allocates a resource tag for a particular resource
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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, GetNLMHandle

AsyncRead

Reads a file directly from cache memory
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwfile.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) Receives the number of bytes actually read from the file.

localSemaphoreHandle

(IN) 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) Returns a pointer to an array of structures which contain the cache buffer pointers, lengths, and completion codes.

numberOfCacheBuffers

(OUT) Receives 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, **ExamineLocalSemaphore**, **OpenLocalSemaphore**, **WaitOnLocalSemaphore**

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

SMP Aware: No

Service: Advanced

Syntax

```
#include <nwfile.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**.

Return Values

None

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, **atexit**, **AtUnload**, **signal**

CancelNoSleepAESProcessEvent

Cancels a previously scheduled event
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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.

Return Values

None

Remarks

The *EventNode* should have been used in a previous call to **ScheduleNoSleepAESProcessEvent**.

See Also

CancelSleepAESProcessEvent, **ScheduleNoSleepAESProcessEvent**

CancelSleepAESProcessEvent

Cancels a previously scheduled event
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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.

Return Values

None

Remarks

The *EventNode* should have been used in a previous call to **ScheduleSleepAESProcessEvent**.

See Also

CancelNoSleepAESProcessEvent

DeallocateDynArrayEntry

Frees the dynamic array entry at the specified index
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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:

-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, **AllocateGivenDynArrayEntry**

GetFileHoleMap

Returns a block allocation map for a file
Local Servers: blocking
Remote Servers: blocking
Classification: 3.x, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwfile.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 necessary information to compute the number of blocks.

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) Receives 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
SMP Aware: No
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

This function returns 0 if successful, or -1 if an invalid setable parameter string was specified.

Remarks

This function obtains the value of a setable parameter. A setable parameter is a NetWare OS parameter that can be set using the **SET** console command.

The *setableParameterString* parameter is the name of the setable parameter, such as "Cache Buffer Size".

Enough space should be set aside for the return value to be copied to the destination address pointed to by *returnValue*. The maximum size of a

server console parameter is 512 bytes.

See Also

ScanSetableParameters, **SetSetableParameterValue**, "SET" in *Supervising the Network*

GetThreadDataAreaPtr

Gets the thread switch Data Area Pointer for the current thread
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: Yes
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, **RegisterForEvent**

gwrite

Writes multiple buffers to a file with a single call
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwfile.h>

int gwrite (
   int                  handle,
   T_mwriteBufferStructure
                        *bufferP,
   LONG                 numberOfBuffers,
   LONG                 *numberOfBuffersWritten);
```

Parameters

handle

(IN) Handle of the file to which data is to be written.

bufferP

(IN) Pointer 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) Receives the number of buffers actually written.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

Remarks

The *bufferP* structure is defined in nwadv.h as:

```
char *mwriteBufferPointer
LONG mwriteBufferLength
```

```
int  reserved
```

See Also

qwrite

ImportSymbol

Returns a pointer to an exported symbol
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
SMP Aware: Yes
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) Specifies 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 Importing a Function: Example.

See Also

GetNLMHandle, **UnimportSymbol**

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

SMP Aware: No

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

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

NWGarbageCollect

Unfragments freed server memory
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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.

Return Values

None

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 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 from a call to **GetNLMHandle**.

Blocking Information: Although **NWGarbageCollect** can block in some instances, it does not always do so.

See Also

FindNLMHandle, **GetNLMHandle**, **MapNLMIDToHandle**

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

SMP Aware: No

Service: Advanced

Syntax

```
#include <nwadv.h>

int NWDeleteSearchPath (
   LONG   searchPathNumber);
```

Parameters

searchPathNumber

(IN) Specifies the search path number to be deleted.

Return Values

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

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

SMP Aware: No

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

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

SMP Aware: No

Service: Advanced

Syntax

```
#include <nwadv.h>

int NWInsertSearchPath (
   LONG   searchPathNumber,
   BYTE  *searchPath);
```

Parameters

searchPathNumber

(IN) Specifies the search path number to be entered.

searchPath

(OUT) Points to a new search path to be added to the search path list.

Return Values

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
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwfile.h>

int qread (
   int    handle,
   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

This function returns the number of bytes read.

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

qwrite

Performs a low-overhead write operation
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwfile.h>

int qwrite (
   int    handle,
   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* can be set to:

4	EBADF	Bad file number.

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

RegisterConsoleCommand

Registers a console command parsing function
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwadv.h>

void RegisterConsoleCommand (
   struct commandParserStructure   *newCommandParser);
```

Parameters

newCommandParser

(IN) Defines a command parsing function.

Return Values

None (See "Remarks" for values returned by function inside *newCommandParser*.)

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 */
   };
```

parseRoutine can have the following possible values returned:

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 ???"

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 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 theContext Problems with OS Threads.

See Also

AllocateResourceTag, **UnRegisterConsoleCommand**

RegisterForEvent

Registers to be notified when a particular event occurs
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwadv.h>
LONG RegisterForEvent (
   LONG   eventType,
   void   (*reportProcedure) (
      LONG   parameter),
   LONG   (*warnProcedure) (
      void   (*OutputRoutine) (
      void   *controlString, ...),
   LONG   parameter) );
```

Parameters

eventType

(IN) Specifies an event type. The event types are listed in nwadv.h; for example, EVENT_VOL_SYS_MOUNT.

reportProcedure

(IN) Lists the events that occurred.

warnProcedure

(IN) Can be NULL, since most events do not support a warning procedure. (See nwadv.h for events which do call the warning procedure.) The warning procedure returns zero or nonzero. If nonzero, the user is given a choice to unload or not.

OutputRoutine

(IN) System-supplied.

parameter

(IN) Depends on event being registered.

Return Values

This function returns a nonzero event handle if successful. Otherwise it returns EFAILURE (-1).

Remarks

Use this function whenever it is necessary to know about any of the events listed in nwadv.h.

NOTE: To register for NDS events, call **NWDSERegisterForEvent**.

The function registered by **RegisterForEvent** 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 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 is 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.

NOTE: It is very important that report routines for events do not block unless it is explicitly allowed. The warn/check routines can sleep (but should not have to). If a report procedure sleeps when prohibited, the server abends. Report routines for the following events **can** sleep:

EVENT_ACTIVATE_SCREEN
EVENT_ALLOCATE_CONNECTION
EVENT_ANY_VOL_MOUNT
EVENT_ANY_VOL_DISMOUNT
EVENT_CLEAR_CONNECTION
EVENT_CLOSE_SCREEN
EVENT_DOWN_SERVER
EVENT_EXIT_TO_DOS
EVENT_LOGIN_USER
EVENT_LOGOUT_CONNECTION
EVENT_MODULE_LOAD
EVENT_MODULE_UNLOAD
EVENT_MODULE_UNLOADED
EVENT_NETWARE_ALERT
EVENT_OPEN_SCREEN
EVENT_PROTOCOL_BIND
EVENT_PROTOCOL_UNBIND
EVENT_REMOVE_PUBLIC
EVENT_SFT3_IMAGE_STATE
EVENT_SFT3_PRESYNC_STATE
EVENT_SFT3_SERVER_STATE
EVENT_VOL_SYS_DISMOUNT
EVENT_VOL_SYS_MOUNT

The report routine for all other events must **not** sleep.

The following summarizes the events, whether they call a warn routine, and whether or not they can sleep:

Event		Calls warn routine	Can Sleep	Description
EVENT_VOL_SYS_MOUNT	0	No	Yes	*parameter* is undefined. Report Routine is called immediately after vol SYS: has been mounted.
EVENT_VOL_SYS_DISMOUNT	1	Yes	Yes	*parameter* is undefined. Warn Routine and Report Routine is called

				Routine is called before vol SYS: is dismounted.
EVENT_ANY_VOL_MOUNT	2	No	Yes	*parameter* is volume number. Report Routine is called immediately after any volume is mounted.
EVENT_ANY_VOL_DISMOUNT	3	Yes	Yes	*parameter* is volume number. Warn Routine and Report Routine is called before any volume is dismounted.
EVENT_DOWN_SERVER	4	Yes	Yes	*parameter* is undefined. Warn Routine and Report Routine is called before the server is shut down. See Using DOWN_SERVER Event: Example.
EVENT_EXIT_TO_DOS	7	No	Yes	*parameter* is undefined. The Report Routine is called before the server exits to DOS. Only available in NetWare 4.x
EVENT_MODULE_UNLOAD	8	Yes	Yes	*parameter* is module handle. Warn Routine and Report Routine is called when a module is unloaded from the console command line. Only the Report Routine is called when a module unloads itself. See Using the MODULE_UNLOAD Event: Example.
EVENT_CLEAR_CONNECTION	9	No	Yes	*parameter* is connection number. Report Routine is called before the connection is cleared.
EVENT_LOGIN_USER	1	No	Yes	*parameter* is

	0			connection number. Report Routine is called after the connection has been allocated.
EVENT_CREATE_BINDERY_OBJ	11	No	No	*parameter* is object ID. Report Routine is called after the object is created and entered in the bindery.
EVENT_DELETE_BINDERY_OBJ	12	No	No	*parameter* is object ID. Report Routine is called before the object is removed from the bindery.
EVENT_CHANGE_SECURITY	13	No	No	*parameter* is a pointer a structure of type EventSecurityChange Struct. Report Routine is called after a security equivalence change has occurred.
EVENT_ACTIVATE_SCREEN	14	No	Yes	*parameter* is screen ID. Report routine is called after the screen becomes the active screen. (Not supported in NetWare 3.x)
EVENT_UPDATE_SCREEN	15	No	No	*parameter* is screen ID. Report routine is called after a change is made to the screen image. (Not supported in NetWare 3.x)
EVENT_UPDATE_CURSOR	16	No	No	*parameter* is screen ID. Report routine is called after a change to the cursor position or state occurs. (Not supported in NetWare 3.x)
EVENT_KEY_WAS_PRESSED	17	No	No	*parameter* is undefined. Report routine is called whenever a key on the keyboard is pressed (including

				shift/alt/control). This routine is called at interrupt time.
EVENT_DEACTIVATE_SCREEN	18	No	No	*parameter* is screen ID. Report routine is called when the screen becomes inactive. (Not supported in NetWare 3.x)
EVENT_TRUSTEE_CHANGE	19	No	No	*parameter* is a pointer to type struct EventTrusteeChangeStruct. The report routine is called everytime there is a change to a trustee in the file system. See Using the TRUSTEE_CHANGE Event: Example.
EVENT_OPEN_SCREEN	20	No	Yes	*parameter* is the screen ID for the newly created screen. The report routine is called after the screen is created. (Not supported in NetWare 3.x)
EVENT_CLOSE_SCREEN	21	No	Yes	*parameter* is the screen ID for the screen that is to be closed. The report routine is called before the screen is closed. (Not supported in NetWare 3.x)
EVENT_MODIFY_DIR_ENTRY	22	No	No	*parameter* is a pointer to a structure of type EventModifyDirEntryStruct which contains the modify information. The report routine is called right after the entry is changed but before the directory entry is unlocked.
EVENT_NO_RELINQUIS	2	No	No	*parameter* is the

H_CONTROL	3			running process. This is called when the timer detects that a process is hogging the processor. The report routine must not sleep. (Not supported in NetWare 3.x)
EVENT_THREAD_SWITCH	25	No	No	*parameter* is the thread ID of the thread that was executing when the thread switch occurred. The report routine is called when the new thread begins executing. The report routine must not go to sleep. This event applies **only** to threads in the calling NLM.
EVENT_MODULE_LOAD	27	No	Yes	*parameter* is module handle. The report routine is called after a module has loaded. (Not supported in NetWare 3.x)
EVENT_CREATE_PROCESS	28	No	No	*parameter* is the PID of the process being created. It is called after the process is created. The report routine must not sleep. (Not supported in NetWare 3.x)
EVENT_DESTROY_PROCESS	29	No	No	*parameter* is the PID of the process being destroyed. It is called before the process is actually destroyed. The report routine must not sleep. (Not supported in NetWare 3.x)
EVENT_NEW_PUBLIC	32	No	No	*parameter* is a pointer to a length preceded string which is the name of the new public entry point. (Not supported in

				NetWare 3.x)
EVENT_PROTOCOL_BIND	33	No	Yes	*parameter* is a pointer to a structure of type EventProtocolBindStruct. This event is generated every time a board is bound to a protocol. (Not supported in NetWare 3.x)
EVENT_PROTOCOL_UNBIND	34	No	Yes	*parameter* is a pointer to a structure of type EventProtocolBindStruct. This event is generated every time a board is unbound from a protocol. (Not supported in NetWare 3.x)
EVENT_ALLOCATE_CONNECTION	37	No	Yes	*parameter* is connection number. Report Routine is called after the connection is allocated. (Not supported in NetWare 3.x)
EVENT_LOGOUT_CONNECTION	38	No	Yes	*parameter* is connection number. Report Routine is called before the connection is logged out. (Not supported in NetWare 3.x)
EVENT_MLID_REGISTER	39	No	No	*parameter* is board number. Report Routine is called after the MLID™ software is registered. (Not supported in NetWare 3.x)
EVENT_MLID_DEREGISTER	40	No	No	*parameter* is board number. Report Routine is called before the MLID is deregistered. (Not supported in NetWare 3.x)
EVENT_DATA_MIGRATI	4	No	No	*parameter* is a pointer

ON	1			to a structure of type EventDateMigrationInfo. This event is generated when a file's data has been migrated. (Not supported in NetWare 3.x)
EVENT_DATA_DEMIGRATION	42	No	No	*parameter* is a pointer to a structure of type EventDateMigrationInfo. This event is generated when a file's data has been de-migrated. (Not supported in NetWare 3.x)
EVENT_QUEUE_ACTION	43	No	No	*parameter* is a pointer to a structure of type EventQueueNote. This event is generated when a queue is activated, deactivated, created, or deleted. (Not supported in NetWare 3.x)
EVENT_NETWARE_ALERT	44	No	Yes	*parameter* is a pointer to a strcutre of type EventNetwareAlertStruct. This event is generated any time the following alert calls are made: SystemAlert (NW 3.0) QueueSystemAlert (NW 3.0) INWSystemAlert (NW 3.11 (temp)) INWQueueSystemAlert (NW 3.11 (temp))

				NetWareAlert (NW 4.x)
EVENT_CLOSE_FILE	50	No	No	*parameter* is a pointer to a structure of type EventCloseFileInfo. (Not supported in NetWare 3.x)
EVENT_CHANGE_TIME	51	No	No	This event is given when the time is changed or when Time Synchronization schedules a nonuniform adjustment. The *parameter* is the UTC time (in seconds) before the time change. The current time is available from the OS. Since you have no way of knowing the magnitude of the time change, nor whether it has taken place or is scheduled for the next clock interrupt, you must detect the time change on your own. In general, if current time is less than old time, or at least two seconds ahead of the old time, then the time change has been applied. You must wait for one of those conditions to be sure that the time change has "settled down" before you can assume that the event has "happened." (Not supported in NetWare 3.x)
EVENT_MODULE_UNLOADED	56	No	Yes	*parameter* is a module handle. Report Routine is called after the exit routine of the NLM has been called, after the resources of

				the NLM have been returned to the OS, and after the resources of the NLM have are unlinked from the OS's lists. The only part of the NLM left is the NLM memory for the load definition structure, data image, and code image. (Not supported in NetWare 3.x)
EVENT_REMOVE_PUBLIC	57	No	Yes	*parameter* is the address of the public entry point. This occurs only on module unload. (Not supported in NetWare 3.x)
EVENT_SFT3_SERVER_STATE	60	No	No	*parameter* is the ServerState Number. This event is available only in the IOEngine. When it occurs, the Report Routine is called with one of the following values: 0 = IOEngineState 1 = PrimaryNoSecondaryState 2 = PrimarySyncingWithSecondaryState 3 = PrimaryTransferingMemoryImageState 4 = PrimaryWithSecondaryState 5 = SecondaryTransferingMemoryImageState 6 = SecondaryMirroredState (Not supported in

				NetWare 3.x)
EVENT_SFT3_IMAGE_STATE	61	No	No	*parameter* is the memory mirror state (0 = not mirrored, 1 = mirrored). This event is available only in the MSEngine. (Not supported in NetWare 3.x)
EVENT_SFT3_PRESYNC_STATE	62	No	Yes	*parameter* is currently unused. This event is called when the primary is ready to synchronize with secondaries. (Not supported in NetWare 3.x)

NOTE: Event 24, EVENT_SYS_ALERT, and its associated structure, EventSystemAlertStruct, have been removed from nwadv.h. Instead, event 44, EVENT_NETWARE_ALERT, should be used with its structure, EventNetwareAlertStruct.

See Also

UnregisterForEvent, **NWDSERegisterForEvent**

SaveThreadDataAreaPtr

Sets the thread switch Data Area Pointer for the current thread
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Advanced

Syntax

```
#include <nwadv.h>

void SaveThreadDataAreaPtr (
   void    *threadDataAreaPtr);
```

Parameters

threadDataAreaPtr
(IN) Specifies 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, **RegisterForEvent**

ScanSetableParameters

Returns information about NetWare server console parameters
Local Servers: blocking
Remote Servers: N/A
Classification: 4.x
SMP Aware: No
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) 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) Specifies or receives the name of a setable parameter (an ASCIIZ string). (Input if *scanCategory* is -2 or -5.)

rType

(OUT) Receives the type of the setable parameter.

rFlags

(OUT) Receives the setable parameter flags.

rCategory

(OUT) Receives the setable parameter category.

rParameterDescription

(OUT) Receives the description of a setable parameter (an ASCIIZ string).

rCurrentValue

(OUT) Receives 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) Receives the lower limit of the setable parameter.

rUpperLimit

(IN/OUT) Receives 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:

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 /* Could be 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
4	offset
5	string
6	trigger

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:

0x000	startup only

1	
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, **SetSetableParameterValue**, "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

SMP Aware: No

Service: Advanced

Syntax

```
#include <nwthread.h>

extern void ScheduleNoSleepAESProcessEvent(
   struct AESProcessStructure   *EventNode);
```

Parameters

EventNode

(IN) Points to a structure that defines the event.

Return Values

None

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 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, seeContext Problems with

OS Threads.

See Also

AllocateResourceTag, **CancelNoSleepAESProcessEvent**, **ScheduleSleepAESProcessEvent**

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

SMP Aware: No

Service: Advanced

Syntax

```
#include <nwthread.h>

extern void ScheduleSleepAESProcessEvent(
   struct AESProcessStructure   *EventNode);
```

Parameters

EventNode

(IN) Points to the AESProcessStructure, which defines the event.

Return Values

None

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 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.

See Also

AllocateResourceTag, **CancelSleepAESProcessEvent**, **ScheduleNoSleepAESProcessEvent**

SetSetableParameterValue

Changes the value of a NetWare server console parameter
Local Servers: blocking
Remote Servers: N/A
Classification: 4.x
SMP Aware: No
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

This function returns 0 if successful, or -1 if an invalid setable parameter string was specified.

Remarks

This function sets the value of a setable parameter. A setable parameter is a NetWare OS parameter that can be set using the SET console command.

The *setableParameterString* is the name of the setable parameter, such as "Cache Buffer Size".

See Also

GetSetableParameterValue, **ScanSetableParameters**, "SET" in

SynchronizeStart

Restarts the NLM startup process when using synchronization mode
Local Servers: blocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwadv.h>

void SynchronizeStart (void);
```

Return Values

None

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 Using SynchronizeStart(): Example

UnimportSymbol

Eliminates dependency of an NLM on the specified external symbol
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
SMP Aware: Yes
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

UnRegisterConsoleCommand

Unregisters a console command parsing function
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Advanced

Syntax

```
#include <nwadv.h>

LONG UnRegisterConsoleCommand (
   struct commandParserStructure   *commandParser);
```

Parameters

commandParser

(IN) Specifies 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

UnregisterForEvent

Cancels a previous registration for event notification
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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

Advanced: Structures

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
Set by AES.

AWakeUpDelayAmount
Contains the number of ticks to wait (developer-defined).

AWakeUpTime
Set by AES.

AProcessToCall
Points to the function to call (developer-defined).

ARTag
Contains the resource tag (developer-defined).

AOldLink
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

Set by **RegisterConsoleCommand**.

parseRoutine

Points to a developer-defined parsing routine

RTag

Contains 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

Handle of the file that was closed.

station

Connection number that closed the file.

task

Task number of the connection that closed the file.

fileHandleFlags

Attributes of the file handle. See *fileHandleFlags* in nwadv.h for a list of the flags.

completionCode

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

Directory number of the entry in the DOS name space.

OwnerDirEntry

Directory number of the entry in an other than DOS name space (if applicable).

OwnerNameSpace

Name space number of this entry.

OwnerFileName

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

Directory number of the entry being modified.

nameSpace

Name space in which the modification is occurring.

modifyBits

Specifies the fields of the directory entry that are being changed (see nwdir.h).

modifyVector

Pointer 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.

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

Bindery object ID of the trustee being changed.

entryID

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. (the flags are EVENT_NEW_TRUSTEE and EVENT_REMOVE_TRUSTEE.)

newRights

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

Contains the address of the first character for the cache buffer.

cacheBufferLength

Contains the number of bytes to be used from the cache buffer.

completionCode

Contains 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

DABelementSize

DABrealloc

Points to a memory allocation function. This function is normally **realloc**, but you can define your own function.

DABgrowAmount

Contains the number of elements by which to increase the dynamic array when more elements are needed.

DABnumEntries

Contains the number of entries in the dynamic array.

T_mwriteBufferStructure

Contains information about a buffer to be used by **gwrite**
Service: Advanced
Defined In: nwadv.h

Structure

```
typedef struct mwriteBufferStructure
{
   char   *mwriteBufferPointer;
   LONG   mwriteBufferLength;
   int    reserved;
} T_mwriteBufferStructure;
```

Fields

mwriteBufferPointer

Points to a buffer.

mwriteBufferLength

Contains the size of the buffer.

Bit Array

Bit Array: Functions

BitClear

Clears the specified bit
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: No
Service: Bit Array

Syntax

```
#include <nwbitops.h>

void BitClear (
   void   *bitArray,
   LONG   bitNumber);
```

Parameters

bitArray

(IN) Points to the bit array.

bitNumber

(IN) Specifies an index into the bit array.

Return Values

None

Remarks

The *bitArray* parameter specifies the target array. The bit number can be greater than 32, targeting a bit well into the target array.

See Also

BitSet

Example

BitClear

```
#include <nwbitops.h>
void   *bitArray;
LONG   bitNumber;
BitClear (bitArray, bitNumber);
```

```
BitClear (bitArray, bitNumber);
```

BitSet

Sets the target bit
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: No
Service: Bit Array

Syntax

```
#include <nwbitops.h>

void BitSet (
   void   *bitArray,
   LONG   bitNumber);
```

Parameters

bitArray

(IN) Points to the bit array.

bitNumber

(IN) Specifies an index into the bit array.

Return Values

None

Remarks

The *bitArray* parameter specifies the target array. The bit number can be greater than 32, targeting a bit well into the target array.

See Also

BitClear, **BitTest**

Example

BitSet

```
#include <nwbitops.h>
void   *bitArray;
LONG   bitNumber;
BitSet (bitArray, bitNumber);
```

```
BitSet (bitArray, bitNumber);
```

BitTest

Determines whether the specified bit is set
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: No
Service: Bit Array

Syntax

```
#include <nwbitops.h>

LONG BitTest (
   void   *bitArray,
   LONG   bitNumber);
```

Parameters

bitArray

(IN) Points to the bit array.

bitNumber

(IN) Specifies an index into the bit array.

Return Values

This function returns a bit value of 0 if the specified bit is cleared. Otherwise, it returns a value of 1.

Remarks

The *bitArray* parameter specifies the target array. The bit number can be greater than 32, targeting a bit well into the target array.

See Also

BitClear, **BitSet**

Example

BitTest

```
#include <nwbitops.h>
LONG   bitValue;
void   *bitArray;
```

```
void   *bitArray;
LONG   bitNumber;
bitValue = BitTest (bitArray, bitNumber);
```

BitTestAndClear

Returns the current value of the specified bit and then clears the bit (if the bit was not already cleared)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 2.x, 3.x, 4.x

SMP Aware: No

Service: Bit Array

Syntax

```
#include <nwbitops.h>

LONG BitTestAndClear (
   void   *bitArray,
   LONG   bitNumber);
```

Parameters

bitArray

(IN) Points to the bit array.

bitNumber

(IN) Specifies an index into the bit array.

Return Values

This function returns an old bit value of 0 if the specified bit is cleared. Otherwise, it returns a value of 1.

Remarks

The *bitArray* parameter specifies the target array. It can be byte-aligned and can point to an array of up to 2^{32} - 1 bits.

See Also

BitTestAndSet

Example

BitTestAndClear

```
#include <nwbitops.h>
LONG   oldBitValue;
```

```
LONG   oldBitValue;
void   *bitArray;
LONG   bitNumber;
oldBitValue = BitTestAndClear (bitArray, bitNumber);
```

BitTestAndSet

Returns the current value of the specified bit and then sets the bit (if the bit was not already set)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 2.x, 3.x, 4.x

SMP Aware: No

Service: Bit Array

Syntax

```
#include <nwbitops.h>

LONG BitTestAndSet (
   void   *bitArray,
   LONG   bitNumber);
```

Parameters

bitArray

(IN) Points to the bit array.

bitNumber

(IN) Specifies an index into the bit array.

Return Values

This function returns an old bit value of 0 if the specified bit is cleared. Otherwise, it returns a value of 1.

Remarks

The *bitArray* parameter specifies the target array. It can be byte-aligned and can point to an array of up to 2^{32} - 1 bits.

See Also

BitTestAndClear

Example

BitTestAndSet

```
#include <nwbitops.h>
LONG   oldBitValue;
```

```
LONG   oldBitValue;
void   *bitArray;
LONG   bitNumber;
oldBitValue = BitTestAndSet (bitArray, bitNumber);
```

ScanBits

Scans a bit array to find the first bit set
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: No
Service: Bit Array

Syntax

```
#include <nwbitops.h>

LONG ScanBits (
   void   *bitArray,
   LONG   startingBitNumber,
   LONG   totalBitCount);
```

Parameters

bitArray

(IN) Points to the bit array.

startingBitNumber

(IN) Specifies the index number of the bit to start searching on.

totalBitCount

(IN) Specifies the size of the bit array.

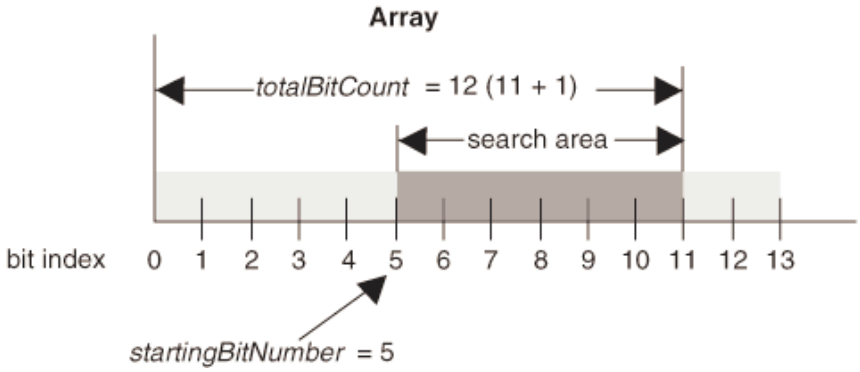
Return Values

This function returns a bit index (relative to the beginning of the array) to the first bit set, or a value of -1 if no bit is set.

Remarks

The *bitArray* parameter specifies the target array, which can begin on a byte boundary. A bit array can be as large as 2^{32} - 1 bits in length.

The *totalBitCount* parameter specified the total number of bits from the beginning of the array to the end of the search area. Therefore, if the search area is from bit index 5 to bit index 11, the *startingBitNumber* would be 5 and the *totalBitCount* would be 12 (the bit index +1).



See Also

ScanClearedBits

Example

ScanBits

```
#include <nwbitops.h>
LONG   bitNumber;
void   *bitArray;
LONG   startingBitNumber;
LONG   totalBitCount;
bitNumber = ScanBits (bitArray, startingBitNumber, totalBitCount);
```

ScanClearedBits

Scans a bit array to find the first bit that has been cleared
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Bit Array

Syntax

```
#include <nwbitops.h>

LONG ScanClearedBits (
   void   *bitArray,
   LONG   startingBitNumber,
   LONG   totalBitCount);
```

Parameters

bitArray

(IN) Points to the bit array.

startingBitNumber

(IN) Specifies the index number of the bit to start searching on.

totalBitCount

(IN) Specifies the size of the bit array.

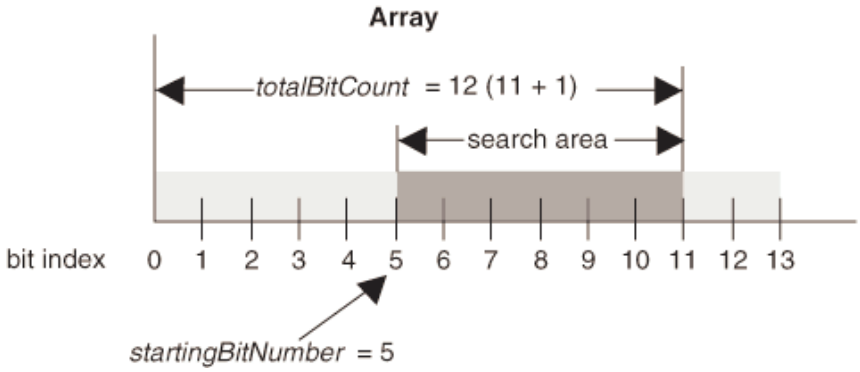
Return Values

This function returns a bit index (relative to the beginning of the array) to the first bit cleared, or a value of -1 if no bit is cleared.

Remarks

The *bitArray* parameter specifies the target array, which can begin on a byte boundary. A bit array can be as large as 2^{32} - 1 bits in length.

The *totalBitCount* parameter specified the total number of bits from the beginning of the array to the end of the search area. Therefore, if the search area is from bit index 5 to bit index 11, the *startingBitNumber* would be 5 and the *totalBitCount* would be 12 (the bit index +1).



See Also

ScanBits

Example

ScanClearedBits

```
#include <nwbitops.h>
LONG   bitNumber;
void   *bitArray;
LONG   startingBitNumber;
LONG   totalBitCount;
bitNumber = ScanClearedBits (bitArray, startingBitNumber, totalBitCount
```

Character Manipulation

Character Manipulation: Functions

isalnum

Tests for an alphanumeric character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isalnum (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

This function or macro returns a value of 0 if the argument is neither an alphabetic character nor a digit. Otherwise, a nonzero value is returned.

Remarks

The **isalnum** function or macro tests if the argument *c* is an alphanumeric character (a to z, A to Z, or 0 to 9). An alphanumeric character is any character for which **isalpha** or **isdigit** is true.

See Also

isalpha, **isdigit**, **islower**

Example

isalnum

```
#include <ctype.h>
#include <stdio.h>
main ()
{
   printf ("%d %d \w", isalnum ('Q'), isalnum ('!'));
```

```
}
```

produces the following:

```
1 0
```

isalpha

Tests for an alphabetic character (a to z or A to Z) (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isalpha (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

isalpha returns a value of 0 if the argument is not an alphabetic character. Otherwise, a nonzero value is returned.

Remarks

The **isalpha** function or macro tests for an alphabetic character (a to z or A to Z). An alphabetic character is any character for which **isupper** or **islower** is true.

See Also

isalnum, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

Example

isalpha

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

main ()
```

```
{
   printf ("%d %d %d", isalpha ('2'), isalpha ('Q'), isalpha ('!'));
}
```

produces the following:

```
0 1 0
```

isascii

Tests for an ASCII character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isascii (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

The **isascii** function or macro returns a nonzero value when the character is in the range from 0 to 127. Otherwise, a value of 0 is returned.

Remarks

The **isascii** function or macro tests for a character in the range from 0 to 127.

See Also

isalpha, **isalnum**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

Example

isascii

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

main ()
{
```

```
   printf ("%d %d %d \u", isascii ('\u'), isascii ('\f'), isascii ('A')
}
```

produces the following:

```
1 1 1
```

iscntrl

Tests for a control character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int iscntrl (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

The **iscntrl** function or macro returns a nonzero value when the argument is a control character. Otherwise, a value of 0 is returned.

Remarks

A control character is any character whose value is from 0 to 31.

See Also

isalnum, **isalpha**, **isdigit**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

Example

iscntrl

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

main ()
{
   printf ("%d %d %d \u", iscntrl ('\u'), iscntrl ('\A'), iscntrl ('f')
```

```
}
```

produces the following:

```
1 0 1
```

isdigit

Tests for a decimal-digit character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isdigit (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

isdigit returns a nonzero value when the argument is a digit. Otherwise, a value of 0 is returned.

Remarks

The **isdigit** function or macro tests for any decimal-digit character (0 to 9).

See Also

isalnum, **isalpha**, **iscntrl**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

Example

isdigit

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

main ()
{
   char   *s, *p = "QRSTU1234ABC";
```

```
   for (s = p; *s; s++)
      printf ("%d", isdigit (*5));
}
```

produces the following:

```
0 0 0 0 0 1 1 1 1 0 0 0
```

isgraph

Tests for any printable character (except a space character) (function or macro)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isgraph (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

This function or macro returns a nonzero value when the argument is a printable character (except a space character). Otherwise, a value of 0 is returned.

Remarks

The **isgraph** function or macro tests for any printable character (except a space character). The **isprint** function is similar, except that the space character is also included in the character set being tested.

See Also

isalnum, **isalpha**, **iscntrl**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

islower

Tests for a lowercase character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int islower (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

islower returns a nonzero value when the argument is a lowercase character. Otherwise, a value of 0 is returned.

Remarks

The **islower** function or macro tests for any lowercase character (a to z) in the ASCII code set.

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

isprint

Tests for a printable character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isprint (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

isprint returns a nonzero value when the argument is a printable character. Otherwise, a value of 0 is returned.

Remarks

The **isprint** function or macro tests for any printable character, including a space character.

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **islower**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

Example

isprint

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

main()
```

```
{
   int i;
   while((i = getch()) != '0')
      printf("%s\r\n",isprint(i) ? "yes" : "no");
}
```

ispunct

Tests for a punctuation character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int ispunct (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

ispunct returns a nonzero value when the argument is a punctuation character. Otherwise, a value of 0 is returned.

Remarks

The **ispunct** function or macro tests for any punctuation character, such as a comma (,) or a period (.).

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **islower**, **isprint**, **isspace**, **isupper**, **isxdigit**, **tolower**, **toupper**

isspace

Tests for a white-space character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isspace (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

isspace returns a nonzero value when the argument is one of the indicated white-space characters. Otherwise, a value of 0 is returned.

Remarks

The **isspace** function or macro tests for the following white-space characters:

`' '	Space
\f	Formfeed
\n	Newline or line feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, **isupper**, **isxdigit**, **tolower**, **toupper**

Example

isspace

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

main()
{
   int i;
   while((i = getch()) != '0')
   printf("%s\r\n",isspace(i) ? "yes" : "no");
}
```

isupper

Tests for an uppercase character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isupper (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

isupper returns a nonzero value when the argument is an uppercase character. Otherwise, a value of 0 is returned.

Remarks

The **isupper** function or macro tests for any uppercase character (A to Z) in the ASCII code set.

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, **isspace**, **isxdigit**, **tolower**, **toupper**

isxdigit

Tests for a hexadecimal-digit character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int isxdigit (
   int   c);
```

Parameters

c

(IN) Specifies the character to be tested.

Return Values

isxdigit returns a nonzero value when the argument is a hexadecimal digit. Otherwise, a value of 0 is returned.

Remarks

The **isxdigit** function or macro tests for any hexadecimal-digit character. These characters include digits (0 to 9) and letters (a to f or A to F).

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **tolower**, **toupper**

mblen

Determines the number of bytes comprising the multibyte character (nonoperational in NetWare® versions 3.11 and earlier)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Character Manipulation

Syntax

```
#include <stdlib.h>

int mblen (
   const char   *s,
   size_t       n);
```

Parameters

s

(IN) Points to the array of multibyte characters.

n

(IN) Specifies the number of bytes of the array pointed to by *s* to be examined.

Return Values

This function returns a value of 0 for any of the following conditions:

s = NULL pointer
s[0] = NULL
n = 0

Remarks

At most, *n* bytes of the array pointed to by *s* are examined.

See Also

mbstowcs, **mbtowc**, **wcstombs**, **wctomb**

mbstowcs

Converts a sequence of multibyte characters into their corresponding wide-character codes and stores them in an array (nonoperational in NetWare versions 3.11 and earlier)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Character Manipulation

Syntax

```
#include <stdlib.h>

size_t mbstowcs (
   wchar_t       *pwcs,
   const char    *s,
   size_t        n);
```

Parameters

pwcs

(OUT) Points to the array of wide-character codes.

s

(IN) Points to the array of multibyte characters to be converted.

n

(IN) Specifies the number of codes to be stored in the array pointed to by *pwcs*.

Return Values

mbstowcs returns the actual number of bytes that have been copied from the array pointed to by *s* to the array pointed to by *pwcs*. The returned value is always less than *n*.

Remarks

The **mbstowcs** function converts a sequence of multibyte characters pointed to by *s* into their corresponding wide-character codes and stores not more than *n* codes into the array pointed to by *pwcs*.

The **mbstowcs** function does not convert any multibyte characters beyond the NULL character. At most, *n* elements of the array pointed to by *pwcs* are modified.

This function is currently implemented for single-byte character coding only.

See Also

mblen, **mbtowc**, **wcstombs**, **wctomb**

mbtowc

Converts a single multibyte character into the wide-character code that corresponds to that multibyte character (nonoperational in NetWare versions 3.11 and earlier)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Character Manipulation

Syntax

```
#include <stdlib.h>

int mbtowc (
   wchar_t       *pwc,
   const char    *s,
   size_t        n);
```

Parameters

pwcs

(OUT) Points to a wide-character code.

s

(IN) Points to the array of multibyte characters to be converted.

n

(IN) Specifies the number of bytes of the array pointed to by *s* to be examined.

Return Values

This function returns a value of 0 for any of the following conditions:

s = NULL pointer

s[0] = NULL

n = 0

Remarks

The **mbtowc** function converts a single multibyte character pointed to by *s* into the wide-character code that corresponds to that multibyte character.

The code for the NULL character is zero. If the multibyte character is valid and *pwc* is not a NULL pointer, the code is stored in the object

pointed to by *pwc*. At most, *n* bytes of the array pointed to by *s* are examined.

This function is currently implemented for single-byte character coding only.

See Also

mblen, **mbstowcs**, **wcstombs**, **wctomb**

tolower

Converts an uppercase character to the corresponding lowercase character
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int tolower (
   int   c);
```

Parameters

c

(IN) Specifies the character to be converted to lowercase.

Return Values

The **tolower** function returns the corresponding lowercase character when the argument is an uppercase character. Otherwise, the original character is returned.

Remarks

The **tolower** function converts an uppercase character to the corresponding lowercase character in the ASCII code set.

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **strlwr**, **strupr**, **toupper**

Example

tolower

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

```
main()
{
   char s[] = "THIRD QUARTER REPORT";
   int i;
   for(i=0;s[i];i++) s[i] = tolower(s[i]);
   printf("%s\r\n",s);
   getch();
}
```

toupper

Converts a lowercase character to the corresponding uppercase character
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Character Manipulation

Syntax

```
#include <ctype.h>

int toupper (
   int   c);
```

Parameters

c

(IN) Specifies the character to be converted to uppercase.

Return Values

The **toupper** function returns the corresponding uppercase character when the argument is a lowercase letter. Otherwise, the original character is returned.

Remarks

The **toupper** function converts a lowercase character to the corresponding uppercase character in the ASCII code set.

See Also

isalnum, **isalpha**, **iscntrl**, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **strlwr**, **strupr**, **tolower**

Example

toupper

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

```
main()
{
   char s[] = "third quarter report";
   int i;
   for(i=0;s[i];i++) s[i] = toupper(s[i]);
   printf("%s\r\n",s);
   getch();
}
```

wcstombs

Converts a sequence of wide-character codes from an array into a sequence of multibyte characters (nonoperational in NetWare versions 3.11 and earlier)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Character Manipulation

Syntax

```
#include <stdlib.h>

size_t wcstombs (
   char            *s,
   const wchar_t   *pwcs,
   size_t          n);
```

Parameters

s

(OUT) Points to the array of multibyte characters.

pwcs

(IN) Points to the array of wide-character codes to be converted.

n

(IN) Specifies the number of bytes of the array pointed to by *s* to be modified.

Return Values

This function returns the actual number of bytes that have been copied from the array pointed to by *pwcs* to the array pointed to by *s*. The returned value is always less than *n*.

Remarks

The **wcstombs** function converts a sequence of wide-character codes from the array pointed to by *pwcs* into a sequence of multibyte characters and stores them in the array pointed to by *s*. The **wcstombs** function stops if a multibyte character would exceed the limit of *n* total bytes, or if the NULL character is stored. At most, *n* bytes of the array pointed to by *s* are modified.

See Also

mblen, mbstowcs, mbtowc, wctomb

wctomb

Determines the number of bytes required to represent the multibyte character corresponding to the specified code (nonoperational in NetWare versions 3.11 and earlier)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Character Manipulation

Syntax

```
#include <stdlib.h>

int wctomb (
   char      *s,
   wchar_t   wchar);
```

Parameters

s

(OUT) Points to the array of multibyte characters.

wchar

(IN) Specifies the wide-character code.

Return

wctomb returns a value of 0 when *s* is a NULL pointer. Otherwise, it returns a value of 1.

Remarks

The **wctomb** function determines the number of bytes required to represent the multibyte character corresponding to the code contained in *wchar*. If *s* is not a NULL pointer, the multibyte character representation is stored in the array pointed to by *s*. At most, MB_CUR_MAX characters are stored.

wctomb is currently implemented for single-byte character coding only.

See Also

mblen, **mbstowcs**, **mbtowc**, **wcstombs**

Device I/O

Device I/O: Functions

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

SMP Aware: No

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, **getche**, **gets**

Example

cgets

```
#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
SMP Aware: Yes
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, **printf**, **vfprintf**

Example

cprintf

```
#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
SMP Aware: No
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, **putch**

Example

cputs

```
#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
SMP Aware: Yes
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, **scanf**

Example

cscanf

To scan a date in the form "Saturday, April 18 1990":

```
#include <nwconio.h>
```

```
#include <nwconio.h>

main ()
{
   int    day, year;
   char   weekday[10], month[12];
   cscanf ("%s %s %d %d", weekday, month, &day, &year);
}
```

_disable

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

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
SMP Aware: No
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, **kbhit**

Example

getch

```
#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

SMP Aware: No

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, **kbhit**, **ungetch**

Example

getche

```
#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, 4.x
SMP Aware: No
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, **inpw**, **outp**, **outpd**, **outpw**

Example

inp

```
#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, 4.x
SMP Aware: No
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, **outp**, **outpd**, **outpw**

Example

inpd

```
#include <nwconio.h>
#define DEVICE 34
```

```
#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, 4.x
SMP Aware: No
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, **inpd**, **outp**, **outpd**, **outpw**

Example

inpw

```
#include <nwconio.h>
#define DEVICE 34
```

```
#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
SMP Aware: No
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, **getche**, **putch**, **ungetch**

Example

kbhit

```
#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

SMP Aware: Yes

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, **vfprintf**, **cprintf**

Example

cprintf

```
#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, 4.x

SMP Aware: No

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int outp (
   int   port,
   int   value);
```

Parameters

port

(IN) Specifies the hardware port.

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, **inpd**, **inpw**, **outpd**, **outpw**

Example

outp

```
#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, 4.x

SMP Aware: No

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int outpd (
   int            port,
   unsigned int   value);
```

Parameters

port

(IN) Specifies the hardware port.

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, **inpw**, **outp**

Example

outpd

```
#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, 4.x

SMP Aware: No

Service: Device I/O

Syntax

```
#include <nwconio.h>

unsigned int outpw (
   int            port,
   unsigned int   value);
```

Parameters

port

(IN) Specifies the hardware port.

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, **inpd**, **inpw**, **outp**, **outpd**

Example

outpw

```
#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
SMP Aware: No
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, **getche**, **ungetch**

Example

putch

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

main ()
```

```
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
SMP Aware: No
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:

F1	0x3B00
F2	0x3C00
F3	0x3D00
F4	0x3E00

F5	0x3F00
F6	0x4000
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 Loading an NLM from an NLM: Example.

See Also

getch, getche

vcprintf

Writes output to the console under format control
Local Servers: blocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
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, **printf**, **sprintf**, **va_arg**, **va_end**, **va_start**, **vprintf**

Example

vcprintf

vcprintf

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
SMP Aware: Yes
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**, **va_arg**, **va_end**, **va_start**, **vscanf**

Example

vcscanf

```
#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 );
}
```

Library

Library: Guides

Library: Concept Guide

The Library API functions are for use by library NLM™ applications (NLM applications that export functions to be called by other NLM applications).

Unique Problems of Library NLM Applications

Library API Example

Library Function List

Library: Functions

Library: Concepts

Unique Problems of Library NLM Applications

NLM applications that are libraries pose a few unique problems:

If a library NLM allocates resources such as memory on behalf of another NLM, the library NLM needs to know when its clients are unloaded or otherwise terminated so that it can free any resources it allocated on the client's behalf.

In many cases, a library NLM needs to have a different instance of data structures for each of its clients. The library needs some way of associating or getting to the proper client data structure whenever it is called by a client.

Sometimes, a library NLM may want to allocate resources on a client's behalf; other times, it may want to allocate resources on its own behalf even though it has been called by a client.

The Library API solves these problems as follows:

It allows a library to register a cleanup function that executes whenever one of its clients terminates.

It allows you to save and retrieve a data area pointer on a per-client NLM basis.

It allows you to specify which NLM resources are allocated when the library allocates resources.

NOTE: It is possible to write a library NLM without using the Library API.

Library API Example

The following example demonstrates how the Library API functions might be used in a "typical" library (Library X). In this example, Library X exports three functions:

LoginToX

DoTheWorkOfX

LogoutFromX

Library X does any internal initialization before it is called by any NLM applications:

```
int main (int argc, char * argv[])
{

/* Initialize myself */
.
.
.
```

Library X registers itself with the NetWare® API (CLIB) using **RegisterLibrary**. **RegisterLibrary** must be called prior to other functions that require a library handle.

Library X saves its own NLM ID for later use. The library specifies which cleanup function should be called whenever a library client is terminated.

```
XNLMID = GetNLMID ();

AtUnload (LibraryXunloadFunc);   /* Define function to be
                                    executed if Library X is
                                    ever UNLOADed.*/
.
.
.
XLibHandle = RegisterLibrary (LibraryXClientCleanupFunc);
```

The library terminates its initial thread since it is not needed. (The library's functions are run by the client's thread of execution.)

```
.
.
.
ExitThread (TSR_THREAD, 0);
}
```

TSR_THREAD specifies to terminate the thread, but not to terminate the NLM, even if the last thread of the NLM is being terminated.

Whenever a client NLM wants to use Library X, the client NLM must first call **LoginToX**. This is not a restriction of the NetWare API, but a requirement that Library X imposes.

```
int LoginToX (int parm1, char * parm2, etc.)
{
```

In **LoginToX**, Library X allocates one ClientStruct for each client:

```
ClientStruct *newClientStructPtr;
newClientStructPtr = (ClientStruct *) malloc (
      sizeof (clientStruct));
if (newClientStructPtr == NULL)
```

```
   return -1;                        /* Code defined by Library X.
                                        Returned if LoginToX fails.*/
```

Library X uses one of the Library API functions, **SaveDataAreaPtr**, to associate the new ClientStruct with this particular client:

```
if (SaveDataAreaPtr (XLibHandle, newClientStructPtr))
   return -1;                    /* Error code defined by Library X.
                                    Returned if LoginToX fails.*/
```

LoginToX is successful, so it returns its success code:

```
return 0;
}
```

A Library X client then calls **DoTheWorkOfX**, which does the real work of Library X:

```
int DoTheWorkOfX (long parm1, short parm2, etc.)
```

DoTheWorkOfX calls the Library API function **GetDataAreaPtr** to retrieve the pointer to the ClientStruct that was saved when this client called **LoginToX**:

```
ClientStruct *clientStructPtr;
clientStructPtr = GetDataAreaPtr (XLibHandle);

if (clientStructPtr == NULL)
   return -2;                /* Return code defined by Library
                                X. Passed back when
                                DoTheWorkOfX is called
                                without first calling LoginToX.*/
```

DoTheWorkOfX performs the work, which includes allocating another data structure for the client. This data structure is pointed to by a field in ClientStruct, which means Library X can later get to this additional structure because it can get to ClientStruct by calling **GetDataAreaPtr**.

Suppose also that, during the execution of **DoTheWorkOfX,** an SPX™ socket must be opened to get data from some other source. Because of the way Library X is designed (Library X uses the same SPX socket for all clients, once it is opened), the open SPX socket should be considered a resource of Library X, not a resource of the client NLM. If the socket was a resource of the client, the socket would be closed when the client was terminated, which would be inappropriate with the way Library X is designed. So Library X does the following to cause the SPX socket to be counted as Library X's own resource:

```
clientNLMID = SetNLMID (XNLMID);    /* Switch to Library X.*/

if (SpxOpenSocket (&Xsocket) != 0)

   /* Do the action for failure of SpxOpen
```

```
SetNLMID (clientNLMID);             /* Switch back to client.*/
```

NOTE: Switching back to clients is very important. If this is not done, any resources the client allocates are considered Library X's resources.

After **DoTheWorkOfX** completes, it returns to the client. The client calls **DoTheWorkOfX** as many times as it needs to. When the client no longer needs to use Library X, it calls **LogoutFromX**. Calling **LogoutFromX** is part of the design of Library X, not of the NetWare API.

```
int LogoutFromX (int parm1, char * parm2, etc.)

ClientStruct *clientStructPtr;
clientStructPtr = GetDataAreaPtr (XLibHandle);
if (clientStructPtr == NULL)
   return -2;

/* Cleanup all the resources allocated for the client */
.
.
.
/* Indicate that this client NLM is no longer a client */
   SaveDataAreaPtr(XLibHandle, NULL);

return 0;                  /* Successfully logged out from X.*/
```

If the client of Library X was **UNLOAD**ed (or otherwise terminated) before it called **LogoutFromX**, then Library X's client cleanup function (**LibraryXClientCleanupFunc**) would be called. The client cleanup function receives one argument, which is the work area pointer (*clientStructPtr*) saved by Library X when it called **SaveDataAreaPtr** in **LoginToX**:

```
int LibraryXClientCleanupFunc (ClientStruct *clientStructPtr)
{

/*
   Clean up all the resources allocated for the client
   Very similar to LogoutFromX
*/
.
.
.
/*
   Not necessary to call SaveDataAreaPtr (XLibHandle, NULL)
   here, because it is done automatically when a client
   terminates
*/

return 0;      /* Zero indicates success.
                  Nonzero indicates failure.
                  This return code is ignored at present.*/
}
```

Library X can service many client NLM applications. In many cases, Library X stays loaded as long as the server is up. In some cases, Library X can be **UNLOAD**ed. When it is, its function, **LibraryXUnloadFunc**, is called because Library X called **AtUnload** to define an **UNLOAD** function:

```
void LibraryXUnloadFunc (void)
{

/*
   Clean up any resources allocated on Library X's own behalf
   such as the SPX socket mentioned earlier
*/
.
.
.
SpxCloseSocket (XSocket);
.
.
.

/*
   Deregister Library X as a library with the
   NetWare API Library
*/

DeregisterLibrary (XLibHandle);

return;
}
```

This completes the example of a typical library NLM.

Library Function List

DeregisterLibrary

Deregisters a previously registered library NLM.

GetDataAreaPtr

Returns a saved data area pointer for the current NLM.

RegisterLibrary

Registers an NLM as a library with NetWare® API.

SaveDataAreaPtr

Associates a data area pointer with a client NLM.

Library: Functions

DeregisterLibrary

Deregisters a previously registered library NLM™ application
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Library

Syntax

```
#include <nwlib.h>

int DeregisterLibrary (
   LONG   libraryHandle);
```

Parameters

libraryHandle

(IN) Specifies the handle returned by **RegisterLibrary**.

Return Values

0	(0x00)	ESUCCESS	
22	(0x16)	EBADHNDL	Invalid library.handle was passed in.

Remarks

Typically, the **DeregisterLibrary** function is called from the library's **AtUnload** function.

See Also

RegisterLibrary

GetDataAreaPtr

Returns a previously saved data area pointer for the current NLM
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Library

Syntax

```
#include <nwlib.h>

void *GetDataAreaPtr (
   LONG   libraryHandle);
```

Parameters

libraryHandle

(IN) Specifies the handle returned by a call to **RegisterLibrary**.

Return Values

This function returns the data area pointer if successful. If a library has called **SetNLMID** before calling **GetDataAreaPtr**, this function can return NULL.

If an error occurs, it returns EFAILURE and sets *errno* to:

2 2	(0x1 6)	EBADHN DL	Invalid library handle was passed in.

Remarks

GetDataAreaPtr returns the data area pointer saved by a previous **SaveDataAreaPtr** call for the current NLM (usually the NLM calling the library; however, see **GetNLMID** and **SetNLMID**) and for the specified library (specified by the *libraryHandle* parameter).

See Also

GetNLMID, **SaveDataAreaPtr**, **SetNLMID**

Example

GetDataAreaPtr

```
#include <nwlib.h>

ClientStruct   *dataAreaPtr;
LONG            libraryHandle;
dataAreaPtr = GetDataAreaPtr (libraryHandle);
```

RegisterLibrary

Registers an NLM as a library with the NetWare® API
Local Servers: blocking
Remote Servers: nonblocking
Classification: 3.x, 4.x
SMP Aware: No
Service: Library

Syntax

```
#include <nwlib.h>

LONG RegisterLibrary (
   int   (*clientCleanupFunc) (void *));
```

Parameters

clientCleanupFunc

(IN) Specifies a function to be called whenever one of the clients of the library is terminated by either an **UNLOAD** command or because the NLM terminated itself by calling **exit**, **abort**, **ExitThread**, or so on (see "Note" below).

Return Values

This function returns a library handle or a value of 0xFFFFFFFF if an error occurs.

Remarks

The **RegisterLibrary** function must be called prior to any other function requiring a library handle.

NOTE: The prototype of the *clientCleanupFunc* indicates that it returns a value of type (int). Although NetWare currently ignores this value, *clientCleanupFunc* functions should always return ESUCCESS (or zero).

NOTE: It is possible to write a library NLM without using the Library functions.

For an NLM to be considered a client of a registered library by the NetWare API, the library must call **SaveDataAreaPtr** with a nonNULL data area pointer while the client NLM is the current NLM (this is usually done when the client makes its first call to the library). Only NLM applications that are clients of registered library NLM applications cause a client cleanup function to be called when they terminate.

NOTE: The library clean-up routines must be given CLIB context if they use NLM API functions that require context. You can set the context using **SetThreadGroupID** but you must set it to a thread group ID that is part of the library, since the cleanup routine is part of the library, not part of your NLM. You should save the default thread group ID when you enter your routine and restore it, using **SetThreadGroupID**, before you leave your routine.

See Also

DeregisterLibrary, **SaveDataAreaPtr**

Example

RegisterLibrary

EXAMPLE OF A LIBRARY NLM

```
#include <stdio.h>
#include <nwconio.h>
#include <nwlib.h>

int   LibHandle;
int      threadGroupID;

/*.......................*/
int LibCleanupFunc
(
void *data
)
{
   int    curThreadGroupID;
   void   *dataAreaPtr;

   data = data;
   /*...we must establish context for the thread running
      the cleanup function...*/

   curThreadGroupID = SetThreadGroupID(threadGroupID);
   printf("Data ptr : %lX\n\n", data);
   printf("%lX Client closed.\n\n", GetThreadID());

   /*...restore the running thread's original context...*/
   SetThreadGroupID(curThreadGroupID);
   return(0);
}
/*.......................*/

main()
{
```

```
   /*...save the thread group ID of the main lib thread group...*/
   threadGroupID = GetThreadGroupID();
   LibHandle = RegisterLibrary(LibCleanupFunc);
   if (LibHandle != -1)
      SuspendThread(GetThreadID());
   else
      ConsolePrintf("\n\nUnable to register library.\n\n");
}
/*.......................*/
```

EXAMPLE OF A CLIENT NLM

```
#include <stdio.h>
#include <nwconio.h>
#include <nwlib.h>
extern   int   LibHandle;
/*.......................*/
main()
{
   void   *dataAreaPtr;
   dataAreaPtr = (void *)0x11223344;
   /*...become a client of the library...*/
   if (SaveDataAreaPtr(LibHandle, dataAreaPtr))
      ConsolePrintf("\n\nUnable to get data area ptr for the
                     library.\n\n");
   getch();
}
/*.......................*/
```

SaveDataAreaPtr

Associates a data area pointer with a particular client NLM
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Library

Syntax

```
#include <nwlib.h>

int SaveDataAreaPtr (
   LONG   libraryHandle,
   void   *dataAreaPtr);
```

Parameters

libraryHandle

(IN) Specifies the handle returned by calling **RegisterLibrary**.

dataAreaPtr

(IN) Points to a client NLM data area.

Return Values

0	(0x00)	ESUCCESS	
5	(0x5)	ENOMEM	Not enough memory.

Remarks

For a client NLM to be considered a client of a registered library by the NetWare API, the library must call **SaveDataAreaPtr** with a nonNULL data area pointer while the client is the current NLM (this is usually done when the client makes its first call to the library).

This function is normally used by library NLM applications to save a pointer to a data area the library allocates for each client. However, the *dataAreaPtr* parameter does not necessarily have to be a pointer, it can be an index into an array or anything else the library wants to associate with each client.

See Also

GetDataAreaPtr

Mathematical Computation

Mathematical Computation: Functions

NOTE: The Mathematical Computation functions are exported by MATHLIB.NLM and MATHLIBC.NLM.

Use MATHLIBC if an 80387 numeric data processor is not installed.

Use MATHLIB if an 80387 numeric data processor is installed.

abs

Returns the absolute value of its integer argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

int abs (
   int   j);
```

Parameters

j
(IN) Specifies an integer argument.

Return Values

abs returns the absolute value of its integer argument.

Remarks

abs returns the absolute value of its integer argument *j*.

See Also

fabs, **labs**

Example

abs

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

main ()
{
   printf ("%d %d %d\n", abs (-5), abs (0), abs (5) );
}
```

produces the following:

produces the following:

```
5      0      5
```

acos

Computes the principal value of the arc cosine of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double  acos (
   double   x);
```

Parameters

x

(IN) Specifies an argument to compute the arc cosine for.

Return Values

The **acos** function returns the arc cosine in the range (0,p). When the argument is outside the permissible range, *errno* is set and **matherr** is called.

Remarks

A domain error occurs for arguments not in the range (-1,1).

See Also

asin, **atan**, **atan2**, **matherr**

Example

acos

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

main ()
{
   printf ("%f\n", acos (.5) );
```

```
}
```

produces the following:

```
1.047197
```

asin

Computes the principal value of the arc sine of the specified argument.
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double asin (
   double   x);
```

Parameters

x

(IN) Specifies an argument whose arc sine is to be computed.

Return Values

The **asin** function returns the arc sine in the range (-pi/, 2pi/2). When the argument is outside the permissible range, *errno* is set and **matherr** is called.

Remarks

A domain error occurs for arguments not in the range (-1,1).

See Also

acos, **atan**, **atan2**, **matherr**

Example

asin

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

main ()
{
   printf ("%f\n",  asin (.5) );
```

```
}
```

produces the following:

```
0.523599
```

atan

Computes the principal value of the arc tangent of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double atan (
   double   x);
```

Parameters

x
(IN) Specifies an argument whose arc tangent is to be computed.

Return Values

The **atan** function returns the arc tangent in the range (-pi/2, pi/2).

See Also

acos, **asin**, **atan2**

Example

atan

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

main ()
{
   printf ("%f\n", atan(.5) );
}
```

produces the following:

```
0.463648
```

atan2

Computes the principal value of the arc tangent of *y/x,* using the signs of both arguments to determine the quadrant of the return value

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Mathematical Computation

Syntax

```
#include <math.h>

double atan2 (
   double   y,
   double   x);
```

Parameters

x and *y*

(IN) Specifies the arguments whose arc tangent is to be computed.

Return Values

The **atan2** function returns the arc tangent of *y/x,* in the range (-pi,pi). When the arguments are outside the permissible range, *errno* is set and **matherr** is called.

Remarks

A domain error occurs if both arguments are zero.

See Also

acos, **asin**, **atan**, **matherr**

Example

atan2

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

main ()
```

```
{
   printf ("%f\n", atan2 (.5, 1.) );
}
```

produces the following:

```
0.463648
```

Bessel Functions

Returns the result of the desired Bessel function of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: No
Service: Mathematical Computation

Syntax

```
#include <math.h>

double j0 (
   double   x);

double j1 (
   double   x);

double jn (
   int      n,
   double   x);

double y0 (
   double   x);

double y1 (
   double   x);

double yn (
   int      n,
   double   x);
```

Return Values

The result of the desired Bessel function of the argument *x* is returned. For **y0**, **y1**, or **yn**, if *x* is negative the routine sets *errno* to EDOM, prints a DOMAIN error message to *stderr*, and returns -HUGE_VAL.

Remarks

Functions **j0**, **j1**, and **jn** return Bessel functions of the first kind.

Functions **y0**, **y1**, and **yn** return Bessel functions of the second kind. The argument *x* must be positive.

See Also

matherr

Example

Bessel Functions

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

main ()
{
    double x, y, z;
     x = j0( 2.4 );
     y = y1( 1.58 );
     z = jn( 3, 2.4 );
     printf( "j0(2.4) = %f, y1(1.58) = %f\n", x, y );
     printf( "jn(3,2.4) = %f\n", z );
}
```

cabs

Computes the absolute value of the complex number value by a square root calculation

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: Mathematical Computation

Syntax

```
#include <math.h>

double cabs (
   struct complex   value);
```

Parameters

value

(IN) Value of the complex number.

Return Values

The absolute value is returned.

Remarks

The struct complex structure is defined as

```
struct complex
{
   double real;
   double imag;
};
```

where *real* is the real part and *imag* is the imaginary part. In certain cases, overflow errors can occur that cause the **matherr** routine to be invoked.

The **cabs** function computes the absolute value of the complex number *value* by a calculation that is equivalent to

```
sqrt ((value.real * value.real) + (value.imag * value.imag))
```

Example

cabs

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

main ()
{
   struct complex c = {-3.0, 4.0};
   printf ("%f\n", cabs (c) );
}
```

produces the following:

```
5.000000
```

ceil

Computes the smallest integer not less than *x*
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double ceil (
   double   x);
```

Parameters

x
(IN) Specifies an argument.

Return Values

The **ceil** function returns the smallest integer not less than *x*, expressed as a double type.

Remarks

The **ceil** function (ceiling function) computes the smallest integer not less than *x*:

```
(ceil (x) ::= -floor (-x))
```

See Also

floor

Example

ceil

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

main ()
{
```

```
{
   printf ("%f %f %f %f %f\n", ceil (-2.1), ceil (-2.),
   ceil (0.0), ceil (2.), ceil (2.1) );
}
```

produces the following:

```
-2.000000 -2.000000 0.000000 2.000000 3.000000
```

cos

Computes the cosine of the argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double cos (
   double   x)
```

Parameters

x
(IN) Argument whose cosine is to be computed.

Return Values

The **cos** function returns the cosine value.

Remarks

The **cos** function computes the cosine of *x* (measured in radians). A large magnitude argument can yield a result with little or no significance.

See Also

acos, **sin**, **tan**

Example

cos

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

main ()
{
   double value;
   value = cos (3.1415278);
```

```
}
```

cosh

Computes the hyperbolic cosine of the argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double cosh (
   double   x);
```

Parameters

x

(IN) Specifies the argument whose hyperbolic cosine is to be computed.

Return Values

The **cosh** function returns the hyperbolic cosine value. When the argument is outside the permissible range, *errno* is set and **matherr** is called.

Remarks

The **cosh** function computes the hyperbolic cosine of *x*. A range error occurs if the magnitude of *x* is too large.

See Also

matherr, **sinh**, **tanh**

Example

cosh

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

main ()
```

```
{
   printf ("%f\n", cosh (.5) );
}
```

produces the following:

```
1.127626
```

difftime

Calculates the difference between two calendar times (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <time.h>

double difftime (
   time_t   time1,
   time_t   time0);
```

Parameters

time1

(IN) Specifies the calendar time to compare to *time0*.

time0

(IN) Specifies the calendar time to compare to *time1*

Return Values

difftime returns the difference between the two times in seconds as a double type.

Remarks

The **difftime** function or macro calculates the difference between *time1* and *time0* (*time1* - *time0*).

See Also

asctime, asctime_r, clock, ctime, ctime_r, gmtime, gmtime_r, localtime, localtime_r, mktime, strftime, time

Example

difftime

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

```
#include <time.h>

void time_test ()
{
   extern compute ();
   time_t start_time, end_time;
   start_time = time (NULL);
   compute();
   end_time = time (0);
   printf("Elapsed time: %u seconds", (unsigned int) difftime(end_time,
          start_time));
}
```

produces the following:

```
Elapsed time: 14 seconds
```

div

Calculates the quotient and remainder (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

div_t div (
   int   number,
   int   denom);
```

Parameters

numer

(IN) Specifies the numerator.

denom

(IN) Specifies the denominator.

Return Values

div returns a structure of type div_t, which contains the fields *quot* and *rem*.

Remarks

The **div** function or macro calculates the quotient and remainder of the division of the numerator *numer* by the denominator *denom*.

See Also

ldiv

Example

div

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

```
void print_time (int seconds)
{
   auto div_t min_sec;
   min_sec = div (seconds, 60);
   printf ("It took %d minutes and %d seconds\n",
      min_sec.quot, min_sec.rem);
}
```

produces the following:

```
It took 2 minutes and 23 seconds
```

exp

Computes the exponential function of the argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double exp (
   double   x);
```

Parameters

x

(IN) Specifies the argument whose exponential function is to be computed.

Return Values

The **exp** function returns the exponential value. When the argument is outside the permissible range, *errno* is set and **matherr** is called.

Remarks

The **exp** function computes the exponential function of *x*. A range error occurs if the magnitude of *x* is too large.

See Also

log, matherr

Example

exp

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

main ()
{
```

```
    printf ("%f\n", exp (.5) );
}
```

produces the following:

```
1.648721
```

fabs

Computes the absolute value of the argument (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double fabs (
   double   x);
```

Parameters

x

(IN) Specifies an argument whose absolute value is to be computed.

Return Values

The **fabs** function or macro returns the absolute value of *x*.

Example

fabs

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

main ()
{
   printf ("%f %f\n", fabs (.5), fabs (-.5) );
}
```

produces the following:

```
0.500000 0.500000
```

floor

Computes the largest integer not greater than the argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double floor (
   double   x);
```

Parameters

x
(IN) Specifies an argument.

Return Values

The **floor** function computes the largest integer not greater than *x*, expressed as a double type.

Remarks

The **floor** function computes the largest integer not greater than *x*.

See Also

ceil, **fmod**

Example

floor

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

main ()
{
   printf ("%f\n", floor (-3.14) );
   printf ("%f\n", floor (-3.) );
```

```
    printf ("%f\n", floor (0.) );
    printf ("%f\n", floor (3.14) );
    printf ("%f\n", floor (3.) );
}
```

produces the following:

```
-4.000000
-3.000000
 0.000000
 3.000000
 3.000000
```

fmod

Computes the floating-point remainder of *x*/*y*
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double fmod (
   double   x,
   double   y);
```

Parameters

x and *y*

(IN) Specifies the arguments for which the floating-point remainder is to be computed.

Return Values

The **fmod** function returns the value *x* - (*i* * *y*), for some integer *i* such that, if *y* is nonzero, the result has the same sign as *x* and a magnitude less than the magnitude of *y*. If the value of *y* is zero, then the value returned is zero.

Remarks

The **fmod** function computes the floating-point remainder of *x*/*y*, even if the quotient *x*/*y* is not representable.

See Also

ceil, **fabs**, **floor**

Example

fmod

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

```
main ()
{
   printf ("%f\n", fmod (4.5,  2.0) );
   printf ("%f\n", fmod (-4.5,  2.0) );
   printf ("%f\n", fmod (4.5, -2.0) );
   printf ("%f\n", fmod (-4.5, -2.0) );
}
```

produces the following:

```
 0.500000
-0.500000
 0.500000
-0.500000
```

frexp

Breaks a floating-point number into a normalized fraction and an integral power of 2

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Mathematical Computation

Syntax

```
#include <math.h>

double frexp (
   double   value,
   int      *exp);
```

Parameters

value

(IN) Original double value.

exp

(OUT) Receives a pointer to the object.

Return Values

The **frexp** function returns the value of *x*, such that *x* is a double with magnitude in the interval (0.5,1) or zero, and *value* equals *x* times 2 raised to the power **exp*. If *value* is zero, then both parts of the result are zero.

Remarks

The **frexp** function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integral power of 2 in the int object pointed to by *exp*.

See Also

ldexp, **modf**

Example

frexp

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

main ()
{
   int expon;
   printf ("%f %d\n", frexp (4.25, &expon), expon);
   printf ("%f %d\n", frexp (-4.25, &expon), expon);
}
```

produces the following:

```
 0.531250 3
-0.531250 3
```

hypot

Computes the length of the hypotenuse of a right triangle whose sides are *x* and *y* adjacent to that right angle

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: No

Service: Mathematical Computation

Syntax

```
#include <math.h>

double hypot (
   double   x,
   double   y);
```

Parameters

x and *y*

(IN) Sides of a right triangle adjacent to the right angle.

Return Values

The value of the hypotenuse is returned. When an error has occurred, *errno* is set.

Remarks

The **hypot** function computes the length of the hypotenuse of a right triangle whose sides are *x* and *y* adjacent to that right angle. The calculation is equivalent to

```
sqrt (x*x + y*y)
```

The computation might cause an overflow, in which case **matherr** is called.

Example

hypot

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

```
main ()
{
   printf ("%f\n", hypot (3.0, 4.0) );
}
```

produces the following:

```
5.000000
```

j0, j1, jn

See **Bessel Functions**

labs

Returns the absolute value of its argument (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

long int labs (
   long int   j);
```

Parameters

j

(IN) Specifies the argument for which the absolute value is to be returned.

Return Values

The **labs** function or macro returns the absolute value of its argument. There is no error return.

Remarks

The **labs** function or macro returns the absolute value of argument*j*.

See Also

abs, **fabs**

Example

labs

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

main ()
{
   long x, y;
```

```
    x = -50000L;
    y = labs (x);
}
```

ldexp

Multiplies a floating-point number by an integral power of 2
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Mathematical Computation

Syntax

```
#include <math.h>

double ldexp (
   double   x,
   int      exp);
```

Parameters

j

(IN) Specifies the floating-point number to be multiplied by an integral power of 2.

exp

(IN) Specifies the integral power of 2.

Return Values

The **ldexp** function returns the value of *x* times 2 raised to the power *exp*.

Remarks

The **ldexp** function multiplies a floating-point number by an integral power of 2. A range error can occur.

See Also

frexp, **modf**

Example

ldexp

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

```
main ()
{
   double value;
   value=ldexp( 4.7072345, 5 );
   printf ( "%f\n", value );
}
```

produces the following:

```
150.631504
```

ldiv

Calculates the quotient and remainder of the division of a number
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

ldiv_t ldiv (
   long int   numer,
   long int   denom);
```

Parameters

numer

(IN) Specifies the numerator.

denom

(IN) Specifies the denominator.

Return Values

This function returns a structure of type ldiv_t that contains the fields *quot* and *rem*, which are both of type long int.

Remarks

The **ldiv** function calculates the quotient and remainder of the division of the numerator *numer* by the denominator *denom*.

See Also

div

Example

ldiv

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

```
void print_time (long int ticks);
{
   ldiv_t sec_ticks;
   ldiv_t min_sec;
   sec_ticks = ldiv (ticks, 100L);
   min_sec = ldiv (sec_ticks.quot, 60L);
   printf ("It took %ld minutes and %ld seconds\n",
      min_sec.quot, min_sec.rem);
}
```

produces the following:

```
It took 14 minutes and 27 seconds
```

log

Computes the natural logarithm (base *e*) of *x*
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double log (
   double   x);
```

Parameters

x

(IN) Specifies the argument for which the natural logarithm is to be computed.

Return Values

The **log** function returns the natural logarithm of the argument. When the argument is outside the permissible range, *errno* is set and **matherr** is called. Unless **matherr** is replaced, **matherr** prints a diagnostic message using the *stderr* stream.

Remarks

A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

See Also

exp, **log10**, **matherr**, **pow**

Example

log

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

```
main ()
{
   printf ("%f\n", log (.5) );
}
```

produces the following:

```
-0.693147
```

log10

Computes the logarithm (base 10) of *x*
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double log10 (
   double   x);
```

Parameters

x

(IN) Specifies the argument for which the logarithm is to be computed.

Return Values

The **log10** function returns the logarithm (base 10) of the argument. When an error has occurred, *errno* is set.

Remarks

The **log10** function computes the logarithm (base 10) of *x*. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

See Also

exp, **log**, **pow**

Example

log10

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

main()
{
```

```
    printf ("%f\n", log10 (.5) );
}
```

produces the following:

```
-0.301030
```

matherr

Invoked each time an error is detected by functions in the math library
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

int matherr (
   struct exception   *err_info);
```

Parameters

err_info

(OUT) Receives a pointer to the exception structure that contains information about the detected error.

Return Values

The **matherr** function returns zero when an error message is to be printed; otherwise it returns a nonzero value. The default **matherr** always returns zero.

Remarks

The **matherr** function is invoked each time an error is detected by functions in the math library. The default **matherr** supplied in the library returns zero, which causes an error message to be displayed upon *stderr* and *errno* to be set with an appropriate error value. An alternative version of this function can be provided, so that mathematical errors are handled by an application.

By calling **RegisterMatherrHandler**, you can provide a developer-written version of **matherr** to take any appropriate action when an error is detected. When zero is returned, an error message is printed upon *stderr* and *errno* is set as was the case with the default function. When a nonzero value is returned, no message is printed and *errno* is not changed. The value *err_info*->*retval* is used as the return value for the function in which the error was detected.

The **matherr** function is passed a pointer to a structure of type struct exception, which contains information about the error that has been

detected:

```
struct exception
{
   int      type;         /* Type of error */
   char     *name;        /* Name of function */
   double   arg1;         /* First argument to function */
   double   arg2;         /* Second argument to function */
   double   retval;       /* Default return value */
};
```

See exception. Only *retval* can be changed by a developer-supplied version of **matherr**.

See Also

RegisterMatherrHandler

Example

matherr

```
#include <math.h>
#include <string.h>
#include <stdio.h>

/*
   Demonstrate error routine in which negative
   arguments to "sqrt" are treated as positive
*/
int main ()
{
   RegisterMatherrHandler (altmatherr);
   printf ("%e\n", sqrt (-5e0) );
   exit ( 0 );
}

int altmatherr (struct exception *err);
{
   if (strcmp (err->name, "sqrt") == 0)
   {
      if (err->type == DOMAIN)
      {
         err->retval = sqrt ( - (err->arg1) );
         return (1);
      }
      else
         return (0);
   }
   else
      return (0);
```

```
}
```

max

Returns the larger of two integers
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

int max (
   int   a,
   int   b);
```

Parameters

a

(IN) Specifies the first integer.

b

(IN) Specifies the second integer.

Return Values

The **max** function returns the larger of the two integers.

See Also

min

Example

max

```
#include <stdlib.h>

external int   ComputeValue1 (void);
external int   ComputeValue2 (void);

main()
{
   int   maxValue;
   value1 = ComputeValue1();
```

```
    value2 = ComputeValue2();
    maxValue = max (value1, value2);
}
```

min

Returns the smaller of two integers
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

int min (
   int   a,
   int   b);
```

Parameters

a

(IN) Specifies the first integer.

b

(IN) Specifies the second integer.

Return Values

The **min** function returns the smaller of the two integers.

See Also

max

Example

min

```
#include <stdlib.h>

external int ComputeValue1 (void);
external int ComputeValue2 (void);

main()
{
   int minValue;
   value1 = ComputeValue1();
```

```
    value2 = ComputeValue2();
    minValue = min (value1, value2);
}
```

modf

Breaks the argument value into integral and fractional parts
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double modf (
   double   value,
   double   *iptr);
```

Parameters

value

(IN) Specifies the value to be broken into integral and fractional parts.

iptr

(OUT) Receives a pointer to the object into which the integral part is stored as a double.

Return Values

The **modf** function returns the signed fractional part of *value*.

Remarks

The **modf** function breaks the argument *value* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a double in the object pointed to by *iptr*.

See Also

frexp, **ldexp**

Example

modf

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

```
#include <stdio.h>

main ()
{
   double integral_value, fractional_part;
   fractional_part = modf (4.5, &integral_value);
   printf ("%f %f\n", fractional_part, integral_value);
   fractional_part = modf (-4.5, &integral_value);
   printf ("%f %f\n", fractional_part, integral_value);
}
```

produces the following:

```
 0.500000  4.000000
-0.500000 -4.000000
```

pow

Computes *x* raised to the power *y*
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double pow (
   double   x,
   double   y);
```

Parameters

x

(IN) Specifies the argument to be raised to the power *y*.

y

(IN) Specifies the integral power.

Return Values

The **pow** function returns the value of *x* raised to the power *y*. When the argument is outside the permissible range, *errno* is set and **matherr** is called. Unless the function is replaced, **matherr** prints a diagnostic message using the *stderr* stream.

Remarks

The **pow** function computes *x* raised to the power *y*. A domain error occurs if *x* is zero and *y* is less than or equal to 0, or if *x* is negative and *y* is not an integer. A range error can occur.

See Also

exp, **log**, **sqrt**

Example

pow

```
#include <math.h>
#include <stdio.h>
main ()
{
     printf ("%f\n", pow (1.5, 2.5) );
}
```

produces the following:

```
2.755676
```

rand, rand_r

Computes a sequence of pseudo-random integers in the range 0 to RAN_MAX (32767)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Platform: NLM

SMP Aware: No

Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

int rand (void);

#include <stdlib.h>

int rand_r (
   unsigned long   *seed,
   int             *value);
```

Parameters

seed

(IN/OUT) Points to a seed value (updated with each iteration).

value

(OUT) Points to a place to store the returned value.

Return Values

Returns the value stored in the *value* parameter (representing a pseudo-random integer).

If the *seed* parameter is set to NULL, -1 will be returned and *errno* will be set to EINVAL.

If the *value* parameter is set to NULL, a value will be returned but the value will not be stored.

Remarks

rand_r is the same as **rand** except that the caller passes storage for the seed and the result rather than relying on a global variable (which has the potential of being modified by subsequent calls to **rand** by other

threads in the same NLM).

rand_r is supported only in CLIB V 4.11 or above.

The sequence of pseudo-random integers can be started at different values by calling the **srand** function.

See Also

srand

RegisterMatherrHandler

Substitutes a custom routine for **matherr**
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

int RegisterMatherrHandler (
   int   (*newFunc) (
      struct exception *));
```

Parameters

newFunc
(IN) Points to the routine to be used as **matherr**.

Return Values

RegisterMatherrHandler returns zero if successful. If a nonzero value is returned, an error occurred resulting in an error handler being registered. If an error occurs, it is most likely because the calling thread does not have CLib context.

Remarks

RegisterMatherrHandler registers a routine (*newFunc*) to be called in place of **matherr**.

If a routine has already been registered, **RegisterMatherrHandler** returns a nonzero value. The following deregisters a previously registered **matherr** handler:

```
RegisterMatherrHandler(NULL);
```

The routine that is registered must meet the requirements of **matherr**.

See Also

matherr

Example

See the example for **matherr**.

sin

Computes the sine of the argument (in radians)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double sin (
   double   x);
```

Parameters

x

(IN) Specifies the argument whose sine is to be computed.

Return Values

The **sin** function returns the sine value.

Remarks

The **sin** function computes the sine of *x* (measured in radians). A large magnitude argument can yield a result with little or no significance.

See Also

acos, **asin**, **atan**, **atan2**, **cos**, **tan**

Example

sin

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

main ()
{
    printf ("%f\n", sin (.5) );
}
```

produces the following:

```
0.479426
```

sinh

Computes the hyperbolic sine of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double sinh (
   double   x);
```

Parameters

x

(IN) Argument whose hyperbolic sine is to be computed.

Return Values

The **sinh** function returns the hyperbolic sine value. When the argument is outside the permissible range, *errno* is set and **matherr** is called. Unless the function is replaced, **matherr** prints a diagnostic message using the *stderr* stream.

Remarks

The **sinh** function computes the hyperbolic sine of *x*. A range error occurs if the magnitude of *x* is too large.

See Also

cosh, **matherr**, **tanh**

Example

sinh

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

main ()
```

```
{
   printf ("%f\n", sinh (.5) );
}
```

produces the following:

```
0.521095
```

sqrt

Computes the non-negative square root of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double sqrt (
   double   x);
```

Parameters

x

(IN) Specifies the argument for which the nonnegative square root is to be computed.

Return Values

The **sqrt** function returns the value of the square root. When the argument is outside the permissible range, *errno* is set and **matherr** is called. Unless the function is replaced, **matherr** prints a diagnostic message using the *stderr* stream.

Remarks

The **sqrt** function computes the nonnegative square root of *x*. A domain error occurs if the argument is negative.

See Also

exp, **log**, **matherr**, **pow**

Example

sqrt

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

```
main ()
{
   printf ("%f\n", sqrt (.5) );
}
```

produces the following:

```
0.707107
```

srand

Starts a new sequence of pseudo-random integers to be returned by subsequent calls to the **rand** function

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Platform: NLM

SMP Aware: Yes

Service: Mathematical Computation

Syntax

```
#include <stdlib.h>

void srand (
   unsigned int   seed);
```

Parameters

seed

(IN) Specifies the initial seed value passed in by the user.

Return Values

None

Remarks

A particular sequence of pseudo-random integers can be repeated by calling **srand** with the same *seed* parameter value.

See Also

rand, rand_r

tan

Computes the tangent of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double tan (
   double   x);
```

Parameters

x

(IN) Specifies the argument whose tangent is to be computed.

Return Values

The **tan** function returns the tangent value. When an error has occurred, *errno* is set.

Remarks

The **tan** function computes the tangent of *x* (measured in radians). A large magnitude argument can yield a result with little or no significance.

See Also

atan, **atan2**, **cos**, **sin**, **tanh**

Example

tan

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

main ()
{
   printf ("%f\n", tan (.5) );
```

```
}
```

produces the following:

```
0.546302
```

tanh

Computes the hyperbolic tangent of the specified argument
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Mathematical Computation

Syntax

```
#include <math.h>

double tanh (
   double   x);
```

Parameters

x

(IN) Specifies the argument whose hyperbolic tangent is to be computed.

Return Values

The **tanh** function returns the hyperbolic tangent value. When an error has occurred, *errno* is set.

Remarks

The **tanh** function computes the hyperbolic tangent of *x*.

When the *x* argument is large, partial or total loss of significance can occur. The **matherr** function is called in this case.

See Also

cosh, **sinh**, **matherr**

Example

tanh

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

main ()
```

```
main ()
{
   printf ("%f\n", tanh (.5) );
}
```

produces the following:

```
0.462117
```

y0, y1, yn

See **Bessel Functions**

Mathematical Computation: Structures

complex

Represents a complex number
Service: Math
Defined In: math.h

Structure

```
struct complex {
       double real;
       double imag;
   };
```

Fields

real

Contains the real part of the complex number.

imag

Contains the imaginary part or the complex number.

div_t

Contains the results of a division in type int
Service: Math
Defined In: stdlib.h

Structure

```
typedef struct {
   int   quot;
   int   rem;
} div_t;
```

Fields

quot

Contains the quotient of a division.

rem

Contains the remainder of a division.

Remarks

Used by **div**.

exception

Contains error information used by **matherr**
Service: Math
Defined In: math.h

Structure

```
struct exception
{
   int      type;
   char     *name;
   double   arg1;
   double   arg2;
   double   retval;
};
```

Fields

type

Contains the error type.

The *type* field contains one of the following values:

DOMAIN	A domain error has occurred, such as **sqrt**(-1e0).
SING	A singularity would result, such as **pow**(0e0,-2).
OVERFLOW	An overflow would result, such as **pow**(10e0,100).
UNDERFLOW	An underflow would result, such as **pow**(10e0,-100).
TLOSS	Total loss of significance would result, such as exp(1000).
PLOSS	Partial loss of significance would result, such as sin(10e70).

name

Points to a string containing the name of the function that detected the error.

arg1

Contains the first argument to the function.

arg2

Contains the second argument to the function.

retval

Contains the default return value. This is the only field that can be changed by a developer-supplied version of **matherr**.

ldiv_t

Contains the results of a division in type long
Service: Math
Defined In: stdlib.h

Structure

```
typedef struct {
   long   quot;
   long   rem;
} ldiv_t;
```

Fields

quot

Contains the quotient of a division.

rem

Contains the remainder of a division.

Remarks

Used by **ldiv**.

Memory Allocation

Memory Allocation: Functions

alloca

Allocates and clears memory space for a block of memory on the stack
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: No
Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

void *alloca (
      size_t   size);
```

Parameters

size

(IN) Specifies the size (in bytes) of the block of memory.

Return Values

alloca returns a pointer to the start of the allocated memory. The return value is NULL if insufficient memory is available or if the value of the *size* parameter is 0.

Remarks

Memory allocation is not limited to 64 KB. The *size* parameter is 32 bits.

The **alloca** function allocates memory space for an object of *size* bytes from the stack. The allocated space is automatically discarded when the current function exits.

The **alloca** function should not be used in an expression that is an argument to a function, because the **alloca** function manipulates the stack.

See Also

free, **malloc**, **NWGarbageCollect**

Example

alloca

```
#include stdio.h
#include string.h
#include nwmalloc.h
FILE   *open_err_file (char *name)
   {
      char *buffer;     /* Allocate temp buffer for file name */
      buffer = alloca (strlen (name) + 5);--
      if (buffer)
      {
         sprintf ("buffer, %s.err", name);
         return (fopen (buffer, "w") );
      }
      return (NULL);
   }
```

calloc

Allocates and clears memory space for an array of objects
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Memory Allocation

Syntax

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

void *calloc (
   size_t   n,
   size_t   size);
```

Parameters

n

(IN) Specifies the number of objects.

size

(IN) Specifies the size (in bytes) of each object.

Return Values

calloc returns a pointer to the start of the allocated memory. The return value is NULL if insufficient memory is available or if the value of the *size* parameter is 0.

Remarks

The **calloc** function initializes all the memory to binary zeroes.

Memory allocation is not limited to 64 KB. The *size* parameter is 32 bits.

The **calloc** function calls the **malloc** function.

A block of memory allocated using the **calloc** function should be freed using the **free** function.

See Also

free, **malloc**

Example

calloc

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

main ()
{
   int      *memoryPointer;
   size_t   n;
   memoryPointer = calloc (n, sizeof (int));
}
```

free

Frees a previously allocated memory block
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Memory Allocation

Syntax

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

void free (
   void   *ptr);
```

Parameters

ptr

(IN) Points to a memory block previously allocated by a call to**calloc**, **malloc**, or **realloc**.

Return Values

The **free** function returns no value.

Remarks

When the value of *ptr* is NULL, the **free** function does nothing; otherwise, the **free** function deallocates the memory block located by the *ptr* parameter. The *ptr* parameter is 32 bits.

After a call to **free**, the freed memory block is available for allocation.

See Also

calloc, **malloc**, **NWGarbageCollect**, **realloc**

Example

free

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

```
main ()
{
   char   *ptr;
   free (ptr);
}
```

malloc

Allocates a block of memory
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Memory Allocation

Syntax

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

void *malloc (
   size_t   size);
```

Parameters

size

(IN) Specifies the size (in bytes) of the memory block.

Return Values

malloc returns a pointer to the start of the newly allocated memory. The return value is NULL if insufficient memory is available or if the requested size is 0.

Remarks

Memory allocation is not limited to 64 KB. The *size* parameter is 32 bits.

The **malloc** function blocks only if the allocated block is greater than or equal to the size of cache_buffer+constant.

See Also

calloc, **free**, **realloc**

Example

malloc

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

```
main ()
{
   char      *memoryPointer;
   size_t    size;
   memoryPointer = malloc (size);
}
```

_msize

Returns the size of a memory block (implemented for NetWare® 3.11 and above)

Local Servers: blocking

Remote Servers: N/A

Classification: Other

SMP Aware: No

Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

size_t _msize (
   void   *buffer);
```

Return Values

The **_msize** function returns the size of the memory block pointed to by *buffer*.

Remarks

The **_msize** function returns the size of the memory block pointed to by *buffer* that was allocated by a call to **calloc**, **malloc**, or **realloc**.

See Also

calloc, **malloc**, **realloc**

Example

_msize

```
#include <nwmalloc.h>

main ()
{
   void *buffer;
   buffer = malloc ( 999 );
   printf ("Size of block is %u bytes\n", _msize (buffer) );
}
```

produces the following:

```
Size of block is 1000 bytes
```

The size of the memory is larger than the requested size due to space required for alignment.

NWGetAvailableMemory

Returns the number of bytes of memory available on the server
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: No
Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

LONG NWGetAvailableMemory (
   void);
```

Return Values

NWGetAvailableMemory returns the amount of memory (in bytes) available on the server.

Remarks

The amount of memory returned by **NWGetAvailableMemory** is the number of current cache buffers mutliplied by the size of a cache buffer. The amount available will usually not be allocatable as one large chunk as it will usually not be contiguous.

__qcalloc

Allocates memory space for an array of objects
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

void *__qcalloc (
   size_t   n,
   size_t   el_size);
```

Parameters

n

(IN) Specifies the number of objects.

el_size

(IN) Specifies the size (in bytes) of the object.

Return Values

_qcalloc returns a pointer to the start of the allocated memory.

Remarks

The **_qcalloc** function initializes all the memory to binary zeroes. The **_qcalloc** function calls the **malloc** function.

See Also

malloc, **realloc**

Example

__qcalloc

```
#include <nwmalloc.h>

int      memoryPtr;
size_t   n;
```

```
size_t   n;
size_t   el_size;
memoryPtr = _ _qcalloc (n, el_size);
```

__qmalloc

Allocates memory for an object
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11. 3.12, 4.x
SMP Aware: No
Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

void *__qmalloc (
   size_t   size);
```

Parameters

size

(IN) Specifies the size (in bytes) of the object.

Return Values

_qmalloc returns a pointer to the start of the newly allocated memory.

Remarks

The cache buffer size can be viewed or changed with the **SET** console command. The default cache buffer size is 4,096 bytes.

See Also

__qcalloc, **realloc**

Example

__qmalloc

```
#include <nwmalloc.h>

int      memoryPtr;
size-t   size;
memoryPtr = _ _qmalloc (size);
```

__qrealloc

Reallocates memory space for an object
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

void *__qrealloc (
   void     *old,
   size_t   size);
```

Parameters

old

(IN) Points to a previously allocated memory block.

size

(IN) Specifies the size (in bytes) of an object.

Return Values

__qrealloc returns a pointer to the start of the reallocated memory.

Remarks

The **__qrealloc** function calls the **__qmalloc** function to enlarge a block of memory.

See Also

calloc, **malloc**

Example

__qrealloc

```
#include <nwmalloc.h>

char     memoryPtr;
char     old;
```

```
char     old;
size_t   size;
memoryPtr = __qrealloc (old, size);
```

realloc

Reallocates a block of memory
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Memory Allocation

Syntax

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

void *realloc (
   void     *oldBlk,
   size_t   size);
```

Parameters

oldBlk

(IN) Points to a previously allocated memory block.

size

(IN) Specifies the size (in bytes) of the memory block.

Return Values

realloc returns a pointer to the start of the reallocated memory. The return value is NULL if there is insufficient memory available or if the requested size is 0.

Remarks

The **realloc** function calls the **malloc** function to enlarge a block of memory.

Memory allocation is not limited to 64 KB. The *size* parameter is 32 bits.

When the value of the *oldBlk* parameter is NULL, a new block of memory of *size* bytes is allocated. Otherwise, the **realloc** function reallocates space of an object of *size* bytes by either:

Shrinking the allocated size of the allocated memory block *oldBlk* when *size* is sufficiently smaller than the size of *oldBlk*.

Allocating a new block and copying the contents of *oldBlk* to the new block when *size* is not sufficiently smaller than the size of *oldBlk*.

Because it is possible that a new block can be allocated, no other pointers should point into the memory of *oldBlk*. When a new block is allocated, these pointers point to freed memory, with possibly disastrous results.

See Also

calloc, **free**, **malloc**

Example

realloc

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

main ()
{
   char      *memoryPointer;
   char      *oldBlk;
   size_t    size;
   memoryPointer = realloc (oldBlk, size);
}
```

stackavail

Returns the number of bytes currently available in the stack
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: Memory Allocation

Syntax

```
#include <nwmalloc.h>

size_t stackavail (void);
```

Return Values

The **stackavail** function returns the number of bytes currently available in the stack.

See Also

alloca, **calloc**, **malloc**

Memory Manipulation

Memory Manipulation: Functions

memchr

Locates the first occurrence of the specified character (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Memory Manipulation

Syntax

```
#include <string.h>

void *memchr (
   const void   *buf,
   int          ch,
   size_t       length);
```

Parameters

buf

(IN) Points to the object to be searched.

ch

(IN) Specifies the character to be located.

length

(IN) Specifies the number of bytes to search.

Return Values

memchr returns a pointer to the located character or NULL if the character does not occur in the object.

Remarks

The **memchr** function or macro locates the first occurrence of *ch* (converted to an unsigned char) in the first *length* characters of the object pointed to by *buf*.

See Also

memcmp, **memcpy**, **memset**

Example

memchr

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

main ()
{
   char   *where;
   char   buffer[80];
   where = memchr (buffer, 'x', 6);
   if (where == NULL)
   {
      printf ("'x' not found\n");
   }
}
```

memcmp

Compares (with case sensitivity) two blocks of memory (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Memory Manipulation

Syntax

```
#include <string.h>

int memcmp (
   const void   *s1,
   const void   *s2,
   size_t       length);
```

Parameters

s1

(IN) Points to the first object.

s2

(IN) Points to the second object.

length

(IN) Specifies the number of bytes to compare.

Return Values

memcmp returns an integer less than, equal to, or greater than zero, indicating that the object pointed to by *s1* is less than, equal to, or greater than the object pointed to by *s2*.

Remarks

The **memcmp** function or macro compares the first *length* characters of the object pointed to by *s1* to the object pointed to by *s2*.

See Also

memchr, **memcpy**, **memicmp**, **memset**

Example

memcmp

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

main ()
{
   char   buffer[80];
   if (memcmp (buffer, "Hello ", 6) < 0)
   {
      printf ("Less than\n");
   }
}
```

memcpy

Copies *length* characters from one buffer to another buffer (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
SMP Aware: Yes
Service: Memory Manipulation

Syntax

```
#include <string.h>

void *memcpy (
   void         *dst,
   const void   *src,
   size_t       length);
```

Parameters

dst

(IN) Points to a buffer into which to copy the characters.

src

(IN) Points to a buffer containing the characters to be copied.

length

(IN) Specifies the number of characters to copy.

Return Values

memcpy returns *dst*.

Remarks

The **memcpy** function or macro copies *length* characters from the buffer pointed to by *src* into the buffer pointed to by *dst*. Copying of overlapping objects have undefined results. See the **memmove** function to copy objects that overlap.

See Also

memchr, **memcmp**, **memmove**, **memset**

Example

memcpy

```
#include <string.h>

main ()
{
   char buffer[80];
   memcpy (buffer, "Hello", 5);
}
```

memicmp

Compares, with case insensitivity (uppercase and lowercase characters are equivalent), the first *length* characters of two objects

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

SMP Aware: Yes

Service: Memory Manipulation

Syntax

```
#include <string.h>

int memicmp (
   const void   *s1,
   const void   *s2,
   size_t       length);
```

Parameters

s1

(IN) Points to the first object.

s2

(IN) Points to the second object.

length

(IN) Specifies the number of characters to compare.

Return Values

The **memicmp** function returns an integer less than, equal to, or greater than zero, indicating that the object pointed to by *s1* is less than, equal to, or greater than the object pointed to by *s2*.

Remarks

The **memicmp** function compares, with case insensitivity (uppercase and lowercase characters are equivalent), the first *length* characters of the object pointed to by *s1* to the object pointed to by *s2*.

See Also

memchr, **memcmp**, **memcpy**, **memset**

Example

memicmp

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

main ()
{
   char   buffer[80];
   if (memicmp (buffer, "Hello", 5) < 0)
   {
      printf ("Less than\n");
   }
}
```

memmove

Copies *length* characters from one buffer to another buffer
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
SMP Aware: Yes
Service: Memory Manipulation

Syntax

```
#include <string.h>

void *memmove (
   void         *dst,
   const void   *src,
   size_t       length);
```

Parameters

dst

(IN) Points to a buffer into which to copy the characters.

src

(IN) Points to a buffer containing the characters to be copied.

length

(IN) Specifies the number of characters to move.

Return Values

The **memmove** function returns *dst*.

Remarks

The **memmove** function copies *length* characters from the buffer pointed to by *src* to the buffer pointed to by *dst*. The *dst* buffer is an exact copy of the *src* buffer. Copying of overlapping objects guarantees a buffer fill. See **memcpy** to copy objects that do not overlap.

See Also

memcpy, **memset**

Example

memmove

```
#include <string.h>

main ()
{
   char buffer[80];
   memmove (buffer+1, buffer, 79);
   buffer[0] = '*';
}
```

memset

Fills the first *length* characters of an object with a specified value (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
SMP Aware: Yes
Service: Memory Manipulation

Syntax

```
#include <string.h>

void *memset (
   void     *s,
   int      c,
   size_t   length);
```

Parameters

s

(IN) Points to the object.

c

(IN) Specifies the value of the character.

length

(IN) Specifies the number of characters to set.

Return Values

memset returns *s*.

Remarks

The **memset** function or macro fills the first *length* characters of the object pointed to by *s* with the value *c*.

See Also

memchr, **memcmp**, **memcpy**, **memmove**

Example

memset

```
#include <string.h>

main ()
{
   char buffer[80];
   memset (buffer, '=', 80);
}
```

NLM Debug

NLM Debug: Functions

assert

Identifies program logic errors
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
Platform: NLM
SMP Aware: Yes
Service: NLM 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
```

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.

BumpFunctionCount

Increments a counter for developer-coded functions for use with NLMDebug.NLM

Local Servers: blocking

Remote Servers: N/A

Classification: NetWare 4.11

SMP Aware: Yes

Service: NLM Debug

Syntax

```
#include <nwdebug.h>

int BumpFunctionCount (
   const char *name);
```

Parameters

name

(IN) Specifies the name of the function or another object to count (maximum length of any registered name is 100 characters).

Return Values

None

Remarks

BumpFunctionCount registers and/or increments a counter to record th enumber of times the object specified in the *name* parameter has been accessed.

BumpFunctionCount is for use by the developer and implements simple profiling as do the server libraries. **BumpFunctionCount** is primarily intended to be called at the beginning of each function.

The source code (including nwdebug.h) must be compiled with the preprocessor symbol **Debug** defined for **BumpFunctionCount** to have any impact.

name is a character string that designates any object that needs to be counted. The count of all objects is displayed by NLMDebug.NLM.

See Also

NWEnableDebugProfile, **NWValidateDebugProfile**

EnterDebugger

Enters system debugger
Local Servers: N/A
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

int EnterDebugger (void);
```

Return Values

None

NWClearBreakpoint

Dynamically clears the breakpoint set with **NWSetBreakpoint**
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

void NWClearBreakpoint (
   int   breakpoint);
```

Parameters

breakpoint
(IN) Specifies the breakpoint to clear.

Return Values

None

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

NWDebugPrintf

Implements **printf**-style output in the NetWare System Debugger
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

int NWDebugPrintf (
   const char *format,
   ...);
```

Parameters

format

(IN) Specifies the format of the output.

suceeding parameters

(IN) See the documentation for **printf**.

Return Values

NWDebugPrintf returns the number of characters written, or it returns a negative value if an output error occurred.

errno is not set because no CLib context is available; instead, the absolute value of the returned value is equivalent to *errno*.

Remarks

NWDebugPrintf is intended for use from a registered alternate debugger.

NWDebugPrintf writes output to the NetWare System Debugger screen under control of the parameter *format*. The format string operates the same as for **printf** except only the following pronouns are supported (while some new ones have been added which are not suggested for the ANSI function).

Format Control Strings The valid conversion type specifiers are:

c	An argument of type int is converted to a value of type char and the corresponding ASCII character code is written to the output stream.
d, i	An argument of type int is converted to a signed decimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
o	An argument of type int is converted to an unsigned octal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
s	Characters from the string specified by an argument of type char *, up to, but not including, the terminating NULL character (\0), are written to the output stream. If a precision is specified, no more than that many characters are written.
S	Characters from a length-preceded string are written to the output stream. If a precision is specified, no more than that many characters are written.
u	An argument of type int is converted to an unsigned decimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
x, X	An argument of type int is converted to an unsigned hexadecimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added. Hexadecimal notation uses digits (0 through 9) and characters (a through f or A through F) for **x** or **X** conversions respectively, as the hexadecimal digits. Subject to the alternate-form control flag, **0x** or **0X** is affixed to the output.
b	An argument of type int is converted to binary representation and written to the debugger screen. Length is specified using an integer corresponding to the number of 1s and 0s desired. If the length is prefixed with a 0, leading zeros are retained. See NWDisplayBinaryAtAddr. Left justifies the value being printed. For example: NWDebugPrintf("%4b", 4) yields `100' NWDebugPrintf("%04b", 4) yields `0100' NWDebugPrintf("%4b", 4 + 0xf0) yields `11110100'
t,T	An argument of any type is converted to a Boolean (truth value) representation and displayed as `TRUE' or `FALSE'. (No localization of these values is supported.) The maximum possible width of t or T is 90.
p, P	Identical to x, X. Do not use far pointers.

Any other conversion type specifier character, including another percent (%) character, is written to the output stream with no special interpretation.

The arguments must correspond with the conversion type specifiers, left to right in the string; otherwise, indeterminate results occur.

WARNING: NWDebugPrintf is provided `as is' and is not expressly supported. Bug reports will be gladly accepted and will be fixed on a low-priority basis. Extending the NetWare System Debugger is hazardous; therefore, you must take full responsibility for what your code does or does not do at this level. Novell is trying to open up to you the capability to extend this debugger.

See Also

NWDisplayBinaryAtAddr, NWDisplayDoubleAtAddr, NWDisplayLConvAtAddr, NWDisplayStringAtAddr, NWDisplayTMAtAddr, NWDisplayUnicodeAtAddr

NWDisplayBinaryAtAddr

Displays an address to a Unicode string in the NetWare System Debugger
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

void NWDisplayBinaryAtAddr (
   void   *addr);
```

Parameters

addr

(IN) Specifies the address of string to be displayed.

Return Values

None

Return Values

NWDisplayBinaryAtAddr displays the long word starting at address *addr* in binary. This is the function called when the command

```
binary <address>
```

is issued to the NetWare System Debugger once Threads.NLM is loaded.

NWDisplayBinaryAtAddr is intended to be called from a registered alternate debugger.

NWDisplayDoubleAtAddr

Displays an address to specified double in the NetWare System Debugger
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

void NWDisplayDoubleAtAddr (
   void   *addr);
```

Parameters

addr

(IN) Specifies the address of double to be displayed.

Return Values

None

Remarks

NWDisplayDoubleAtAddr displays a double in binary. The sign occupies the first bit, the exponent occupies the next 11 bits, while the mantissa occupies the remaining 52 bits.

NWDisplayDoubleAtAddr is the function called when the command

```
double <address>
```

is called from the NetWare System Debugger once Threads.NLM is loaded.

NWDisplayDoubleAtAddr is intended to be called from a registered alternate debugger.

NWDisplayLConvAtAddr

Displays an ANSI locale structure in the NetWare System Debugger
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>
#include <locale.h>

void NWDisplayLConvAtAddr (
   void  *lc);
```

Parameters

lc

(IN) Specifies the address of the structure to be displayed.

Return Values

None

Remarks

NWDisplayLConvAtAddr displays the ANSI locale structure (struct lconv) starting at the address specified by *lc*.

NWDisplayLConvAtAddr is the function called when the command

```
lconv <address>
```

is called from the NetWare System Debugger once CLib.NLM is loaded.

NWDisplayLConvAtAddr is intended to be called from a registered alternate debugger.

NWDisplayStringAtAddr

Displays an address to an ASCIIZ string in the NetWare System Debugger
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

void NWDisplayStringAtAddr (
   void    *s,
   size_t   len);
```

Parameters

s

(IN) Specifies the address of the string to be displayed.

len

(IN) Specifies the length of string in bytes.

Return Values

None

Remarks

NWDisplayStringAtAddr displays characters starting at address *s* and extending for *len* characters or until a null character is encountered (whichever is less).

NWDisplayStringAtAddr is the function called when the command

```
string <address> [length]
```

is called from the NetWare System Debugger once Threads.NLM is loaded.

NWDisplayStringAtAddr is intended to be called from a registered alternate debugger.

NWDisplayTMAtAddr

Displays an ANSI time break-down structure in the NetWare System Debugger

Local Servers: blocking

Remote Servers: N/A

Classification: NetWare 4.11

SMP Aware: No

Service: NLM Debug

Syntax

```
#include <nwdebug.h>
#include <time.h>

void NWDisplayTMAtAddr (
   void  *tm);
```

Parameters

tm

(IN) Specifies the address of the structure to be displayed.

Return Values

None

Remarks

NWDisplayTMAtAddr displays the ANSI time break-down structure (struct tm) starting at the address specified by *tm*.

NWDisplayTMAtAddr is the function called when the command

```
tm <address>
```

is called from the NetWare System Debugger once CLib.NLM is loaded.

NWDisplayTMAtAddr is intended to be called from a registered alternate debugger.

NWDisplayUnicodeAtAddr

Displays an address to a Unicode string in the NetWare System Debugger
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

void NWDisplayUnicodeAtAddr (
   void *s,
   size_t len);
```

Parameters

s

(IN) Specifies the address of the string to be displayed.

len

(IN) Specifies the length of the string in Unicode characters.

Return Values

None

Remarks

NWDisplayUnicodeAtAddr displays Unicode characters starting at the address specified by *s* and extending for *len* characters or until a null word <0000> is encountered (whichever is less) as ASCII characters intermixed with numeric representation in the form <nn>.

NWDisplayUnicodeAtAddr is the function called when the command

```
unicode <address> [length]
```

is called from the NetWare System Debugger once Threads.NLM is loaded.

NWDisplayUnicodeAtAddr is intended to be called from a registered alternate debugger.

NWEnableDebugProfile

Toggles between enabling or disabling the profiling of developer-coded function calls

Local Servers: blocking

Remote Servers: N/A

Classification: NetWare 4.11

SMP Aware: No

Service: NLM Debug

Syntax

```
#include <nwdebug.h>

void NWEnableDebugProfile (
   int flag);
```

Parameters

flag

(IN) Specifies either TRUE or FALSE indicating whether calls to developer-coded functions should be counted.

Return Values

None

Remarks

NWEnableDebugProfile enables or disables function call profiling for server-library functions to be counted and displayed by the Function-Call Profiler option in NLMDebug.NLM. Only functions or other objects counted by calling **BumpFunctionCount** are tabulated.

See Also

NWValidateDebugProfile

NWSetBreakpoint

Sets a breakpoint programmatically
Local Servers: nonblocking
Local Servers: blocking
Classification: 3.x, 4.x
SMP Aware: No
Service: NLM 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_BREAK POINT	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

NWValidateDebugProfile

Determines whether profiling is active for developer code
Local Servers: blocking
Remote Servers: N/A
Classification: NetWare 4.11
SMP Aware: No
Service: NLM Debug

Syntax

```
#include <nwdebug.h>

int NWValidateDebugProfile (void);
```

Return Values

NWValidateDebugProfile returns non-zero whenever NLMDebug.NLM has been used to turn on function-call profiling in the NLM to be debugged (as named in NLMDebug.NLM, option: Function-Call Profiling).

Remarks

NWValidateDebugProfile is not typically called directly. Instead, it is usually called by **BumpfunctionCount**.

See Also

BumpFunctionCount, **NWEnableDebugProfile**

perror

Prints an error message
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
Platform: NLM
SMP Aware: Yes
Service: NLM Debug

Syntax

```
#include <stdio.h>

void perror (
   const char   *prefix);
```

Parameters

prefix

(IN) Specifies 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

NWSNUT

NWSNUT: Guides

NWSNUT: Concept Guide

NWSNUT provides the developer with the tools needed to create a user interface similar to that of many NetWare® utilities, such as PCONSOLE. This interface allows the user to manipulate data by using menus, lists and forms.

General Information

- The NWSNUT Environment
- Zones
- Other NWSNUT Functions
- NWSNUT Example
- NWSNUT Function List

Portals

- Portals
- Basic Steps for Using Portals
- Special Purpose Portals

Lists

- Lists
- Basic Steps for Using Lists
- Handling Lists
- Manipulating List Elements
- Marking Lists
- Routines for List Elements
- Saving Lists
- Sorting Lists
- Specialized Lists

Menus

Menus

Basic Steps for Using Menus

Forms

Forms

Basic Steps for Using Forms

Text in NWSNUT

Text in NWSNUT

NWSNUT Messages

Help Screens

Additional Links

NWSNUT: Functions

NWSNUT: Structures

NWSNUT: Task Guide

Basic Steps for Using Portals

Basic Steps for Using Lists

Basic Steps for Using Menus

Basic Steps for Using Forms

Additional Links

NWSNUT: Functions

NWSNUT: Structures

NWSNUT: Tasks

Basic Steps for Using Forms

The four steps below summarize the **basic** steps for using forms. The examples are taken from ndemo.c, which can be found in the EXAMPLES directory.

1. **Initialize the form.**

   ```
   NWSInitForm (handle);
   ```

   NWSInitForm initializes the pointers for the current form.

2. **Build the form by adding fields to it.**

   ```
   line = 0;
   NWSAppendCommentField (line, 1, "Boolean Field:", handle);
   NWSAppendBoolField (line, 25, NORMAL_FIELD, &myBoolean,
      NULL, handle);
   line += 2;
   NWSAppendCommentField (line, 1, "Integer Field:", handle);
   NWSAppendIntegerField (line, 25, NORMAL_FIELD, &myInteger,
      0, 9999, NULL, handle);
   ```

   To build the form, you must call one of the "append" functions for each of the fields that you want to add to the form. Specify the placement of each field within the form by specifying the portal line and column. Each field type limits the kind of input that the user can enter for the field and the actions that can be associated with it. You can assign a help context to most fields. There are 12 types of fields that can be used in a form as shown in Form Field Types.

   For more information, see the following:

   Form Field Types

   Field Structure

   Prompt Fields

   Menu Fields

   Custom Fields

   Form Editing Functions

3. **Display the form and allow the user to edit it.**

```
NWSEditPortalForm (FORM_HEADER, 11, 40, 16, 50, F_NOVERIFY,
   FORM_HELP, EXIT_FORM_MSG, handle);
```

Once the form is built, a **form editing** function is called to display the form and allow the user to access, modify and enter data into it. NWSNUT provides functions for editing a form (see Form Editing Functions).

4. **Destroy the form.**

```
NWSDestroyForm (handle);
```

NWSDestroyForm frees all of the fields in the current form and reinitializes form pointers.

Basic Steps for Using Lists

The four steps below summarize the **basic** steps for using lists. The examples are taken from ndemo.c, which can be found in the EXAMPLES directory.

1. **Initialize the list.**

```
NWSInitList (handle, Free);
```

This function initializes a LISTPTR structure.

> **NOTE:** If a list has previously been created, it must be saved or destroyed before creating a new list.

2. **Add items to the list.**

```
NWSAppendToList (NWSGetMessage (LIST_ITEM_1,
                &handle->messages),
                (void *) 0, handle);
NWSAppendToList (NWSGetMessage (LIST_ITEM_2,
                &handle->messages),
                (void *) 0, handle);
NWSAppendToList (NWSGetMessage (LIST_ITEM_3,
                &handle->messages),
                (void *) 0, handle);
NWSAppendToList (NWSGetMessage (LIST_ITEM_4,
                &handle->messages),
                (void *) 0, handle);
```

NWSInsertInList or **NWSInsertInPortalList** can also be used to add items to a list.

3. **Display the list and allow the user to access it.**

```
NWSList (LIST_HEADER, 10, 40, 4,
        strlen (NWSGetMessage (LIST_HEADER,
        &handle->messages)) + 4,
        M_ESCAPE | M_SELECT, NULL, handle,
        NULL, ListAction, 0);
```

NWSList displays the list and allows the user to access it. NWSNUT has control of the process while the user makes a selection. Control is turned over to the ListAction routine when the user selects a list item.

When a list element is created, custom data may be attached to it by using the *otherInfo* parameter. This data is put in the *otherInfo* field of the LIST structure for the new list element. This custom data can be any type, including another list. If you attach a list to another list or to a form, you must first call **NWSInitListPtr** to initialize a LISTPTR structure for it.

4. **Destroy the list.**

```
NWSDestroyList (handle);
```

This function frees all of the list nodes and initializes the list pointers.

Parent Topic: Lists

Basic Steps for Using Menus

The four steps below summarize the **basic** steps for using menus. The examples are taken from ndemo.c, which can be found in the EXAMPLES directory.

1. **Initialize the menu.**

```
NWSInitMenu (handle);
```

NWSInitMenu initializes the pointers for the new menu.

2. **Add options to the menu.**

```
NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
   "Menu Item 1       ", &handle->messages);
NWSSetDynamicMessage(DYNAMIC_MESSAGE_TWO,
   "Menu Item 2       ", &handle->messages);
NWSSetDynamicMessage(DYNAMIC_MESSAGE_THREE,
   "Menu Item 3       ", &handle->messages);
NWSAppendToMenu (DYNAMIC_MESSAGE_ONE, 1, handle);
NWSAppendToMenu (DYNAMIC_MESSAGE_TWO, 2, handle);
NWSAppendToMenu (DYNAMIC_MESSAGE_THREE, 3, handle);
```

NWSAppendToMenu adds an option to the menu. You assign an option value to each menu item when it is added to the menu. When the user selects this option, its value is passed to the action routine specified

when **NWSMenu** is called.

3. **Display the menu and allow the user to choose from its options.**

```
NWSMenu (MENU_HEADER, 10, 40, NULL, MenuAction, handle,
   (void *)handle);
```

When you call **NWSMenu**, you specify an action routine for the menu. The action routine receives the value of the option that the user chooses.

4. **Destroy the menu.**

```
NWSDestroyMenu (handle);
```

NWSDestroyMenu frees the nodes and menu pointers for the menu.

Parent Topic: Menus

Basic Steps for Using Portals

The four steps below summarize the **basic** steps for using portals. The examples are taken from ndemo.c, which can be found in the EXAMPLES directory.

1. **Create a portal.**

```
portalNumber = NWSCreatePortal(portalTop, portalLeft,
   portalFrameHeight, portalFrameWidth, portalVirtualHeight,
   portalVirtualWidth, SAVE, "Demonstration Portal", 0,
   DOUBLE, 0, CURSOR_ON, VIRTUAL, handle);
```

The following properties can be specified for the portal:

Frame size

Virtual display area size

Border type and video attribute

Whether to write directly to the physical screen or to the virtual display area

Header text and attribute

Cursor state

Whether to save screen data beneath the portal

Notice that the *frameHeight* and *frameWidth* parameters include the width of the portal border and header (see the NWSNUT Screen Displaying a Portal figure). Therefore, the portal's physical display area is smaller than its frame area. The terms **portal line** and **portal column**

refer to a line and column inside this portal display area.

See the following topics for more information:

Video Attribute

Positioning the Portal

Screen Palette

2. **Convert the portal number to a PCB.**

```
NWSGetPCB (&pPtr, portalNumber, handle);
```

This function returns the PCB for the portal, which is required for all functions that write to portals.

3. **Write data to the portal.**

```
/******** clear the portal, and bring it to the front ******/
NWSClearPortal (pPtr);
NWSSelectPortal (portalNumber, handle);
ptr = "This is a portal";
NWSShowPortalLine (0, 0, ptr, strlen (ptr),   pPtr);
ptr = "It may contain any type of data";
NWSShowPortalLine (2, 0, ptr, strlen (ptr),   pPtr);
NWSUpdatePortal (pPtr);  /**** cause it to be displayed on the sc
```

The example shows just one way to write to a portal. See Basic Portal Activities for a summary of functions used for writing to portals.

For more information, see the following:

Selecting Portals

The Portal Cursor

Scrolling the Portal

Line Drawing Characters

4. **Destroy the portal.**

```
NWSDestroyPortal (portalNumber, handle);
```

NWSDestroyPortal destroys the portal and cleans up all resources used by the portal.

Parent Topic: Portals

NWSNUT: Concepts

Alerts and Errors

NWSNUT has two displays for informing the user of errors---alerts and error portals. Alerts are created and displayed by calling **NWSAlert** or **NWSAlertWithHelp**. Error portals are created with **NWSDisplayErrorText** and **NWSDisplayErrorCondition**.

NWSAlert displays the alert message in a portal appropriately sized for the message that you pass it. It displays until the user presses **Esc** or **Enter**.

NWSDisplayErrorText and **NWSDisplayErrorCondition** display the error text, along with a message indicating the severity of the error, and an optional message identifier for the error message. The two functions are similar, but **NWSDisplayErrorCondition** allows you to display the name of the procedure that resulted in the error and to create an error list for the procedure. These two functions also optionally display the name and version number of the NLM with the error number. If you do not want this information displayed, call **NWSSetErrorLabelDisplayFlag**.

Parent Topic: NWSNUT Messages

Basic Portal Activities

NWSShowPortalLine

Displays a line of text. Portal line and column can be specified.

NWSShowPortalLineAttribute

Changes the video attribute of a specified portion of a portal line.

NWSDisplayTextInPortal

Displays text in an existing portal. Starting portal line for text and indent level can be specified.

NWSDisplayTextJustifiedInPortal

Same as above, but text width and video attribute can be specified.

NWSFillPortalZone

Fills a specified region of the portal with a character.

NWSFillPortalZoneAttribute

Fills a specified region of the portal with a video attribute.

NWSClearPortal

Blanks out a portal.

NWSUpdatePortal

Redraws a virtual portal to show changes since creation or last update.

NWSDrawPortalBorder

Redraws the border of a portal.

Parent Topic: Basic Steps for Using Portals

Confirmation of a Decision

NWSNUT provides a function that creates a special-purpose portal to allow a user to confirm a decision. **NWSConfirm** displays a portal that contains a menu with only "yes" and "no" options.

Parent Topic: User Input

Creating Messages

Messages can be created by using the Message Librarian (MSGLIB.EXE). This allows messages to be segregated from the code for easier translation. Each message is given a message name, also known as a **message identifier**. For example, you might assign the message identifier WAIT_MSG to the message "Please wait".

Many NWSNUT functions request the message identifier for the message, but some request a pointer to BYTE. If this is the case, the message can be retrieved from the message table by calling **NWSGetMessage**. For example:

```
NWSAppendToList (NWSGetMessage (LIST_ITEM_1,
   &handle->messages), (void *) 0, handle);
```

After the messages are created, they must be imported for NWSNUT using the NetWare Message Internationalization Tools.

Parent Topic: NWSNUT Messages

Custom Fields

NWSAppendToForm allows the developer to create a specialized field if necessary. This function allows you to:

Fill *fieldXtra* and *fieldData* of the FIELD structure

Specify routines for formatting, key input, input verification, and memory release for the *fieldData* and *fieldXtra* fields

Specify action keys for the field

Assign a video attribute to the field

Custom routines for a field can also be set by calling **NWSSetFieldFunctionPtr**, and retrieved by calling **NWSGetFieldFunctionPtr**.

Parent Topic: Forms

Direct Portals

Direct portals do not have a "staging area" as do virtual portals. Data written to the portal is written directly to the screen. Therefore, the display area of a direct portal is the portal frame size minus the header and borders. The size of a direct portal is limited by the size of the screen.

Parent Topic: Portals

Dynamic Messages

In addition to the message table specified when **NWSInitializeNut** is called, you can specify messages by using the dynamic messages held in the MessageInfo structure. Call **NWSSetDynamicMessage** to enter a message into this structure. If you are hard-coding your messages into the NLM, you can use dynamic messages for those functions that require a message identifier. For example:

```
NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
   "Menu Item 1       ", &handle->messages);
NWSAppendToMenu (DYNAMIC_MESSAGE_ONE, 1, handle);
```

Parent Topic: NWSNUT Messages

Editing a String

NWSNUT provides two functions that allow the user to edit a string---**NWSEditString** and **NWSEditText**. Both functions create a portal in which the user can edit a string, but **NWSEditString** allows you to specify an insertion routine and an action routine.

Parent Topic: User Input

Field Structure

Each "append" function creates a structure of type FIELD, defined in nwsnut.h. The FIELD structure is defined in as follows:

```
typedef struct fielddef {
   LIST              *element;
   LONG              fieldFlags;
   LONG              fieldLine;
   LONG              fieldColumn;
   LONG              fieldWidth;
   LONG              fieldAttribute;
   int               fieldActivateKeys;
   void              (*fieldFormat)(struct fielddef *field,
                        BYTE *text, LONG buffLen);
   LONG              (*fieldControl)(struct fielddef *field,
                        int selectKey, int *fieldChanged,
                        NUTInfo *handle);
   int               (*fieldVerify)(struct fielddef *field,
                        BYTE *data, NUTInfo *handle);
   void              (*fieldRelease)(struct fielddef *field);
   BYTE              *fieldData;
   BYTE              *fieldXtra;
   int               fieldHelp;
   struct fielddef   *fieldAbove;
   struct fielddef   *fieldBelow;
   struct fielddef   *fieldLeft;
   struct fielddef   *fieldRight;
   struct fielddef   *fieldPrev;
   struct fielddef   *fieldNext;
   void              (*fieldEntry)(struct fielddef *field,
                        void *fieldData, NUTInfo *handle);
   void              *customData;
   void              (*customDataRelease)(
                        void *fieldCustomData,
                        NUTInfo *handle);
   NUTInfo           *nutInfo;
} FIELD
```

NOTE: Fields in this structure should not be changed directly. Use NWSNUT functions for building and manipulating forms.

Most of these fields can be changed by using various form functions. Exceptions are *nutInfo* and fields having to do with a field's relationship to other fields in the form (*fieldAbove, fieldBelow,* and so on).

The *element* field points to a list structure.

The *fieldFlags* field is set with most "append" functions and can have the

values summarized in the following table.

Table auto. Field Flags

Value	Meaning
NORMAL_FIELD	Normal, editable field.
LOCKED_FIELD	Inaccessible field.
SECURE_FIELD	Noneditable field.
REQURED_FIELD	Verify field on form exit.
HIDDEN_FIELD	Hidden fields are not seen by the user. These fields are also locked.
PROMPT_FIELD	Prompt fields cannot be selected by the user. These fields are also locked.
ULOCKED_FIELD	Field locked by user.
FORM_DESELECT	Causes form deselection before action and verify routines are called.
NO_FORM_DESELECT	Form is not deselected before action and verify routines are called.
DEFAULT_FORMAT	Normal field-controlled justification.
RIGHT_FORMAT	Right justification.
LEFT_FORMAT	Left justification.
CENTER_FORMAT	Centered.

Prompt and comment fields are automatically locked and inaccessible to the user.

The *fieldLine* and *fieldColumn* fields position the form field. The *fieldLine* field contains the portal line on which the form field is located. The *fieldColumn* field contains the portal column on which the left-most character of the field is located. The *fieldWidth* field contains the maximum width of the form field. Each "append" function allows you to set the *fieldLine* and *fieldColumn*.

The *fieldAttribute* field contains the video attribute for the form field.

The *fieldActivateKeys* field contains the keys that activate the form field.

The following fields contain information about what routines are used for the form field:

The *fieldFormat* field points to the routine used to format the form field.

The *fieldControl* field points to the routine that is called when the form field is selected.

The *fieldVerify* field points to the routine used to verify input to the form field.

The *fieldRelease* field points to the routine called to release memory allocated for *fieldData* and *fieldXtra*.

The *customDataRelease* field points to a routine to release data in *customData*. The parameters match **NWSFree** so that **NWSAlloc** can be used to allocate memory for *customData*, further guaranteeing that memory is freed.

The *fieldEntry* field points to a routine called when any field in the form is entered.

The *fieldData* field points to data associated with the form field. The*fieldXtra* field points to additional control information associated with the form field.

The *fieldHelp* field contains the help context for the form field. Each of the "append" functions allows you to set the help context.

The *customData* field contains user-defined data to be attached to the form field.

Parent Topic: Forms

Form Editing Functions

NWSEditForm

Allows the user to edit the form.

NWSEditPortalForm

Same as **NWSEditForm**, but allows you to specify help context for the form.

NWSEditPortalFormField

Same as **NWSEditForm**, but allows you to specify help context and first field to highlight.

NWSSetFormRepaintFlag

Repaints the form to refect changes made to the form.

Parent Topic: Forms

Form Field Types

Field	Function	Use
Boolean	**NWSAppendBoolField**	Creates a special menu list where the choices are yes and

		no.
Comment	**NWSAppendCommentField**	Creates a field with a prompt. The user cannot edit this field.
Custom	**NWSAppendToForm**	Creates a field that is developer-defined.
Hexadecimal	**NWSAppendHexField**	Creates a field that accepts only hexadecimal input from the user.
Hot Spot	**NWSAppendHotSpotField**	Creates a field that calls a "spot action" routine when selected.
Integer	**NWSAppendIntegerField**	Creates a field that accepts only an integer as input from the user.
Menu	**NWSAppendMenuField**	Creates a field that displays a menu when selected.
Password	**NWSAppendPasswordField**	Creates an edit field for password input.
Prompt	**NWSAppendPromptField**	Creates a field with a prompt. The user cannot edit this field.
Scrollable String	**NWSAppendScrollableString Field**	Creates a scrollable field containing an editable string.
String	**NWSAppendStringField**	Creates a field containing an editable string.
Unsigned Integer	**NWSAppendUnsignedInteger Field**	Creates a field that accepts only an unsigned integer as input from the user.

Parent Topic: Forms

Forms

A form is another specific type of list. Forms are designed to allow the user to input various types of data.

Related Topics

Basic Steps for Using Forms

Form Field Types

Field Structure

Prompt Fields

Menu Fields

Custom Fields

Form Editing Functions

Function Keys

NWSNUT allows you to enable or disable function keys, either one at a time or in groups. A set of function keys can be saved so that it can be used again. See the following table for a list of NWSNUT's Function Key functions.

NWSDisableAllFunctionKeys
Disables all function keys.

NWSDisableFunctionKey
Disables a single function key.

NWSEnableAllFunctionKeys
Enables all function keys.

NWSEnableFunctionKey
Enables a single function key.

NWSEnableFunctionKeyList
Enables a list of function keys.

NWSSaveFunctionKeyList
Saves a list of function keys.

Parent Topic: User Input

Handling Lists

NWSSetList makes a list current. To obtain the pointers for a list, call

NWSGetList. To obtain a pointer to the head or tail of a list, call **NWSGetListHead** or **NWSGetListTail**, respectively.

Related Topic: Saving Lists

Parent Topic: Lists

Help Screens

Help screens are created with helplib.exe. The help file for the various languages is located in the same directory as the message file for that language. In addition, a help file can be linked to the NLM with NLMLINK. As with the message files, you can pass a pointer to the help file (after it is loaded into memory) when NWSNUT is initialized with **NWSInitializeNut**, or you can pass a NULL for the *helpScreens* parameter and let NWSNUT retrieve the appropriate help file for you.

The helpOffset required with any of the help functions (for example, **NWSPushHelpContext**), is the offset or **help identifier** assigned to the help screen in the nlmname.hlh file produced by helplib.exe.

To use help, call **NWSPushHelpContext**. This enables the specified help screen until another help screen is pushed, or until the previous help screen is popped. When the user presses **F1** NWSNUT displays the current help screen (the last one pushed). After the help is no longer needed, pop it from the help stack by calling **NWSPopHelpContext**. For example:

```
#include "myNLM.HLH" // includes definition for MY_FIRST_HELP

   NWSPushHelpContext (MY_FIRST_HELP, handle);

   /*
      from this point on, whenever the user presses F1, the
      help screen identified by MY_FIRST_HELP will be displayed
   */

   NWSPopHelpContext (handle);

   // the previous help is now in force.
```

A specific help screen can be displayed without pushing it onto the help stack by calling **NWSDisplayHelpScreen**.

Pre-Help Portals

In addition to help, NWSNUT allows the creation of "pre-help" portals. A **pre-help portal** is a portal with a message that remains on the screen for a long period of time, such as "Press <F1> for help." To display a pre-help

message, call **NWSDisplayPreHelp**. To remove a pre-help portal, call **NWSRemovePreHelp**.

Parent Topic: Text in NWSNUT

Interrupt Keys

An **interrupt key** is a key that is assigned a procedure, so that the procedure is called when the interrupt key is pressed. NWSNUT provides four functions dealing with interrupt keys, as listed in the following table.

NWSDisableAllInterruptKeys
Disables all interrupt keys.

NWSEnableInterruptKey
Enables in interrupt key and associates that key with a routine.

NWSEnableInterruptList
Enables a list of interrupt keys.

NWSSaveInterruptList
Saves a list of interrupt keys.

Parent Topic: User Input

Keyboard Input in NWSNUT

The following table lists functions that deal with keyboard input.

NWSGetKey
Reads one key from the keyboard buffer.

NWSKeyStatus
Indicates whether a key is waiting in the keyboard buffer.

NWSUngetKey
Inserts a key into the keyboard buffer.

NWSWaitForEscape
Waits for the **Esc** key to be pressed.

NWSWaitForEscapeOrCancel
Waits for either the **Esc** or cancel (**F7**) key to be pressed.

NWSWaitForKeyAndValue
Waits for one of the keys in a specified set to be pressed.

Parent Topic: User Input

Line Drawing Characters

To obtain line drawing characters for drawing in a portal, call **NWSGetLineDrawCharacter** (see "character and key constants" in NWSNUT.H).

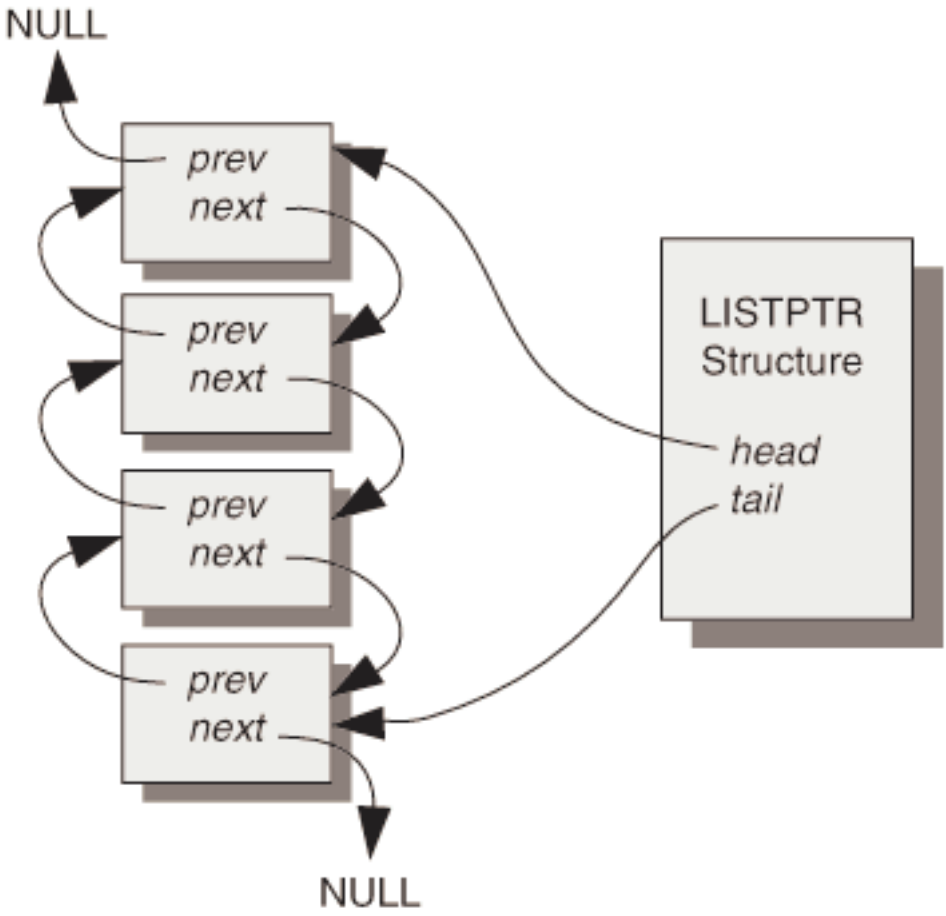
Parent Topic: Basic Steps for Using Portals

Lists

A **list** is a data structure that NWSNUT uses to allow the user to select from a number of items or options. A list is displayed in an updated virtual portal. The user can scroll the portal and select one or more items. The format, presentation, and content of a list are controlled by the calling NLM. This is done by using either default NWSNUT routines or NWSNUT client-specified routines to format, present, and verify the list contents.

A list is made up of a LISTPTR structure and a number of LIST structures that are linked together. The LISTPTR structure contains pointers to the first and last elements of the list (the *head* and *tail* fields of LISTPTR). Each list element in the list is defined by a LIST structure which contains pointers to both the previous and following elements in the list. The first list element points to NULL as its previous element (LIST.prev), and the last list element points to NULL as its following list element (LIST.next). See the following figure for a representation of the list elements and their relationship to the LISTPTR structure. The LIST and LISTPTR structures are defined in nwsnut.h.

Figure 18. List Structure



NOTE: Fields in the LISTPTR and LIST structures should not be changed directly. NWSNUT functions keep the information in these structures current.

Related Topics

Basic Steps for Using Lists

Manipulating List Elements

Handling Lists

Routines for List Elements

Saving Lists

Specialized Lists

Manipulating List Elements

Once a list is created, its elements can be manipulated in several ways. Elements can be added, deleted, modified, marked and sorted.

The following lists functions used to add, delete and modify list elements:

NWSAppendToList

Adds an element to the end of the current list

NWSInsertInList, NWSInsertInPortalList

Inserts an element into a list at a specific location

NWSDeleteFromPortalList

Deletes current and marked list elements

NWSDeleteFromList

Delete a specified list element

NWSModifyInPortalList

Modifies the text field of a list element

Related Topics

Marking Lists

Sorting Lists

Parent Topic: Lists

Marking Lists

The user marks a list item by pressing **F5**. The marked item is displayed with a reverse normal video attribute and the *marked* field of the associated LIST structure is set. In this way, the user can select several list elements to perform an action on at one time. The following table summarizes functions that deal with marking list elements:

NWSIsAnyMarked

Determines whether any items in a list are marked

NWSUnmarkList

Changes the status of all list elements to unmarked

NWSPushMarks

Saves the marked status of all list elements in the current list

NWSPopMarks

Retrieves the saved marked status of all list elements in the current list.

Parent Topic: Manipulating List Elements

Menu Fields

Creating a menu field is more complicated than creating other fields. To create a menu field, you must complete the following steps:

Initialize the menu field by calling **NWSInitMenuField**.

Build the menu by calling **NWSAppendToMenuField** once for each menu option in the menu field.

Append the menu field to the form by calling **NWSAppendMenuField**.

The following example from NDEMO.C illustrates this process:

```
mfctl = NWSInitMenuField (FORM_MENU_HEADER, 10, 40,
   FormMenuAction, handle);
NWSAppendToMenuField (mfctl, MENU_TEXT_ONE, 1, handle);
NWSAppendToMenuField (mfctl, MENU_TEXT_TWO, 2, handle);
menuChoice = 1;           /* display the text for option one */
line += 2;
NWSAppendCommentField (line, 1, "Menu Field:", handle);
NWSAppendMenuField (line, 25, NORMAL_FIELD, &menuChoice,
   mfctl, NULL, handle);
```

When the user selects the menu field, the menu appears and behaves like a stand-alone menu.

Parent Topic: Forms

NWSNUT Example

To get a basic idea of how NWSNUT works, examine ndemo.c, which can be found in the EXAMPLES directory. Please note that this example demonstrates several different ways of handling messages in an NWSNUT NLM. Therefore, it is not as consistent as a typical NLM.

See Using NWSNUT Interface: Example.

NWSNUT Function List

NOTE: NWSNUT functions require that NWSNUT.NLM be loaded on the server. NetWare 3.12 requires that AFTER311.NLM be loaded also. NWSNUT works with NetWare 3.11 if AFTER311.NLM is loaded.

However, take into consideration that the end user might not have AFTER311.NLM.

NWSAlert

Displays an alert portal.

NWSAlertWithHelp

Displays an alert portal with access to help.

NWSAlloc

Allocates memory for NWSNUT purposes.

NWSAppendBoolField

Adds a yes/no field to a form.

NWSAppendCommentField

Adds a comment field to a form.

NWSAppendHexField

Adds a form field that accepts only hexadecimal input.

NWSAppendHotSpotField

Adds a form field that calls a routine when selected.

NWSAppendIntegerField

Adds a form field that accpts only a digital number as input.

NWSAppendMenuField

Adds a form field that displays a menu when selected.

NWSAppendPasswordField

Adds a form field that enables password input.

NWSAppendPromptField

Adds a prompt field to a form.

NWSAppendScrollableStringField

Appends a form field into which scrollable text can be entered.

NWSAppendStringField

Adds a form field containing an editable string.

NWSAppendToForm

Adds a customized field to a form.

NWSAppendToList

Adds an element to the current list.

NWSAppendToMenu

Adds an option to a menu.

NWSAppendToMenuField

Adds an option to a menu field.

NWSAppendUnsignedIntegerField

Adds a form field that only accepts an unsigned integer as input.

NWSAsciiHexToInt

Converts an ASCII-represented hexadecimal number to an integer.

NWSAsciiToInt

Converts an ASCII-represented decimal number to an integer.

NWSAsciiToLONG

Converts an ASCII-represented number to type LONG.

NWSClearPortal

Blanks out a portal.

NWSComputePortalPosition

Calculates the screen line and column for positioning a portal given its size.

NWSConfirm

Displays a yes/no portal allowing the user to confirm a decision.

NWSCreatePortal

Creates a portal.

NWSDeleteFromList

Deletes an element from a list.

NWSDeleteFromPortalList

Deletes current and marked elements from a list.

NWSDeselectPortal

Makes a portal inactive.

NWSDestroyForm

Destroys a form.

NWSDestroyList

Destroys a list.

NWSDestroyMenu

Destroys a menu.

NWSDestroyPortal

Destroys a portal.

NWSDisableAllFunctionKeys

Disables all function keys.

NWSDisableAllInterruptKeys

Disables all interrupt keys.

NWSDisableInterruptKey

Disables a specified interrupt key.

NWSDisableFunctionKey

Disables a function key.

NWSDisablePortalCursor

Disables the portal cursor.

NWSDisplayErrorCondition

Displays an error portal listing the routine that resulted in the error condition.

NWSDisplayErrorText

Displays an error portal.

NWSDisplayHelpScreen

Displays a help portal.

NWSDisplayInformation

Displays text in a portal. Palette, video attribute, and behavior can be specified.

NWSDisplayInformationInPortal

Displays text in a new portal. Justification of portal with respect to the screen, minimum and maximum size of portal, justification style of text, portal palette, video attribute of text, and minimization of text can be specified.

NWSDisplayPreHelp

Displays a pre-help message.

NWSDisplayTextInPortal

Displays text in an existing portal. Starting portal line for text and indent level can be specified.

NWSDisplayTextJustifiedInPortal

Same as above, but text width and video attribute can be specified.

NWSDrawPortalBorder

Redraws the border of a portal.

NWSEditForm

Displays a form and allows the user to edit it.

NWSEditPortalForm

Same as **NWSEditForm**, but help context can be specified.

NWSEditPortalFormField

Same as **NWSEditForm**, but help context and first field to highlight can be specified.

NWSEditString

Displays a string inside a portal and allows the user to edit it. Prompt text, input character restrictions, and action routines can be specified.

NWSEditText

Displays a string inside a portal and allows the user to edit it using the

NWSNUT screen editor.

NWSEditTextWithScrollBars

Allows the user to edit scrollable text within a portal.

NWSEnableAllFunctionKeys

Enables all function keys.

NWSEnableFunctionKey

Enables a function key.

NWSEnableFunctionKeyList

Enables a list of function keys.

NWSEnableInterruptKey

Enables an interrupt key and associates that key with a routine.

NWSEnableInterruptList

Enables a list of interrupt keys.

NWSEnablePortalCursor

Enables the portal cursor.

NWSEndWait

Removes a "please wait" portal.

NWSFillPortalZone

Fills a specified region of the portal with a character.

NWSFillPortalZoneAttribute

Fills a specified region of the portal with the video attribute.

NWSFree

Frees memory allocated with**NWSAlloc**.

NWSGetADisk

Displays a portal prompting the user to insert the specified floppy disk into the disk drive.

NWSGetDefaultCompare

Obtains the default compare function.

NWSGetFieldFunctionPtr

Obtains the customized routines for the specified field.

NWSGetHandleCustomData

Obtains the function for handling developer-defined data.

NWSGetKey

Reads one key from the keyboard buffer.

NWSGetLineDrawCharacter

Gets a line drawing character.

NWSGetList

Returns the pointers for the current list.

NWSGetListHead

Returns the first element in the current list.

NWSGetListNotifyProcedure

Obtains the routine to be called when a list element is highlighted.

NWSGetListSortFunction

Returns a pointer to the currently set list sort function.

NWSGetListTail

Returns the last element in the current list.

NWSGetMessage

Retrieves the specified message.

NWSGetNUTVersion

Returns the version of NWSNUT the NLM is using.

NWSGetPCB

Obtains the portal control block of a portal.

NWSGetScreenPalette

Returns the current screen palette.

NWSGetSortCharacter

Returns the weighted value used for sorting a given character.

NWSInitForm

Initializes a form.

NWSInitializeNut

Sets up NWSNUT context for your NLM.

NWSInitList

Initializes a list.

NWSInitListPtr

Initializes a list that is appended to another list or form.

NWSInitMenu

Initializes a menu.

NWSInitMenuField

Initializes a menu field for a form.

NWSInsertInList

Inserts an element into a list.

NWSInsertInPortalList

Inserts an element into a list using a specified insertion routine.

NWSIsAnyMarked

Indicates whether any elements in the current list are marked.

NWSIsdigit

Tests whether a character is an ASCII representation of a decimal number.

NWSIsxdigit

Tests whether a character is an ASCII representation of a hexadecimal number.

NWSKeyStatus

Indicates whether a key is waiting in the keyboard buffer.

NWSList

Displays a list and allows the user to perform list operations.

NWSMemmove

Copies bytes from one buffer to another.

NWSMenu

Displays a menu and allows the user to choose options from it.

NWSModifyInPortalList

Modifies the text field of a list element.

NWSPopHelpContext

Pops a help context off of the help stack.

NWSPopList

Pops a set of list pointers from the list stack.

NWSPopMarks

Retrieves the marked status of all list elements in the current list.

NWSPositionCursor

Positions the cursor relative to the entire screen.

NWSPositionPortalCursor

Positions the portal cursor within the portal.

NWSPromptForPassword

Enables a console operator to input a password to an NLM, with optional forced verification.

NWSPushHelpContext

Saves a help context onto the help stack.

NWSPushList

Pushes the current list pointers onto the list stack.

NWSPushMarks

Saves the marked status of all list elements in the current list.

NWSRemovePreHelp

Removes a pre-help portal.

NWSRestoreDisplay

Clears the screen.

NWSRestoreList

Takes the specified list from the save stack and makes it current.

NWSRestoreNut

Cleans up resources allocated by NWSNUT for your NLM.

NWSRestoreZone

Saves data in a buffer to the screen.

NWSSaveFunctionKeyList

Saves a list of function keys.

NWSSaveInterruptList

Saves a list of interrupt keys.

NWSSaveList

Saves the current list into the specified slot in the save stack.

NWSSaveZone

Saves a defined area on the screen to a buffer.

NWSScreenSize

Calculates the screen size.

NWSScrollPortalZone

Scrolls the portal display area the specified direction and number of lines.

NWSScrollZone

Allows a console operator to scroll the contents of a zone in a defined screen area, thus creating new lines.

NWSSelectPortal

Makes a portal active.

NWSSetDefaultCompare

Sets the default compare function.

NWSSetDynamicMessage

Stores a dynamic message into the MessageInfo structure.

NWSSetErrorLabelDisplayFlag

Sets the flag that determines whether **NWSDisplayErrorCondition** and **NWSDisplayErrorText** display NLM name and version information.

NWSSetFieldFunctionPtr

Sets the customized routines for a field.

NWSSetFormRepaintFlag

Repaints the form to show changes made to the form but not yet reflected on the screen.

reflected on the screen.

NWSSetHandleCustomData

Sets the function for handling developer-defined data.

NWSSetList

Makes the specified list current.

NWSSetListNotifyProcedure

Sets the routine to be called when a list element is highlighted.

NWSSetListSortFunction

Enables the use of a customized list sort function.

NWSSetScreenPalette

Sets the screen palette.

NWSShowLine

Displays text at a specified screen location.

NWSShowLineAttribute

Identical to **NWSShowLine**, but also allows screen attribute specification.

NWSShowPortalLine

Displays a line of text. Portal line and column can be specified.

NWSShowPortalLineAttribute

Displays text with a specified screen attribute.

NWSSortList

Sorts list elements.

NWSStartWait

Displays a "please wait" portal.

NWSStrcat

Appends a copy of one string to the end of another.

NWSToupper

Returns the uppercase value of the specified byte.

NWSTrace

Displays an information portal and waits for **Esc** to be pressed.

NWSUngetKey

Inserts a key into the keyboard buffer.

NWSUnmarkList

Removes marks from a list.

NWSUpdatePortal

Redraws a virtual portal to show changes made since creation or last update.

NWSViewText

Displays text within a portal.

NWSViewTextWithScrollBars

Displays scrollable text within a portal.

NWSWaitForEscape

Waits for the **Esc** key to be pressed.

NWSWaitForEscapeOrCancel

Waits for the **Esc** or Cancel (**F7**) key to be pressed.

NWSWaitForKeyAndValue

Waits until the user presses one of the keys in a specified set.

Menus

A **menu** is a specialized form of a list. However, the parameters necessary for building a menu are fewer than those for a list, since the specifics of the menu format are built into NWSNUT.

See Basic Steps for Using Menus.

NWSNUT Messages

Text strings displayed as titles (headers) or prompts are referred to as **messages**. To allow for translation and management, messages are usually not embedded (hard-coded) within an NLM, but are read from a message file.

This message file is located in the SYS:SYSTEM\NLS**nnn** directory, where **nnn** represents the number representing the language of the message file (for example, German, French, or English). The message file has the same name as the NLM with the extension .MSG (for example, the message file for ndemo.nlm is ndemo.msg). In addition to the message file in the NLS directory, a default set of messages can be linked to the NLM by using the Novell® linker NLMLINK.

An NLM using NWSNUT can specify its message file in either of two ways.

If the *messageTable* parameter for **NWSInitializeNut** is NULL, then NWSNUT retrieves the message file for the calling NLM, using the currently defined NLM language to determine which message file to read. Should there be no message file for the calling NLM in the appropriate NLS**nnn** directory, NWSNUT uses the default messages linked to the calling NLM.

Whether the message table is linked to or retrieved from a message file, the file containing the messages must be built with the Novell Message

Tools.

The calling NLM can pass to NWSNUT a pointer to a pointer to an array of messages as the *messageTable* parameter. The format of a message table passed in this manner is:

```
char *programMesgTable[] =
{
   "NetWare Loadable Module",
   "<Press ESCAPE To Continue>",
   "Please Wait",
   "Error Report"
};
```

Whenever NWSNUT requires that a parameter be a message identifier, such as parameter 1 in **NWSInitializeNut**, pass the array index (0, 1, and so on) of the desired message.

Related Topics

Creating Messages

Dynamic Messages

Alerts and Errors

Parent Topic: Text in NWSNUT

Other NWSNUT Functions

Other functions provided by NWSNUT are for getting the version of NWSNUT the NLM is currently using and for clearing the screen. Call **NWSGetNUTVersion** to get the version of NWSNUT the NLM is currently using and call **NWSRestoreDisplay** to clear the screen.

Portals

The portal is the central element of NWSNUT. A **portal** is a rectangle or "window" in which NWSNUT displays all output to the screen. Text, lists, menus, and forms are displayed within portals.

When a portal is created, NWSNUT creates a **Portal Control Block** (PCB) to manage the information about the portal. The PCB structure is defined in NWSNUT.H and described in the function description for **NWSGetPCB**. This structure holds information about the portal including its position on the screen, its frame size, and cursor state and position.

> **NOTE:** Fields in the PCB structure should not be changed directly. NWSNUT functions keep the information in this structure current.

Some of the portal control functions require a portal number. This is the value returned by **NWSCreatePortal**. Other portal functions require a pointer to a PCB. The PCB pointer can be obtained by calling **NWSGetPCB**.

Portal Types: Portals are classified as one of two types, direct or virtual, depending on whether the portal's screen access is direct or buffered. The portal type is determined by the value of the *directFlag* parameter (VIRTUAL or DIRECT) that is passed to **NWSCreatePortal** when the portal is created. See Virtual Portals and Direct Portals.

Related Topics

Basic Steps for Using Portals

Special Purpose Portals

Positioning the Portal

NWSNUT provides two functions that can help you calculate the positioning parameters to use when creating portals. **NWSScreenSize** returns the number of display lines and columns on the server screen. **NWSComputePortalPosition** returns the zero-based top row and left-most column (the line and column parameters for **NWSCreatePortal**), given the desired line and column to center the portal on.

Parent Topic: Basic Steps for Using Portals

Prompt Fields

Notice that **NWSAppendCommentField** and **NWSAppendPromptField** do essentially the same thing. The difference between the two functions is that **NWSAppendCommentField** takes a string (type BYTE) as input for its message, whereas **NWSAppendPromptField** takes a message identifier (see NWSNUT Messages).

Parent Topic: Forms

Routines for List Elements

Although the calling NLM turns control over to NWSNUT while the user makes a selection (when **NWSList** is called), the NLM can take back control when a list item is either highlighted or selected. After the NLM routine is completed, it returns to **NWSList**. In this manner the caller is not bothered by presentation specifics, but maintains control of the outcome of list selection.

By using the **list entry procedure**, the calling NLM can have control whenever the user moves the cursor to any item on the list. By using the **action procedure**, the calling NLM can have control whenever the user selects any item on the list. The entry procedure is part of theLIST structure and is set by calling **NWSSetListNotifyProcedure**. To obtain the entry procedure, call **NWSGetListNotifyProcedure**. The action procedure is specified when **NWSList** is called to allow the user to manipulate the list.

Parent Topic: Lists

Saving Lists

There are two ways to save a list. You can save it into the save stack or into the list stack. The **list stack** is a LIFO stack, so only the last list saved can be popped from the stack. You save a list into a specified slot in the**save stack**, so that list is available to be pulled from the stack at any time. **NWSPushList** and **NWSPopList** are used to save lists to the list stack. **NWSSaveList** and **NWSRestoreList** are used to save lists to the save stack. Both stacks are held in the NUTInfo structure.

Parent Topic: Lists

Screen Palette

The screen palette determines the colors of your portal. The screen palette can be set by calling **NWSSetScreenPalette**. NWSNUT provides the following palettes:

NORMAL_PALETTE

INIT_PALETTE

HELP_PALETTE

ERROR_PALETTE

WARNING_PALETTE

OTHER_PALETTE

To obtain the current screen palette, call **NWSGetScreenPalette**.

Parent Topic: Basic Steps for Using Portals

Scrolling the Portal

You can scroll the display area of a portal up or down by calling **NWSScrollPortalZone**. This function moves the display area up or down the number of lines that you specify. Either a direct or virtual portal can be scrolled. If a virtual portal is scrolled, you must call**NWSUpdatePortal** to transfer the scroll to the physical screen.

Parent Topic: Basic Steps for Using Portals

Selecting Portals

To select a portal, call **NWSSelectPortal**. Selecting a portal accomplishes the following:

Brings the portal to the front

Highlights the portal's border and title

Enables the portal cursor if it was flagged CURSOR_ON when the portal was created (PCB.cursorState == 1)

Deselecting the portal by calling **NWSDeselectPortal** dims the border and disables the portal cursor.

Parent Topic: Basic Steps for Using Portals

Sorting Lists

The list can be sorted by calling **NWSSortList**. This function uses the *defaultCompareFunction* from the NUTInfo structure to sort. If you want to create your own sorting routine, call **NWSSetDefaultCompare**. To obtain the current default compare function, call **NWSGetDefaultCompare**. **NWSGetSortCharacter** returns the weighted value used for sorting a given character. If you want to create you own list sort function call **NWSSetListSortFunction**. Call **NWSGetListSortFunction** to retrieve the current list sort function being used by the NLM.

NWSGetDefaultCompare

Determines whether any items in a list are marked.

NWSGetListSortFunction

Returns a pointer to the currently set list sort function.

NWSGetSortCharacter

Changes the status of all list elements to unmarked.

NWSSetDefaultCompare

Saves the marked status of all list elements in the current list.

NWSSetListSortFunction

Enables the use of a customized list sort function.

NWSSortList

Retrieves the saved marked status of all list elements in the current list.

Parent Topic: Manipulating List Elements

Special Purpose Portals

In addition to providing functions that allow you to create and manipulate portals, NWSNUT provides several special purpose portals. Each of these functions automatically creates a portal. The following summarizes these functions:

NWSDisplayInformation

Displays text in a portal. Video attribute can be specified for the text and palette can be specified for the portal.

NWSDisplayInformationInPortal

Displays text in a portal. Justification style, indention, video attribute, and text minimization style can be specified for the text. Minimum and maximum size and palette can be specified for the portal.

NWSViewText

Displays text within a portal.

NWSViewTextWithScrollBars

Displays scrollable text within a portal.

NWSShowLine

Displays text at a specified screen location.

NWSShowLineAttribute

Identical to **NWSShowLine**, but also allows screen attribute specification.

NWSEditString

Allows the user to edit text within a portal. Insertion and action routines can be specified, and input type can be restricted.

NWSEditText

Allows the user to edit text within a portal.

NWSEditTextWithScrollBars

Allows the user to edit scrollable text within a portal.

NWSStartWait

Creates a portal containing a "please wait" message.

NWSEndWait

Destroys the portal created by **NWSStartWait**.

NWSDisplayErrorCondition

Creates an error portal that displays an error message and the name of the procedure that resulted in the error.

NWSDisplayErrorText

Creates an error portal that displays an error message.

NWSAlert

Creates an alert portal.

NWSAlertWithHelp

Creates an alert portal with help context.

NWSConfirm

Creates a confirm portal.

NWSGetADisk

Creates a portal prompting the user to insert a floppy disk.

NWSTrace

Displays an information portal on the screen.

Some of these functions are mentioned later when user input is discussed. For detailed information about these functions, see the function descriptions in NWSNUT: Functions.

Parent Topic: Portals

Specialized Lists

Menus and Forms are special types of lists. They are created with different functions and are not directly manipulated as lists, but their NWSNUT-internal structure is based on a list.

Parent Topic: Lists

Text in NWSNUT

As a user interface, NWSNUT interacts with the user by displaying and receiving user information in the form of text. NWSNUT provides two primary methods of presenting textual information:

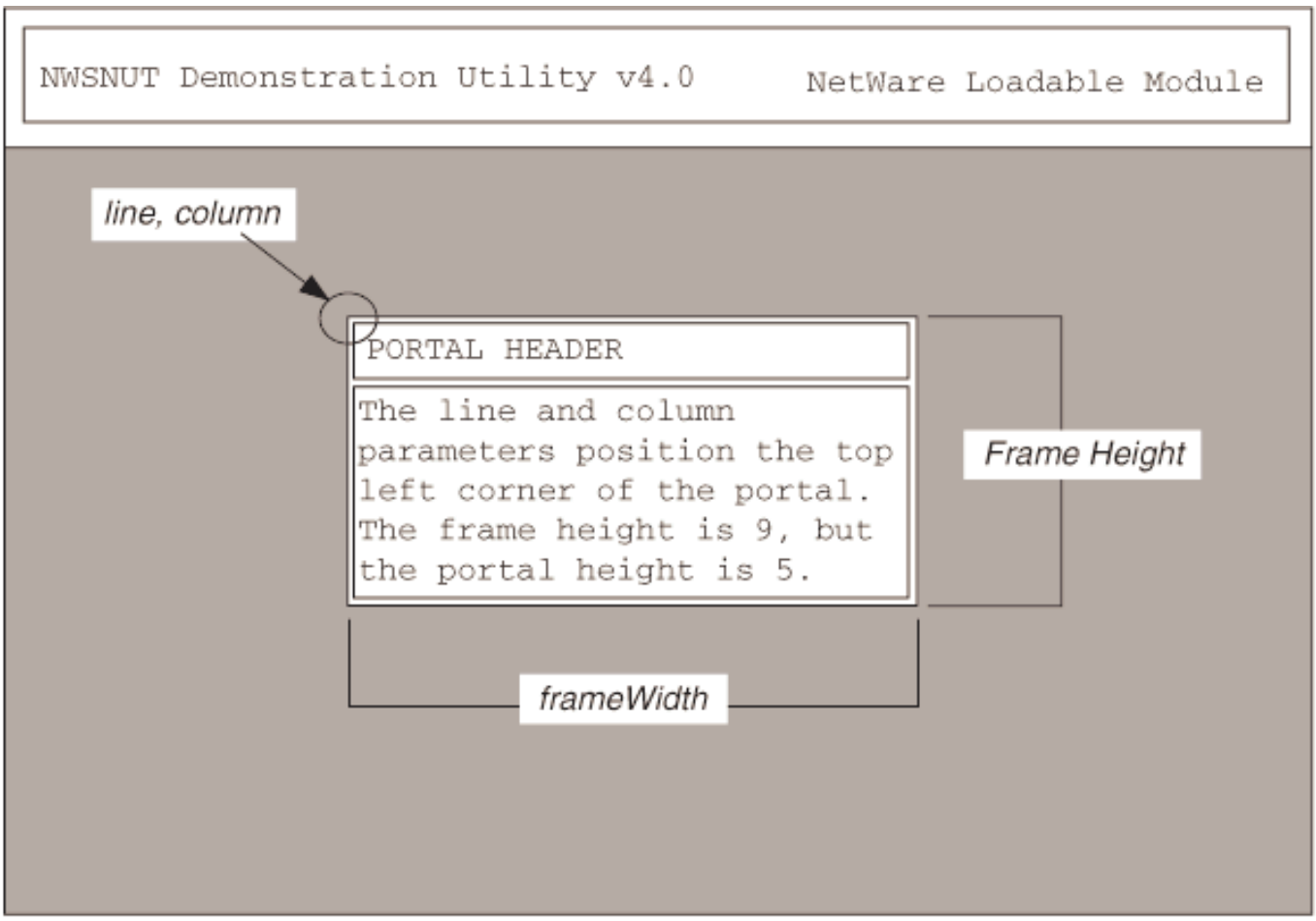
NWSNUT Messages

Help Screens

The NWSNUT Environment

The NWSNUT environment is created by calling **NWSInitializeNut**. This function creates the NWSNUT screen with header and a NUTInfo structure containing NWSNUT data. An example of the NWSNUT screen is illustrated in the following figure.

Figure 19. NWSNUT Screen Displaying a Portal



The NUTInfo structure is defined in nwsnut.h. This structure is used to keep track of the status of such things as the current portal, saved lists, help context, messages, and so forth for your NLM. A pointer to this structure is passed to various NWSNUT functions to keep track of the current state of NWSNUT

NOTE: Fields in the NUTInfo structure should not be changed directly. NWSNUT functions keep the information in this structure

current.

You can also place custom data into the NUTInfo structure and specify a custom data release function for it by calling **NWSSetHandleCustomData**. To retrieve the data and release this function, call **NWSGetHandleCustomData**.

When you have finished using the NWSNUT library, call **NWSRestoreNut**. This function cleans up resources used by NWSNUT.

The Portal Cursor

When you create a portal, you specify its cursor state as either CURSOR_ON (enabled) or CURSOR_OFF (disabled). When the portal is deselected, the cursor is always disabled. When it is selected, the cursor state is that specified when the portal was created. The following lists functions that NWSNUT provides for changing the cursor state and manipulating the cursor.

NWSEnablePortalCursor

Enables the portal cursor.

NWSDisablePortalCursor

Disables the portal cursor.

NWSPositionCursor

Positions the cursor relative to the entire screen.

NWSPositionPortalCursor

Positions the portal cursor within the portal.

User Input

NWSNUT provides several ways to receive user input. You can obtain key strokes, strings, fill text buffers, use forms, or you may let users choose a yes or no answer. In addition to regular keys, NWSNUT allows the use of function keys and interrupt keys.

See the following for information about processing input:

Keyboard Input in NWSNUT

Function Keys

Interrupt Keys

Editing a String

Confirmation of a Decision

Video Attribute

Video attribute refers to the manner in which a character on the screen is displayed. NWSNUT provides the following video attributes:

VNORMAL (normal)

VINTENSE (brighter)

VREVERSE (reverse video)

VBLINK (blinking characters)

VIBLINK (blinking intense characters)

VRBLINK (blinking reverse characters)

Parent Topic: Basic Steps for Using Portals

Virtual Portals

A **virtual portal** is an area of memory into which data intended for the portal is written. When **NWSUpdatePortal** is called, this data is transferred to the physical screen.

The size of the virtual display area is determined by the values of the *virtualHeight* and *virtualWidth* parameters passed to **NWSCreatePortal**. Only a section of the virtual display area the size of the portal's physical display area can be viewed on the screen. The portal must be scrolled to view hidden areas of the virtual display area. Virtual portals can be any size, regardless of screen size.

Changes to virtual portals are placed in a buffer and are not displayed until the portal is updated by **NWSUpdatePortal**.

Parent Topic: Portals

Zones

Zones are used to display text only anywhere on the console screen. NWSNUT provides three functions for manipulating zones. They are as follows:

NWSRestoreZone

Restores text saved in a buffer to the screen.

NWSSaveZone

Saves text in a defined area on the screen to a buffer.

NWSScrollZone

Enables text displayed in a defined area on the screen to be scrolled.

NWSNUT: Functions

NWSAlert

Displays an alert portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSAlert (
   LONG      centerLine,
   LONG      centerColumn,
   NUTInfo  *handle,
   LONG      message,
   ...);
```

Parameters

centerLine

(IN) Specifies the screen line to center the alert portal on.

centerColumn

(IN) Specifies the screen column to center the alert portal on.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM™ application.

message

(IN) Specifies the message identifier of the alert message.

Return Values

(0xFFFFFFFF)	Memory allocation error
(0xFFFFFFFE)	Portal creation error
(0xFF)	Escape key was pressed
(0xFE)	F7 key was pressed

Remarks

NWSAlert draws a portal, beeps the speaker, displays a message within the portal, and waits for an escape key. When the escape key is hit, it erases the portal.

NWSAlert uses the WARNING_PALETTE and prints the message in reverse video.

Optional parameters can be added at the end of the parameter list to satisfy the requirements of the *message* parameter (for example, if the message contains %d or %s).

If an alert portal exists on the screen by reason of a call to this routine, the *errorDisplayActive* field in the NUTInfo structure contains a nonzero value.

To enable the user to access help from the alert portal, call the **NWSAlertWithHelp** function.

See Also

NWSAlertWithHelp

NWSAlertWithHelp

Displays an alert portal with access to help screens
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSAlertWithHelp (
   LONG      centerLine,
   LONG      centerColumn,
   NUTInfo  *handle,
   LONG      message,
   LONG      helpContext,
   ...);
```

Parameters

centerLine

(IN) Specifies the screen line to center the alert portal on.

centerColumn

(IN) Specifies the screen column to center the alert portal on.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

message

(IN) Specifies the message identifier of the alert message.

helpContext

(IN) Specifies the help context.

Return Values

(0xFFFFFFFF)	Memory allocation error
(0xFFFFFFFE)	Portal creation error

(0xFF)	Escape key was pressed
(0xFE)	F7 key was pressed

Remarks

NWSAlertWithHelp draws a portal, beeps the speaker, displays a message within the portal, and waits for an escape key. When the escape key is hit, it erases the portal. **NWSAlertWithHelp** is identical to **NWSAlert**, except that help can be accessed from the portal. The *helpContext* parameter specifies the help context of the portal.

The **NWSAlertWithHelp** function used the WARNING_PALETTE and prints the message in reverse video.

Optional parameters can be added at the end of the parameter list to satisfy the requirements of the *message* parameter (for example, if the message contains %d or %s).

If an alert portal exists on the screen by reason of a call to this routine, the *errorDisplayActive* field in the NUTInfo structure contains a nonzero value.

See Also

NWSAlert

NWSAlignChangedList

Aligns the list display line with the portal frame
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 2.2, 3.11, 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <.h>

LONG NWSAlignChangedList (
   int       oldIndex,
   LIST     *newElement,
   LONG      oldLine,
   NUTInfo  *handle);
```

Parameters

oldIndex

(IN) Specifies the index within the list of the previously highlighted element (prior to any changes).

newElement

(IN) Points to the new element to be highlighted.

oldLine

(IN) Specifies the display line within the portal of the previously highlighted element.

handle

(IN) Points to the NUTInfo structure containing site information allocated to the calling NLM.

Return Values

Returns the new portal line to highlight if successful. Otherwise, zero is returned.

Remarks

NWSAlignChangedList is a list display function which calculates the portal line to be highlighted after insertion or deletion of one or more list elements.

Whenever the size of a list changes, call **NWSAlignChangedList**.

Call the **NWSGetListIndex** function to obtain the *oldIndex* parameter value before changing the list.

See Also

NWSDeleteFromList, **NWSGetListIndex**

NWSAlloc

Allocates memory for NWSNUT purposes
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void *NWSAlloc (
   LONG      numberOfBytes,
   NUTInfo  *handle);
```

Parameters

numberOfBytes

(IN) Specifies the number of bytes to allocate.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

NWSAlloc allocates memory for NWSNUT applications. It can be used by developers using NWSNUT. All memory allocated by **NWSAlloc** should be freed by calling the **NWSFree** or **NWSRestoreNut** functions.

See Also

NWSFree

NWSAppendBoolField

Appends a boolean choice field to a form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendBoolField (
   LONG      line,
   LONG      column,
   LONG      fflag,
   BYTE     *data,
   LONG      help,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the boolean field.

column

(IN) Specifies the portal column for the boolean field.

fflag

(IN) Specifies the field control flag.

data

(IN/OUT) Points to a byte in which to store the new boolean value. This value is also displayed on the form.

help

(IN) Specifies the help context for the boolean field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form

non NULL	Pointer to FIELD item that has been appended to form

Remarks

NWSAppendBoolField builds the menu list necessary and then appends the field to the current form.

The *data* parameter points to the boolean variable and can be changed.

The *fflag* parameter sets the value of the *fieldFlags* field in the FIELD structure. The *fieldFlags* field can have the following values:

Value	Meaning
NORMAL_FIELD	Normal, editable field.
LOCKED_FIELD	Nonaccessible field.
SECURE_FIELD	Noneditable field.
REQURED_FIELD	Verify field on form exit.
HIDDEN_FIELD	Hidden fields are not seen by the user. These fields are also locked.
PROMPT_FIELD	Prompt fields cannot be selected by the user. These fields are also locked.
ULOCKED_FIELD	Field locked by user.
FORM_DESELECT	Causes form deselection before action and verify routines are called.
NO_FORM_DESELECT	Form is not deselected before action and verify routines are called.
DEFAULT_FORMAT	Normal field-controlled justification.
RIGHT_FORMAT	Right justification.
LEFT_FORMAT	Left justification.
CENTER_FORMAT	Centered.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSInitForm

Example

See the example for **NWSInitForm**.

NWSAppendCommentField

Appends a comment field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendCommentField (
   LONG      line,
   LONG      column,
   BYTE     *prompt,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the comment field.

column

(IN) Specifies the portal column for the comment field.

prompt

(IN) Points to the comment text to be appended to the current form.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form
not NULL	Pointer to FIELD item that has been appended to form

Remarks

The comment field is a text string that appears in a form, but is not

available for editing in the form. It is often used as a prompt.

To specify prompt text by a message identifier, call **NWSAppendPromptField**.

If help is specified for a form field, it is not displayed unless the user presses **Enter**, followed by the **F1**.

The FIELD structure is described in NWSNUT: Structures.

See Also

NWSAppendPromptField, **NWSInitForm**

Example

See the example for **NWSInitForm**.

NWSAppendHexField

Appends a hexadecimal field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendHexField (
   LONG      line,
   LONG      column,
   LONG      fflag,
   int      *data,
   int       minimum,
   int       maximum,
   LONG      help,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the hex field.

column

(IN) Specifies the portal column for the hex field.

fflag

(IN) Specifies the field control flag.

data

(IN/OUT) Points to an integer in which to store the value of the new hex field. This value is also displayed on the form.

minimum

(IN) Specifies the minimum value that can be stored in the new hex field.

maximum

(IN) Specifies the maximum value that can be stored in the new hex field.

help

(IN) Specifies the help context for the new hex field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form
not NULL	Pointer to FIELD item that has been appended to form

Remarks

The Hex field is an integer size variable field that can be edited from a form. The characters (0 - 9) and (A - F) are the only input allowed when editing this field. The upper and lower limits on the value that can be input is specified by the *minimum* and *maximum* parameters.

The *data* parameter points to an integer where the value of the hex field is stored. This field can be edited.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See **NWSAppendBoolField** or NWSNUT.H for values.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSAppendUnsignedIntegerField, **NWSAppendIntegerField**, **NWSInitForm**

Example

See the example for **NWSInitForm**.

NWSAppendHotSpotField

Appends a hot spot field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendHotSpotField (
   LONG      line,
   LONG      column,
   LONG      fflag,
   BYTE     *displayString,
   LONG      (*SpotAction) (
      FIELD     *fp,
      int       selectKey,
      int       *changedField,
      NUTInfo   *handle),
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the hot spot field.

column

(IN) Specifies the portal column for the hot spot field.

fflag

(IN) Specifies the field control flag.

displayString

(IN) Specifies the string to be displayed in the form to mark the hot spot.

SpotAction

(IN) Specifies the routine to be called when the hot spot field is selected.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

The hot spot field is a field in the form that when selected calls a hot spot action routine. This type of field can be used to accomplish many desired actions from a form. For example, you can have an action routine that generates and calls a list.

If the hot spot field calls a list, you must call **NWSPushList** first, because the form is a specialized list. Call **NWSPopList** before returning to the form.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See **NWSAppendBoolField** or NWSNUT.H for values.

The *SpotAction* parameter sets the *fieldControl* field in the FIELD structure. This parameter specifies the routine to be called when this form field is selected.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSInitForm

Example

See the example for **NWSInitForm**.

NWSAppendIntegerField

Appends an integer field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendIntegerField (
   LONG      line,
   LONG      column,
   LONG      fflag,
   int      *data,
   int       minimum,
   int       maximum,
   LONG      help,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the integer field.

column

(IN) Specifies the portal column for the integer field.

fflag

(IN) Specifies the field control flag.

data

(IN/OUT) Points to an integer in which to store the value of the new hex field. This value is also displayed on the form.

minimum

(IN) Specifies the minimum value that can be stored in the new integer field.

maximum

(IN) Specifies the maximum value that can be stored in the new integer field.

help

(IN) Specifies the help context for the new integer field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

The comment field is a integer size variable field that can be edited from a form. The characters (0 - 9) are the only input allowed when editing this field. The upper and lower limits for input are set by the *minimum* and *maximum* parameters.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See **NWSAppendBoolField** or NWSNUT.H for values.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSAppendHexField, **NWSAppendUnsignedIntegerField**, **NWSInitForm**

Example

See the example for **NWSInitForm**.

NWSAppendMenuField

Appends a menu field to a form.
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendMenuField (
   LONG        line,
   LONG        column,
   LONG        fflag,
   int         data,
   MFCONTROL  *menu,
   LONG        help,
   NUTInfo    *nutInfo);
```

Parameters

line

(IN) Specifies the portal line for the menu field.

column

(IN) Specifies the portal column for the menu field.

fflag

(IN) Specifies the field attributes.

data

(OUT) Receives the option number of the selected menu option.

menu

(IN) Points to an MFCONTROL structure containing menu option information.

help

(IN) Specifies the help context for the menu.

nutInfo

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

This function appends a menu field to a form. If the menu field is selected the menu specified by *menu* is displayed and the user can choose from the options in the menu.

The *data* parameter receives the option value for the selected option (defined by **NWSAppendToMenuField**). This value is passed to the action routine associated with the menu (see **NWSInitMenuField**).

The *menu* parameter is the menu control structure returned by **NWSInitMenuField**. The MFCONTROL structure is described in **NWSInitMenu**.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See **NWSAppendBoolField** or NWSNUT.H for values.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSAppendToMenuField, **NWSInitMenuField**

Example

See the example for **NWSInitForm**.

NWSAppendPasswordField

Appends a password field to a form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD  *NWSAppendPasswordField (
   LONG      line,
   LONG      column,
   LONG      width,
   LONG      fflag,
   BYTE     *data,
   LONG      maxDataLen,
   LONG      help,
   LONG      verifyEntry,
   LONG      passwordPortalHeader,
   LONG      maskCharacter,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the line on which the password field appears.

column

(IN) Specifies the column on which the password field begins

width

(IN) Specifies the width of the password field.

fflag

(IN) Specifies the field control flag.

data

(IN) Points to a buffer that receives the password string.

maxDataLen

(IN) Specifies the maximum length of the string to which *data* points.

help

(IN) Specifies the help context.

verifyEntry

(IN) Provides for password verification.

passwordPortalHeader

(IN) Specifies the header string for the password box.

maskCharacter

(IN) Designates an optional ASCII character to mask the password.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Function did not allocate a field structure.
not NULL	Pointer to FIELD structure that has been allocated to the NLM.

Remarks

NWSAppendPasswordField sets up a form field that displayed the password, masked by a character of your choice. The function also provides a password entry portal that allows you to enter the password to be displayed in the form field.

You can begin entering the password in two ways. Either press the Enter key and begin typing the password, or simply begin typing. In the first case, the Enter key brings up the password entry portal, and the first alphanumeric character you type starts the password string. In the second case, the first alphanumeric character both brings up the password entry portal and starts the password string.

The following information refers to the parameters of **NWSAppendPasswordField**:

If you don't want a header for the password box, pass NO_MESSAGE to *passwordPortalHeader*.

To enable forced password verification, set *forceVerify* to TRUE. NULL disables password verification.

The *fflag* parameter sets the value of the *fieldFlags* field in the FIELD structure. The *fieldFlags* field can have the following values:

Value	Meaning
NORMAL_FIELD	Normal field that can be edited.

LOCKED_FIELD	Nonaccessible field.
SECURE_FIELD	Field that cannot be edited.
REQURED_FIELD	Verify field on form exit.
HIDDEN_FIELD	Hidden fields are not seen by the user. These fields are also locked.
PROMPT_FIELD	Prompt fields cannot be selected by the user. These fields are also locked.
ULOCKED_FIELD	Field locked by user.
FORM_DESELECT	Causes form deselection before action and verify routines are called.
NO_FORM_DESELECT	Form is not deselected before action and verify routines are called.
DEFAULT_FORMAT	Normal field-controlled justification.
RIGHT_FORMAT	Right justification.
LEFT_FORMAT	Left justification.
CENTER_FORMAT	Centered.

To accommodate the terminating null byte, make sure that the buffer to which *data* points is at least one byte longer than the string limited by *maxDataLen*.

NWSAppendPasswordField will display a confirmation portal only if you pass a nonzero value to *verifyEntry*. Zero disables the confirmation portal.

Although the character specified by *maskCharacter* covers the characters of the password in the field, no characters are displayed in the entry portal as the user enters the password.

See Also

NWSPromptForPassword

NWSAppendPromptField

Appends a prompt field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendPromptField (
   LONG      line,
   LONG      column,
   LONG      promptMessageNumber,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the prompt field.

column

(IN) Specifies the portal column for the prompt field.

promptMessageNumber

(IN) Specifies the message identifier of the message to be shown in the prompt field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

The prompt field is a noneditable field that appears in a form. It can be used for information or prompts. The*promptMessageNumber* parameter specifies the message identifier of the prompt to be displayed.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSAppendCommentField, **NWSInitForm**

Example

See the example for **NWSInitForm**.

NWSAppendScrollableStringField

Appends to a form a field into which scrollable text can be entered.
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendScrollableStringField (
   LONG      line,
   LONG      column,
   LONG      width,
   LONG      fflag,
   BYTE     *data,
   LONG      maxLen,
   BYTE     *cset,
   LONG      editFlags,
   LONG      help,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the line on which the scrollable field appears.

column

(IN) Specifies the column on which the left-most edge of the scrollable field appears

width

(IN) Specifies the width of the scrollable field.

fflag

(IN) Specifies the control *fieldFlags* field in the FIELD structure.

data

(IN/OUT) Points to the BYTE array where the string is stored. The string is displayed in the form field.

maxLen

(IN) Specifies the maximum number of characters the function will accept.

cset

(IN) Points to a NULL-terminated BYTE array of characters allowed as input to the string field.

editFlags

(IN) Specifies the text edit flag

help

(IN) Specifies the help context for the scrollable string field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

NWSAppendScrollableStringField creates a form field displaying a character array that can be edited when selected from the form. The string scrolls horizontally, and thus may be longer than the visible string field in the form.

The *data* parameter points to the string to be edited. An existing string can be specified, or the user can be prompted to enter a new string.

The *maxLen* parameter specifies the number of characters the function will accept. Make sure the buffer that accepts the string is at least one byte longer than *maxLen* to accommodate the terminating null byte.

The *cset* parameter defines allowable input for the scrollable field. This parameter is operative if you pass only EF_SET in the *editFlags* parameter; otherwise, *editFlags* determines allowable input. *cset* can be specified by a list of characters, a range of characters, or a combination of the two. For example, *cset* could be "ABCDEFG", "A..G", "a..z0..9A..Z", "0..9+-,.", and so on.

The *editFlags* parameter stipulates which characters are allowed in the field. These flags can be ORed, and may include one or more of the following:

Value	Meaning
EF_ANY	Allows any ASCII character

EF_DECIMAL	Allows only decimal characters
EF_HEX	Allows any hexadecimal character (letters may be upper or lower case)
EF_NOSPACES	Accepts any ASCII character, but disables space bar
EF_UPPER	Accepts all ASCII characters except lower case alphabetic
EF_DATE	Accepts only decimal characters, hyphen, and forward slash
EF_TIME	Accepts only decimal characters, colon, period, and lower case a, p, and m (converts upper case entries to lower case)
EF_FLOAT	Accepts only numerals and period
EF_SET	Accepts set defined in *cset* parameter if no other edit flag is set
EF_NOECHO	Disables appearance of the text on the screen as the text is being keyed in; accepts same character set as EF_ANY.
EF_FILENAME	Accepts all characters except , < > ? " | [ ] * + and =

If help is specified for a form field, it is not displayed unless the user presses **Enter**, followed by **F1**.

See Also

NWSAppendStringField, **NWSEditString**

NWSAppendStringField

Appends a string field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendStringField (
   LONG      line,
   LONG      column,
   LONG      width,
   LONG      fflag,
   BYTE     *data,
   BYTE     *cset,
   LONG      help,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the string field.

column

(IN) Specifies the portal column for the string field.

width

(IN) Specifies the maximum length of the string field.

fflag

(IN) Specifies the field control flag.

data

(IN/OUT) Points to the BYTE array where the string to be edited and displayed on the form is stored.

cset

(IN) Points to a NULL-terminated BYTE array of characters allowed for input when editing the new string field.

help

(IN) Specifies the help context for the new string field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

A string field is a character array that can be edited when selected from a form.

The *cset* can be specified by a list of characters, a range of characters, or a combination of the two. For example, *cset* could be "ABCDEFG", "A..G", "a..z0..9A..Z", "0..9+-,.", and so on.

The *data* parameter can specify an existing string or the user can be prompted to enter a new string.

The data in the string field cannot be longer than the value specified by the *width* parameter. Otherwise, the string is truncated and not editable.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See **NWSAppendBoolField** or NWSNUT.H for values.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSInitForm

Example

See the example for **NWSInitForm**.

NWSAppendToForm

Appends a customized field to a form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendToForm (
   LONG      fline,
   LONG      fcol,
   LONG      fwidth,
   LONG      fattr,
   void      (*fFormat) (
      struct fielddef  *field,
      BYTE             *text,
      LONG              buffLen),
   LONG      (*fControl) (
      struct fielddef  *field,
      int               selectKey,
      int              *fieldChanged,
      NUTInfo          *handle),
   int      (*fVerify) (
      struct fielddef  *field,
      BYTE             *data,
      NUTInfo          *handle),
   void      (*fRelease) (
      struct fielddef  *field),
   BYTE     *fData,
   BYTE     *fXtra,
   LONG      fflag,
   LONG      fActivateKeys,
   LONG      fhelp,
   NUTInfo  *handle);
```

Parameters

fline

(IN) Specifies the portal line for the new field.

fcol

(IN) Specifies the portal column for the new field.

fwidth

(IN) Specifies the width in characters of the new field.

fattr

(IN) Specifies the display attribute for field.

fFormat

(IN) Specifies the routine to format field, (NULL for default).

fControl

(IN) Specifies the routine to handle normal key input for this field, (NULL for default).

fVerify

(IN) Specifies the routine to verify field contents after editing, (NULL for default).

fRelease

(IN) Specifies the routine to free the *fData* and *fXtra* memory and release these parameters (NULL for default).

fData

(IN/OUT) Specifies data to be displayed in field. If the form is edited by the user, this field and *fXtra* can be modified by the user.

fXtra

(IN/OUT) Specifies additional data for field.

fflags

(IN) Specifies the field control flag.

fActivateKeys

(IN) Specifies the bit mask describing possible action keys for the field.

fhelp

(IN) Specifies the help context for the new field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form
not NULL	Pointer to FIELD item that has been appended to form

Remarks

NWSAppendToForm allows the developer to define a special-purpose field for a form. The developer can specify routines for formatting (*fFormat*), key input (*fControl*), input verification (*fVerify*), and memory release (*fRelease*) for the data in *fXtra* and *fData*.

The *fattr* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

The *fActivateKeys* parameter specifies the action keys for the field, as defined in NWSNUT.H. Possible values are:

M_ESCAPE	Escape key enabled.
M_INSERT	Insert key enabled.
M_DELETE	Delete key enabled.
M_MODIFY	Modify key (F3) enabled.
M_SELECT	Select key (Enter) enabled.
M_MDELETE	Delete key enabled for marked items.
M_CYCLE	Tab enabled.
M_MMODIFY	Modify key enabled for marked items.
M_MSELECT	Select key (Enter) enabled for marked items.
M_NO_SORT	Do not sort list.

These values can be ORed together to define a combination of action keys.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See

NWSAppendBoolField or NWSNUT.H for values.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSInitForm

Example

See the example for **NWSInitForm**.

NWSAppendToList

Appends an item and its customized data to the current list
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LIST *NWSAppendToList (
   BYTE     *text,
   void     *otherInfo,
   NUTInfo  *handle);
```

Parameters

text

(IN) Points to a string to be shown when the current list is presented.

otherInfo

(IN) Specifies customized data for the new item that is being appended to the current list.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append item to current list.
not NULL	Pointer to LIST item that has been appended to the current list.

Remarks

NWSAppendToList creates a structure of type LIST that defines a list item within the current list.

The *otherInfo* parameter is defined by the developer and can point to any

kind of information associated with the list item, including other lists. This parameter sets the *otherInfo* field in the LIST structure.

See Also

NWSAppendToList, **NWSInitList**, **NWSList**, **NWSSetListNotifyProcedure**

Example

See the example for **NWSInitList**.

NWSAppendToMenu

Adds an option to the current menu
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LIST *NWSAppendToMenu (
   LONG      message,
   LONG      option,
   NUTInfo  *handle);
```

Parameters

message

(IN) Specifies the message identifier of the message to be added to the current menu.

option

(IN) Specifies the value to be returned from the **NWSMenu** function if this element is selected.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append item to current list.
Non-NULL	Pointer to LIST item that has been appended to the current list.

Remarks

NWSAppendToMenu creates a structure of type LIST that defines a menu item within the current menu.

See Also

NWSDestroyMenu, **NWSInitMenu**, **NWSMenu**

Example

See the example for **NWSInitMenu**.

NWSAppendToMenuField

Adds a menu item to a menu associated with a field in a form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSAppendToMenuField(
   FMCONTROL  *m,
   LONG        optiontext,
   int         option,
   NUTInfo    *nutInfo);
```

Parameters

m

(IN) Points to an MFCONTROL structure containing the menu field information.

optiontext

(IN) Specifies the message identifier of the menu option text.

option

(IN) Specifies the menu option number.

nutInfo

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

Nonzero	Successful.
Zero	Not successful.

Remarks

NWSAppendToMenuField is used to build a menu that is associated with a field in a form. Call **NWSAppendToMenuField** once to add each option in the menu.

The *m* parameter is the MFCONTROL structure for the menu field returned by **NWSInitMenuField**.

The *optiontext* parameter specifies the message identifier of the text to display in the menu for this option.

The *option* parameter defines the integer to be passed to the menu action routine when this option is selected.

See Also

NWSAppendMenuField, **NWSInitMenuField**

Example

See the example for **NWSInitForm**.

NWSAppendUnsignedIntegerField

Appends an unsigned integer field to the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

FIELD *NWSAppendUnsignedIntegerField (
   LONG      line,
   LONG      column,
   LONG      fflag,
   LONG     *data,
   LONG      minimum,
   LONG      maximum,
   LONG      help,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the portal line for the integer field.

column

(IN) Specifies the portal column for the integer field.

fflag

(IN) Specifies the field control flag.

data

(IN/OUT) Points to an unsigned long in which to store the value of the new integer field. This field can be modified by the user.

minimum

(IN) Specifies the minimum value that can be stored in the new integer field.

maximum

(IN) Specifies the maximum value that can be stored in the new integer field.

help

(IN) Specifies the help context for the new integer field.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

NULL	Unable to append field to form.
not NULL	Pointer to FIELD item that has been appended to form.

Remarks

The unsigned integer field is a integer size variable field that can be edited from a form. The characters (0 - 9) are the only input allowed when editing this field. Upper and lower limits on the value of the integer are defined by the *minimum* and *maximum* parameters.

The *fflag* parameter sets the *fieldFlags* field in the FIELD structure. See **NWSAppendBoolField** or NWSNUT.H for values.

If help is specified for a form field, it is not displayed unless the user presses the Enter key, followed by the F1 key.

See Also

NWSAppendHexField, **NWSAppendIntegerField**, **NWSInitForm**

Example

See the example for **NWSInitForm**.

NWSAsciiHexToInt

Converts an ASCII-represented hexadecimal number to an integer
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSAsciiHexToInt (
   BYTE   *data);
```

Parameters

data

(IN) Points to a string that contains an ASCII-represented hexadecimal number.

Return Values

A signed integer.

Remarks

This function returns the integer value represented by the ASCII string *data*.

See Also

NWSAsciiToInt, **NWSAsciiToLONG**

NWSAsciiToInt

Converts an ASCII-represented decimal number to an integer
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSAsciiToInt (
   BYTE  *data);
```

Parameters

data

(IN) Points to a string that contains an ASCII-represented decimal number.

Return Values

A signed integer.

Remarks

This function returns the integer value represented by the ASCII string *data*.

See Also

NWSAsciiHexToInt, **NWSAsciiToLONG**

NWSAsciiToLONG

Converts an ASCII representation of a number to a number of type LONG
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSAsciiToLONG (
   BYTE  *data);
```

Parameters

data

(IN) Points to a string containing an ASCII representation of a number of type LONG.

Return Values

A number of type LONG.

Remarks

This function returns the LONG value represented by the ASCII string *data*.

See Also

NWSAsciiHexToInt, **NWSAsciiToInt**

NWSClearPortal

Blanks out the specified portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSClearPortal (
   PCB  *portal);
```

Parameters

portal

(IN) Points to a portal control block.

Return Values

None

Remarks

This function blanks out the portal specified by *portal*. The *portal* parameter can be obtained by calling **NWSGetPCB**.

See Also

NWSCreatePortal, **NWSDestroyPortal**, **NWSGetPCB**

NWSComputePortalPosition

Calculates the screen line and column for positioning a portal
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSComputePortalPosition (
   LONG      centerLine,
   LONG      centerColumn,
   LONG      height,
   LONG      width,
   LONG     *line,
   LONG     *column,
   NUTInfo  *handle);
```

Parameters

centerLine

(IN) Specifies the vertical line about which to center the portal.

centerColumn

(IN) Specifies the horizontal column about which to center the portal.

height

(IN) Specifies the height of the portal.

width

(IN) Specifies the width of the portal.

line

(OUT) Specifies the zero-based top row of the portal.

column

(OUT) Specifies the zero-based left column of the portal.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Success
Nonzero	Failure

Remarks

This function calculates the top row (*line*) and left-most column (*column*) for portal placement given its height, width, and which row and column it should be centered on. The output from this function can be used as input to **NWSCreatePortal**.

See Also

NWSCreatePortal

NWSConfirm

Draws a yes/no portal and allows the user to confirm a decision
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSConfirm (
   LONG      header,
   LONG      centerLine,
   LONG      centerColumn,
   LONG      defaultChoice,
   int       (*action) (
      int    option,
      void  *parameter),
   NUTInfo  *handle,
   void     *actionParameter);
```

Parameters

header

(IN) Specifies the message identifier for the header text.

centerLine

(IN) Specifies the row for the center of the confirm portal.

centerColumn

(IN) Specifies the column for the center of the confirm portal.

defaultChoice

(IN) Specifies the option to highlight (0 = No, 1 = Yes).

action

(IN) Specifies the optional action procedure to be called when user makes a selection (NULL = no action).

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

actionParameter

(IN) Specifies an optional parameter for the action procedure.

Return Values

-1	The user pressed ESCAPE.
0	The user selected NO.
1	The user selected YES.

Any other value returned indicates an error.

Remarks

This function draws a yes/no box that allows the user to confirm a choice. The *action* parameter specifies a routine to be called if yes or no is selected. The *option* parameter passed to the action routine indicates whether Yes (1) or No (0) was chosen.

NWSCreatePortal

Creates a NWSNUT portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSCreatePortal (
   LONG      line,
   LONG      column,
   LONG      frameHeight,
   LONG      frameWidth,
   LONG      virtualHeight,
   LONG      virtualWidth,
   LONG      saveFlag,
   BYTE     *headerText,
   LONG      headerAttribute,
   LONG      borderType,
   LONG      borderAttribute,
   LONG      cursorFlag,
   LONG      directFlag,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the screen line to place the top of the portal.

column

(IN) Specifies the screen column to place the left side of the portal.

frameHeight

(IN) Specifies the height of the new portal frame.

frameWidth

(IN) Specifies the width of the new portal frame.

virtualHeight

(IN) Specifies the height of the display area in the new portal.

virtualWidth

(IN) Specifies the width of the display area in the new portal.

saveFlag

(IN) Specifies whether to save the current data on the screen under this portal.

SAVE---saves current screen data.

NO_SAVE---does not save the screen data.

headerText

(IN) Specifies the header to be displayed at the top of the portal.

headerAttribute

(IN) Specifies the screen attribute to display the header with.

borderType

(IN) Specifies the border type of the portal (NOBORDER, SINGLE, or DOUBLE, as defined in NWSNUT.H).

borderAttribute

(IN) Specifies the Screen attribute for the border when the portal is selected.

cursorFlag

(IN) CURSOR_ON or CURSOR_OFF when the portal is drawn.

directFlag

(IN) Specifies whether to write to the physical or virtual screen. DIRECT---write directly to physical screen. VIRTUAL---write to virtual screen.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

If successful, this function returns the portal index of the new portal. Otherwise it returns a large value:

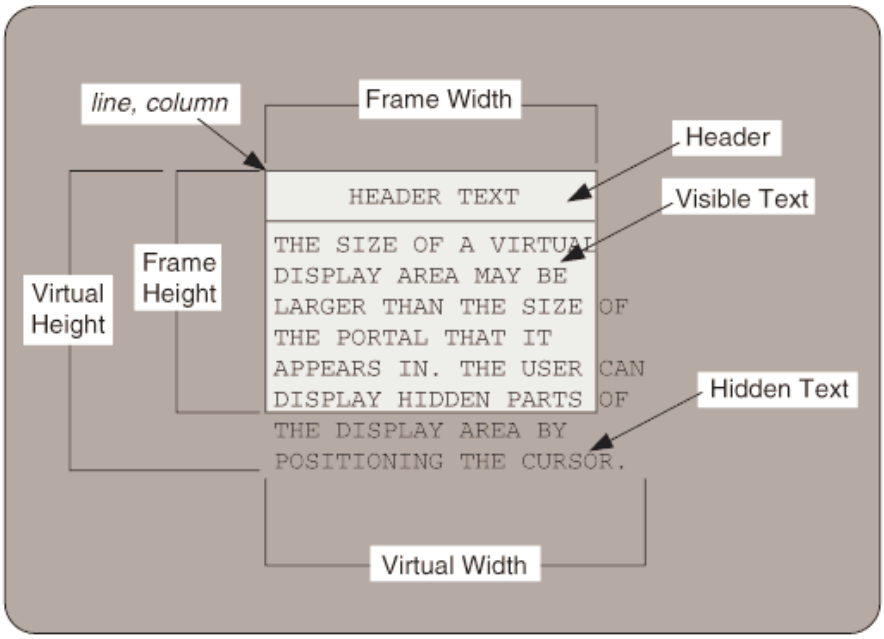
(0xFFFFFFFE)	Unable to allocate memory for PCB, virtual screen, or save area.
(0xFFFFFFFF)	Maximum number of portals already defined.

Remarks

This function returns the portal index number of the new portal. To obtain the portal control block (PCB) of the new portal, call **NWSGetPCB** for the portal index number.

The size of the virtual display (*virtualHeight* X *virtualWidth*) area can be greater than the size of the portal frame (*frameHeight* X *frameWidth*). The user can display hidden parts of the virtual display area by moving the cursor to a line or column that is hidden (or, in other words, scroll the text by positioning the cursor). There is no practical limit to the size of the virtual screen.

The following illustrates a portal.



The *directFlag* indicates whether to write to the physical or virtual screen. If DIRECT is specified, data is written to the physical screen, limiting the amount of data that can be written. If VIRTUAL is specified, the data written can be any size, but it is not written until **NWSUpdatePortal** is called.

The *saveFlag* parameter determines whether what is on the screen beneath the portal is saved. If *saveFlag* is SAVE, what the portal covers is redisplayed when the portal is destroyed.

NOTE: If several portals are displayed with the SAVE option, they must be destroyed in the opposite order from that in which they were created, because each call to NWSDestroyPortal restores what was on the screen when that portal was created.

The *headerAttribute* and *borderAttribute* parameters can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

See Also

NWSClearPortal, **NWSDestroyPortal**, **NWSDisplayTextInPortal**, **NWSDisplayTextJustifiedInPortal**, **NWSUpdatePortal**

NWSDeleteFromList

Removes the specified element from the current list
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LIST *NWSDeleteFromList (
   LIST     *deleteElement,
   NUTInfo  *handle);
```

Parameters

deleteElement

(IN) Points to the element to be deleted.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

If successful, a pointer to the next element in the list is returned.

If this is the last element in the list chain, a pointer to the previous list element is returned.

If the list is empty after deleting the element, NULL is returned.

Remarks

NWSDeleteFromList removes the element specified by the *deleteElement* parameter from the current list.

The *deleteElement* parameter specifies the LIST structure that was returned by the **NWSAppendToList** function when the element was created.

NOTE: **NWSDeleteFromList** assumes the *otherInfo* field of the LIST structure points to one contiguous memory block. If this memory block contains pointers to additional memory blocks, you must specifically

free these memory blocks before calling **NWSDeleteFromList**.

See Also

NWSAlignChangedList, **NWSAppendToList**, **NWSDeleteFromPortalList**, **NWSGetListIndex**, **NWSInsertInList**

NWSDeleteFromPortalList

Deletes all selected list elements (that is, current and marked elements) from the current list

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSDeleteFromPortalList (
   LIST     **currentElement,
   int      *currentLine,
   LIST     *(*deleteProcedure) (
      LIST     *el,
      NUTInfo  *handle,
      void      parameters),
   LONG      deleteCurrentMessageID,
   LONG      deleteMarkedMessageID,
   NUTInfo  *handle,
   void      parameters);
```

Parameters

currentElement

(IN) Specifies the element to be deleted.

currentLine

(IN) Specifies the current line of the list.

deleteProcedure

(IN) Points to the delete routine to be used.

deleteCurrentMessageID

(IN) Specifies the message identifier for the confirmation prompt if only the current list element is to be deleted.

deleteMarkedMessageID

(IN) Specifies the message identifier for the confirmation prompt if both the current and marked list elements are to be deleted.

Return Values

-1	List is empty, <Esc> was pressed, or "no" was chosen on confirmation.
0	One or more list items were deleted.

Remarks

This function determines whether more than one item is marked, then prompts the user with a confirm box to verify the deletion. The confirm box prompts the user with either the *deleteCurrentMessageID* (if no items are marked) or the *deleteMarkedMessageID* (if one or more items are marked).

The user marks a list item by highlighting it and pressing <F5>.

See Also

NWSAppendToList, **NWSDeleteFromList**, **NWSInsertInList**

NWSDeselectPortal

Deselects the currently active portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDeselectPortal (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function deselects the current portal and unhighlights its border. After this function is called, there is no current portal.

See Also

NWSSelectPortal

NWSDestroyForm

Frees all of the fields in the current form and reinitializes the form pointers
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDestroyForm (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function destroys the current form and reinitializes form pointers.

See Also

NWSInitForm

Example

See the example for **NWSInitForm**.

NWSDestroyList

Frees all of the nodes in the current list and initializes the list pointers
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDestroyList (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function destroys the current list and reinitializes the current list pointers.

See Also

NWSInitList, **NWSList**

Example

See the example for **NWSInitList**.

NWSDestroyMenu

Destroys the current menu
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDestroyMenu (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function frees the nodes in the current menu and frees the menu pointers.

See Also

NWSInitMenu, **NWSMenu**

Example

See the example for **NWSInitMenu**.

NWSDestroyPortal

Destroys a portal and cleans up all resources used by that portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDestroyPortal (
   LONG      portalNumber,
   NUTInfo  *handle);
```

Parameters

portalNumber

(IN) Specifies the portal index of the portal to be destroyed.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

The portal index number is returned by **NWSCreatePortal**.

See Also

NWSCreatePortal

NWSDisableAllFunctionKeys

Disables all function keys
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisableAllFunctionKeys (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

When the user presses a function key that has been previously defined and enabled, the computer beeps and ignore the input. The keystroke does not return from **NWSGetKey**. If the function key has not been defined, it is returned from **NWSGetKey**.

See Also

NWSDisableFunctionKey, **NWSEnableAllFunctionKeys**, **NWSEnableFunctionKey**, **NWSEnableFunctionKeyList**, **NWSSaveFunctionKeyList**

NWSDisableAllInterruptKeys

Disables all interrupt keys
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisableAllInterruptKeys (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function disables all previously defined interrupt keys and destroys the association between the keys and their associated routines.

See Also

NWSEnableInterruptKey, **NWSEnableInterruptList**, **NWSSaveInterruptList**

NWSDisableFunctionKey

Disables a function key which has previously had an interrupt procedure defined for it

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisableFunctionKey (
   LONG      key,
   NUTInfo  *handle);
```

Parameters

key

(IN) Specifies the key to be disabled.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

If *key* has ben previously defined and enabled, when the user presses *key* the computer beeps and ignore the input. The keystroke does not return from **NWSGetKey**. If *key* has not been defined, it is returned from **NWSGetKey**.

See Also

NWSDisableAllFunctionKeys, **NWSEnableFunctionKey**, **NWSEnableFunctionKeyList**, **NWSSaveFunctionKeyList**

NWSDisableInterruptKey

Enables a procedure to be called whenever a given key is pressed
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisableInterruptKey (
   LONG      key,
   NUTInfo  *handle);
```

Parameters

key

(IN) Specifies the interrupt key to disable.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function destroys the association between a defined interrupt key and an specified function. That association is established by calling **NWSEnableInterruptKey**.

See Also

NWSDisableAllInterruptKeys, **NWSEnableInterruptList**, **NWSEnableInterruptKey**, **NWSSaveInterruptList**

NWSDisablePortalCursor

Flags the cursor not to be shown when the specified portal is current
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisablePortalCursor (
   PCB  *portal);
```

Parameters

portal

(IN) Points to the NWSNUT portal control block of the portal for which to disable the cursor.

Return Values

None

Remarks

The *portal* parameter can be obtained by calling **NWSGetPCB**.

See Also

NWSEnablePortalCursor

NWSDisplayErrorCondition

Displays an error box listing the name of the routine which resulted in the error condition and an appropriate error message

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisplayErrorCondition (
   BYTE       *procedureName,
   int         errorCode,
   LONG        severity,
   PROCERROR  *errorList,
   NUTInfo    *handle,
   ...);
```

Parameters

procedureName

(IN) Specifies the name of the function that resulted in the error condition.

errorCode

(IN) Specifies the error code that occurred in the routine specified by *procedureName*.

severity

(IN) Specifies the severity of the error condition.

errorList

(IN) Points to a list of error codes and associated message identifiers in an array of type PROCERROR.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

At the top of the error box this function displays the name of the NLM as defined to the linker in the "description" line, followed by the version of the utility, and the message identifier in the message data base. This is displayed as Name-MajorVersion.MinorVersion-ErrorMessageNumber (for example, MyNLM-2.00-43). If you do not want this information to be displayed, call **NWSSetErrorLabelDisplayFlag**.

The *severity* parameter indicates the severity of the error and determines which message about program execution is displayed in addition to your message. This parameter can have one of the following values:

Severity	Message
SEVERITY_INFORM	Program execution should continue normally.
SEVERITY_WARNING	Program execution may not continue normally.
SEVERITY_FATAL	Program execution cannot continue normally.

If an error portal is displayed by this function, the *errorDisplayActive* field of the NUTInfo structure contains a nonzero value.

The developer builds an array of errors and associated messages for the *errorList* parameter. The PROCERROR structure is defined in NWSNUT.H as follows:

```
typedef struct PCERR_
{
   int   ccodeReturned;
   int   errorMessageNumber;
} PROCERROR;
```

Parameters required by the error message (for example, %s, %d) are optionally added to the end of the parameter list.

The error list must be terminated with a structure that contains the value -1 or -2 in the *ccodeReturned* field. If the value is set to -2, the message associated with *errorMessageNumber* is used as the default message for any error number which does not have a corresponding entry in the list. If the final entry is -1, a default message is displayed by NWSNUT.

See Also

NWSDisplayErrorText, **NWSSetErrorLabelDisplayFlag**

NWSDisplayErrorText

Displays an error portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisplayErrorText (
   LONG      message,
   LONG      severity,
   NUTInfo  *handle,
   ...);
```

Parameters

message

(IN) Specifies the message identifier of the error message.

severity

(IN) Specifies the severity of the error condition.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

At the top of the error box this function displays the name of the NLM as defined to the linker in the "description" line, followed by the version of the utility, and the message identifier in the message data base. This is displayed as Name-MajorVersion.MinorVersion-ErrorMessageNumber (for example, MyNLM-2.00-43). If you do not want this information to be displayed, call **NWSSetErrorLabelDisplayFlag**.

The *severity* parameter indicates the severity of the error and determines which message about program execution is displayed in addition to your

message. This parameter can have one of the following values:

Severity	Message
SEVERITY_INFORM	Program execution should continue normally.
SEVERITY_WARNING	Program execution may not continue normally.
SEVERITY_FATAL	Program execution cannot continue normally.

If an error portal is displayed by this function, the*errorDisplayActive* field of the NUTInfo structure contains a nonzero value.

Parameters required by *message* (for example, %s, %d) are optionally added to the end of the parameter list.

To display more detailed information, see **NWSDisplayErrorCondition**.

See Also

NWSDisplayErrorCondition, **NWSSetErrorLabelDisplayFlag**

NWSDisplayHelpScreen

Displays a help portal on the screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisplayHelpScreen (
   LONG      offset,
   NUTInfo  *handle);
```

Parameters

offset

(IN) Specifies the help identifier of the help message.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function displays a help screen. The user presses <Escape> to remove the help screen.

The offset parameter specifies the help identifier assigned to the help screen when it was created.

See Also

NWSPopHelpContext, **NWSPushHelpContext**

NWSDisplayInformation

Displays text in a portal on the screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSDisplayInformation (
   LONG      header,
   LONG      pauseFlag,
   LONG      centerLine,
   LONG      centerColumn,
   LONG      palette,
   LONG      attribute,
   BYTE     *displayText,
   NUTInfo  *handle);
```

Parameters

header

(IN) Specifies the message identifier for the header text.

pauseFlag

(IN) Specifies the behavior of the portal.

centerLine

(IN) Specifies the center row of the portal.

centerColumn

(IN) Specifies the center column of the portal.

palette

(IN) Specifies the palette to use for the display.

attribute

(IN) Specifies the screen attribute for the text.

displayText

(IN) Points the text to be displayed in the portal.

handle

(IN) Points to a NUTInfo structure that contains state information

allocated to the calling NLM.

Return Values

(0xFFFFFFFF)	Error.
(0xFF)	pauseFlag != 0 and escape key was hit.
(0xFE)	pauseFlag != 0 and F7 key was hit.

If *pauseFlag* == 0, the portal index is returned.

Remarks

The *pauseFlag* parameter can have the following values:

Value	**Portal Behavior**	**Message Displayed at Bottom of Portal**
0	Draw portal, display message, return.	(none)
1	Draw portal, display message, wait for ENTER, erase portal, return.	<Press ENTER to continue>
2	Draw portal, display message, wait for ENTER or F7, erase portal, return.	<Press CANCEL (F7) to abort>
3	Draw portal, enable help key, wait for ENTER, return	<Press HELP (F1) for more information>
4	Draw portal, wait for ESCAPE, return	<Press ESCAPE to continue>
5	Allow both escape and return	

The *palette* parameter can have one of the following values:

NORMAL_PALETTE	
INIT_PALETTE	
HELP_PALETTE	
ERROR_PALETTE	
WARNING_PALETTE	

OTHER_PALETTE	

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

See Also

NWSDisplayInformationInPortal, **NWSDisplayTextInPortal**, **NWSDisplayTextJustifiedInPortal**, **NWSViewText**

NWSDisplayInformationInPortal

Displays text in a portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSDisplayInformationInPortal (
   LONG      header,
   LONG      portalJustifyLine,
   LONG      portalJustifyColumn,
   LONG      portalJustifyType,
   LONG      portalPalette,
   LONG      portalBorderType,
   LONG      portalMaxWidth,
   LONG      portalMaxHeight,
   LONG      portalMinWidth,
   LONG      portalMinHeight,
   LONG      textLRJustifyType,
   LONG      textLRIndent,
   LONG      textTBJustifyType,
   LONG      textTBIndent,
   LONG      textAttribute,
   LONG      textMinimizeStyle,
   BYTE     *text,
   NUTInfo  *handle);
```

Parameters

header

(IN) Specifies the message identifier for the portal.

portalJustifyLine

(IN) Specifies the screen line to justify the portal frame against.

portalJustifyColumn

(IN) Specifies the screen column justify the portal frame against.

portalJustifyType

(IN) Specifies the type of justifiction for the portal frame.

portalPalette

(IN) Specifies the palette for the portal.

portalBorderType

(IN) Specifies the type of border for the portal (NOBORDER, SINGLE, or DOUBLE).

portalMaxWidth

(IN) Specifies the maximum width of the portal including borders.

portalMaxHeight

(IN) Specifies the maximum height of the portal including borders and header.

portalMinWidth

(IN) Specifies the minimum width of the portal including borders.

portalMinHeight

(IN) Specifies the minimum height of the portal including borders and header.

textLRJustifyType

(IN) Specifies how to justify the text from left to right.

textLRIndent

(IN) Specifies how to indent the text left to right.

textTBJustifyType

(IN) Specifies how to justify the text top to bottom.

textTBIndent

(IN) Specifies how the to indent text top to bottom.

textAttribute

(IN) Specifies the screen attribute for the text.

textMinimizeStyle

(IN) Specifies whether to minimize the text.

text

(IN) Points to the text to display in the portal.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

If successful, this function returns the portal index. Otherwise, one of the following error values is returned.

-1	The *text* parameter is a NULL pointer, memory cannot be allocated, or all portal slots are in use.

-2	Memory cannot be allocated for portal creation.
-3	Text does not fit in the specified area.
-4	The portal does not fit on the screen.

If an undefined value is returned, *text* does not fit in the portal, or *text* [0]==0.

Remarks

This function draws a portal, displays text in the portal, and returns without erasing the portal. This function allows the developer to specify the following:

Placement of the portal

Justification and indent of text within the portal

Minimization of the text

Minimum and maximum size of the portal

The *portalJustifyLine*, *portalJustifyColumn*, and *portalJustifyType* parameters position the portal on the screen. The *portalJustifyLine* is the line for top-bottom justification of the portal. Line 0 is the line below the screen header. The *portalJustifyColumn* is the column for left-right justification of the portal. Column 0 is the left screen edge. The *portalJustifyType* can have the following values:

JTOP
JBOTTOM
JRIGHT
JLEFT
JCENTER
JTOPLEFT
JTOPRIGHT
JBOTTOMLEFT
JBOTTOMRIGHT

The following bits are defined for the *palette* parameter:

	NORMAL_PALETTE
	INIT_PALETTE
	HELP_PALETTE
	ERROR_PALETTE
	WARNING_PALETTE
	OTHER_PALETTE

The *portalMaxWidth*, *portalMaxHeight*, *portalMinWidth*, and *portalMinHeight* parameters allow you to restrict the size of your portal. If you have no size preference, enter 0 for these parameters.

The *textMinimizeStyle* parameter indicates whether to display the text in smaller size:

SNORMAL	Display the text in normal size.
SMINWIDTH	Display the text with minimum width.
SMINHEIGHT	Display the text with minimum height.

The *textAttribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

The *textLRIndent* parameter does different things depending on the text's left-right justification style (*textLRJustifyType*). The following summarizes the meaning of *textLRIndent* for each value of *textLRJustifyType*:

textLRJustifyType	**textLRIndent specifies**
JCENTER	The number of spaces on both beginning and end of lines
JLEFT	The number of spaces on left side of lines
JRIGHT	The number of spaces on right side of lines

The *textTBIndent* parameter does different things depending on the text's top-bottom justification style (*textTBJustifyType*). The following summarizes the meaning of *textTBIndent* for each value of

textTBJustifyType:

textTBJustifyType	textTBIndent specifies
JCENTER	The number of blank lines on both top and bottom of portal
JTOP	The number of blank lines on top of portal
JBOTTOM	The number of blank lines on bottom of portal

See Also

NWSDisplayInformation, **NWSDisplayTextInPortal**, **NWSDisplayTextJustifiedInPortal**, **NWSViewText**

NWSDisplayPreHelp

Displays a prehelp portal on the screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDisplayPreHelp (
   LONG      line,
   LONG      column,
   LONG      message,
   NUTInfo  *handle);
```

Parameters

line

(IN) Specifies the line to center the prehelp portal on.

column

(IN) Specifies the column to center the prehelp portal on.

message

(IN) Specifies the message identifier of the prehelp message.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

The prehelp portal is a portal displaying a message that stays on the screen, such as "Press <F1> for help." To remove the portal, call **NWSRemovePreHelp**.

See Also

NWSRemovePreHelp

NWSDisplayTextInPortal

Displays text in an existing portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSDisplayTextInPortal (
   LONG   line,
   LONG   indentLevel,
   BYTE  *text,
   LONG   attribute,
   PCB   *portal);
```

Parameters

line

(IN) Specifies the starting portal line for the text.

indentLevel

(IN) Specifies the number of blank spaces at the beginning of each line.

text

(IN) Points to the text to display in the portal.

portal

(IN) Specifies the portal control block of the portal to display the text.

Return Values

If successful, this function returns the number of the next available line in the portal. If -1 is returned, the message does not fit in the portal.

Remarks

This function writes to either the virtual or the physical display area of an existing portal, depending on the *directFlag* in the portal's PCB structure (set by **NWSCreatePortal**). The text is wrapped if necessary.

The *indentLevel* parameter allows you to indent the text from the edge of the portal.

See Also

NWSDisplayInformation, **NWSDisplayInformationInPortal**, **NWSDisplayTextJustifiedInPortal**, **NWSViewText**

NWSDisplayTextJustifiedInPortal

Displays justified text in an existing portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSDisplayTextJustifiedInPortal (
   LONG   justify,
   LONG   line,
   LONG   column,
   LONG   textWidth,
   BYTE  *text,
   LONG   attribute,
   PCB   *portal);
```

Parameters

justify
(IN) Specifies the justification style for the text.

line
(IN) Specifies the starting portal line for the text.

column
(IN) Specifies the portal column for text justification.

textWidth
(IN) Specifies the maximum width of the text.

text
(IN) Points to the text to be displayed.

attribute
(IN) Specifies the display attribute for the text.

portal
(IN) Specifies the portal control block of the portal to display the text.

Return Values

If successful, this function returns the number of the next available line in the portal. If -1 is returned, the message does not fit in the portal.

Remarks

This function writes to either the virtual or the physical display area of an existing portal, depending on the *directFlag* in the portal's PCB (set by **NWSCreatePortal**). The text is wrapped if necessary.

This function is similar to **NWSDisplayTextInPortal**, but it allows the developer to specify justification and display attribute information for the text.

The meaning of the *column* parameter depends on the justification style specified by the *justify* parameter. The following describes the meaning of *column* for each value of *justify*:

justify	***column* specifies**
JCENTER	The center of each text line.
JLEFT	The left side of each text line
JRIGHT	The right side of each text line

If 0 is specified for *textWidth*, the limit on text line width is the portal or virtual display width.

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

See Also

NWSDisplayInformation, **NWSDisplayInformationInPortal**, **NWSDisplayTextInPortal**, **NWSViewText**

NWSDrawPortalBorder

Draws a border for the specified portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSDrawPortalBorder (
   PCB  *portal);
```

Parameters

portal

(IN) Specifies the portal control block for the portal to have the border.

Return Values

None

Remarks

This function draws a line box around the portal specified by *portal*. This function is often used to redraw portal borders. The *portal* parameter can be obtained by calling **NWSGetPCB**.

NWSEditForm

Displays the current form and allows the user to edit it
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSEditForm (
   LONG      headernum,
   LONG      line,
   LONG      col,
   LONG      portalHeight,
   LONG      portalWidth,
   LONG      virtualHeight,
   LONG      virtualWidth,
   LONG      ESCverify,
   LONG      forceverify,
   LONG      confirmMessage,
   NUTInfo  *handle);
```

Parameters

headernum

(IN) Specifies the message identifier for the header text.

line

(IN) Specifies the top-most row for the form portal.

col

(IN) Specifies the left-most column for the form portal.

portalHeight

(IN) Specifies the portal height.

portalWidth

(IN) Specifies the portal width.

virtualHeight

(IN) Specifies the displayable area height.

virtualWidth

(IN) Specifies the displayable area width.

ESCverify

(IN) Specifies whether to verify when the escape key is hit:

TRUE = verify ESCAPE key.

forceverify

(IN) Specifies whether to verify any changes made to the form:

TRUE = verify regardless of changes.

confirmMessage

(IN) Specifies the message identifier for the exit confirmation message.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Discard form
1	Save form
-1	Memory allocation or other error.

Remarks

This function creates a portal for the form, displays it, and allows the user to edit it.

The *ESCverify* and *forceverify* parameters indicate whether to verify changes or exit from the form. If either of these parameters are TRUE, a confirm box with the message specified by *confirmMessage* is displayed upon verification.

See Also

NWSEditPortalForm, **NWSEditPortalFormField**, **NWSInitForm**

NWSEditPortalForm

Displays the current form and allows the user to edit it
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSEditPortalForm (
   LONG      header,
   LONG      centerLine,
   LONG      centerColumn,
   LONG      formHeight,
   LONG      formWidth,
   LONG      controlFlags,
   LONG      formHelp,
   LONG      confirmMessage,
   NUTInfo  *handle);
```

Parameters

header

(IN) Specifies the message identifier for the header text.

centerLine

(IN) Specifies the screen line to center the form on.

centerColumn

(IN) Specifies the screen column to center the form on.

formHeight

(IN) Specifies the height of the form in screen rows.

formWidth

(IN) Specifies the width of the form in screen columns.

controlFlags

(IN) Specifies format and verification behavior of the form (see Remarks section below).

formHelp

(IN) Specifies the help context for the form. If no help context is desired, specify F_NOHELP.

confirmMessage

(IN) Specifies the message identifier for the message to be displayed to allow the user to confirm changes made to the form.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

-2	Portal is too large.
-1	Memory allocation or other error.
0	Discard form
1	Save form

Remarks

This function creates a portal for the form, displays it, and allows the user to edit it.

This function is similar to **NWSEditForm**, but requires less input. The *ESCVerify* and *forceVerify* parameters of **NWSEditForm** are replaced by F_VERIFY and F_FORCE values for the *controlFlags* parameter of **NWSEditPortalForm**. Moreover, **NWSEditPortalForm** does not allow the form to be larger than the portal, as **NWSEditForm** does.

See Also

NWSEditForm, **NWSEditPortalFormField**, **NWSInitForm**

Example

See the example for **NWSInitForm**.

NWSEditPortalFormField

Displays the current form and allows the user to edit it
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSEditPortalFormField (
   LONG      header,
   LONG      cline,
   LONG      ccol,
   LONG      formHeight,
   LONG      formWidth,
   LONG      controlFlags,
   LONG      formHelp,
   LONG      confirmMessage,
   FIELD    *startField,
   NUTInfo  *handle);
```

Parameters

header

(IN) Specifies the message identifier for the form's header text.

cline

(IN) Specifies the screen line to center the form on.

ccol

(IN) Specifies the screen column to center the form on.

formHeight

(IN) Specifies the height of the form in screen rows.

formWidth

(IN) Specifies the width of the form in screen columns.

controlFlags

(IN) Specifies format and verification behavior of the form (see Remarks section below).

formHelp

(IN) Specifies the help context for the form. If no help context is

desired, specify F_NOHELP.

confirmMessage

(IN) Specifies the message identifier of the text to be displayed to confirm changes to the form.

startField

(IN) Points to the field to highlight first.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

-2	Portal is too large.
-1	Memory allocation or other error.
0	Discard form
1	Save form

Remarks

This function is similar to **NWSEditPortalForm**, but **NWSEditPortalFormField** allows you to specify the starting field to highlight (that is, where the cursor is positioned when the user enters the form).

See Also

NWSEditForm, **NWSEditPortalForm**, **NWSInitForm**

NWSEditString

Allows the user to edit a string in a portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSEditString (
   LONG      centerLine,
   LONG      centerColumn,
   LONG      editHeight,
   LONG      editWidth,
   LONG      header,
   LONG      prompt,
   BYTE     *buf,
   LONG      maxLen,
   LONG      type,
   NUTInfo  *handle,
   int       (*insertProc) (
      BYTE  *buffer,
      LONG   maxLen,
      void  *parameters),
   int       (*actionProc) (
      LONG   action,
      BYTE  *buffer,
      void  *parameters),
   void     *parameters);
```

Parameters

centerLine

(IN) Specifies the center row of the new portal.

centerColumn

(IN) Specifies the center column of the new portal.

editHeight

(IN) Specifies the height of the new portal.

editWidth

(IN) Specifies the width of the new portal.

header

(IN) Specifies the message identifier for header text.

prompt

(IN) Specifies the message identifier for prompt text.

buf

(IN/OUT) On input, this parameter points to the display text. This text can be changed by the user during the edit process.

maxLen

(IN) Specifies the maximum length of the edit string.

type

(IN) Specifies the characters to accept.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

insertProc

(IN) Points to optional procedure to handle insertion of characters.

actionProc

(IN) Points to optional procedure to be called when user presses a key.

parameters

(IN) Specifies the characters allowed as input for editing the string.

Return Values

This function can return one of the following:

<0	Error	
1	E_ESCAPE	<Escape> was pressed to terminate editing
2	E_SELECT	<Select> was pressed to terminate editing
4	E_EMPTY	String is empty
8	E_CHANGE	String was changed

Otherwise, the value returned from the action procedure (*actionProc*) is returned (unless it returns -1, in which case editing continues).

Remarks

This function creates a portal and displays *buf* in the portal so that it can

be edited. The edited string is placed back in *buf*.

The length of the string can be too long to be displayed at one time in the portal. The user can move by pressing the left and right arrow keys to move the visible of the string. If the portal display area is more than one line tall, the text is wrapped if it overflows the first line.

The following bits have been defined for *type* parameter:

EF_ANY	Any type of input is accepted.
EF_DECIMAL	Only decimal digits are accepted (0 - 9).
EF_HEX	Only hexadecimal digits are accepted (0 - 9, A - F).
EF_NOSPACES	Spaces are not accepted.
EF_UPPER	Input is be converted to upper case.
EF_DATE	The input must be in date format.
EF_TIME	The input must be in time format.
EF_FLOAT	The input must be a floating-point number.
EF_SET	The input must be in the set of characters specified in parameters
EF_NOECHO	The input is not echoed to the screen.

The *parameters* parameter receives a character set that can be accepted for editing *buf*. For example, *parameters* could be "ABCDEFG", "A..G", "a..z0..9A..Z", "0..9+-,.", and so on.

The insertion procedure *insertProc* is called if the user presses the Insert key. If the completion code from *insertProc* is TRUE, the text is redisplayed and considered changed.

A completion code of -1 from *actionProc* causes the user to stay in the text-edit function. Any other value from *actionProc* is returned as a return value of **NWSEditString**.

See Also

NWSEditText

NWSEditText

Allows the user to edit text in a portal with the NWSNUT screen editor
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSEditText (
   LONG      centerLine,
   LONG      centerColumn,
   LONG      height,
   LONG      width,
   LONG      headerNumber,
   BYTE     *textBuffer,
   LONG      maxBufferLength,
   LONG      confirmMessage,
   LONG      forceConfirm,
   NUTInfo  *handle);
```

Parameters

centerLine

(IN) Specifies the center row of the new portal.

centerColumn

(IN) Specifies the center column of the new portal.

editHeight

(IN) Specifies the height of the new portal.

editWidth

(IN) Specifies the width of the new portal.

header

(IN) Specifies the message identifier for header text.

textBuffer

(IN/OUT) On input, this parameter points to the text to display. This text can be changed by the user during the edit process.

maxLen

(IN) Specifies the maximum length of the edit string.

confirmMessage

(IN) Specifies the message identifier for confirmation message when exiting.

forceConfirm

(IN) Boolean, force confirmation upon exit.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

Any combination of the following can be returned:

<0	Error	
1	E_ESCAPE	<Escape> was pressed to terminate editing
2	E_SELECT	<Select> was pressed to terminate editing
4	E_EMPTY	String is empty
8	E_CHANGE	String was changed

Remarks

This function is similar to **NWSEditString**, but it does not offer the ability to specify action or insertion procedures, and you cannot limit the type of input to the string.

The text does not wrap in this function except on `\n' characters.

See Also

NWSEditString

NWSEditTextWithScrollBars

Enables a console operator to input paragraphs of text into an NLM through a scrollable portal equipped with scrolling location indicator bars.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSEditTextWithScrollBars (
   LONG      centerLine,
   LONG      centerColumn,
   LONG      height,
   LONG      width,
   LONG      headerNumber,
   BYTE     *textBuffer,
   LONG      maxBufferLength,
   LONG      confirmMessage,
   LONG      forceConfirm,
   LONG      scrollBarFlag,
   NUTInfo  *handle);
```

Parameters

centerLine

(IN) Specifies the screen line to center the portal on.

centerColumn

(IN) Specifies the screen column to center the portal on.

height

(IN) Specifies the height of the portal.

width

(IN) Specifies the width of the portal.

headerNumber

(IN) Specifies the message identifier for header text.

textBuffer

(IN/OUT) On input, this parameter points to the text to display. This text can be changed by the user during the edit process.

maxBufferLength

(IN) Specifies the maximum length of the text.

confirmMessage

(IN) Specifies the message identifier for the exit confirmation message.

forceConfirm

(IN) Boolean, specifies whether to confirm changes.

scrollBarFlag

(IN) Specifies the presence and operation of the scroll bars.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

Any combination of the following can be returned:

<0	Error	
1	E_ESCAPE	<Escape> was pressed to terminate editing
2	E_SELECT	<Enter> was pressed to terminate editing
4	E_EMPTY	String is empty
8	E_CHANGE	String was changed

Remarks

The *centerLine* and *centerColumn* parameters specify the screen location of the portal. 0, 0 centers the portal on the screen, excluding the NLM header. Other values passed to *centerLine* and *centerColumn* locate the center of the portal relative to the top-most line below the NLM header and the left-most column of the screen. However, since these values designate the portal center, they must also take into account the dimensions of the portal itself.

The horizontal scroll bar represents the cursor position relative to the lines of text shown between the upper and lower portal boundaries, rather than in the buffer. The vertical scroll bar shows the cursor position relative to all text lines whether or not all lines are displayed on screen.

For an explanation of the *scrollBarFlag* parameter, see the "Remarks" section of **NWSViewTextWithScrollBars**.

The *forceConfirm* parameter indicates whether to verify changes on exit

from the form. If this parameter is TRUE, a confirm box with the message specified by *confirmMessage* is displayed upon verification.

See Also

NWSEditForm, **NWSEditText**, **NWSEditPortalForm**, **NWSEditPortalFormField**, **NWSInitForm**

Example

NWSEditTextWithScrollBars

```
ccode = NWSEditTextWithScrollBars(0, 0, 18, 78, DYNAMIC_MESSAGE_ONE,
     data, fileSize+1, DYNAMIC_MESSAGE_TWO, FALSE, SHOW_VERTICAL_SCROLL
     SHOW_HORIZONTAL_SCROLL_BAR | SHOW_CONSTANT_SCROLL_BARS, handle);
```

NWSEnableAllFunctionKeys

Enables all function keys
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEnableAllFunctionKeys (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function reverses the effect of **NWSDisableAllFunctionKeys**.

See Also

NWSDisableAllFunctionKeys, **NWSDisableFunctionKey**, **NWSEnableFunctionKey**, **NWSEnableFunctionKeyList**, **NWSSaveFunctionKeyList**

NWSEnableFunctionKey

Allows a function key to be used as input to NWSNUT routines
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEnableFunctionKey (
   LONG      key,
   NUTInfo  *handle);
```

Parameters

key

(IN) Function key to be enabled.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function reverses the effect of **NWSDisableFunctionKey**.

See Also

NWSDisableAllFunctionKeys, **NWSDisableFunctionKey**, **NWSEnableAllFunctionKeys**, **NWSEnableFunctionKeyList**, **NWSSaveFunctionKeyList**

NWSEnableFunctionKeyList

Allows a list of function keys to be used as input to NWSNUT routines
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEnableFunctionKeyList (
   BYTE     *keyList,
   NUTInfo  *handle);
```

Parameters

keyList

(IN) Points to a list of keys to be enabled.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function enables the function keys specified by the *keyList* parameter.

See Also

NWSDisableAllFunctionKeys, **NWSDisableFunctionKey**, **NWSEnableAllFunctionKeys**, **NWSEnableFunctionKey**, **NWSSaveFunctionKeyList**

NWSEnableInterruptKey

Enables a procedure to be called whenever a given key is pressed
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEnableInterruptKey (
   LONG      key,
   void      (*interruptProc) (
      void  *handle),
   NUTInfo  *handle);
```

Parameters

key

(IN) Specifies the key to link the procedure to.

interruptProc

(IN) Points to the procedure to be called when the interrupt key is pressed.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function associates *interruptProc* with *key* so that the routine specified by *interruptProc* is called when *key* is pressed. To destroy this association, call **NWSDisableInterruptKey**.

See Also

NWSDisableAllInterruptKeys, **NWSDisableInterruptKey**, **NWSEnableInterruptList**

NWSEnableInterruptList, NWSSaveInterruptList

NWSEnableInterruptList

Enables the list of interrupt keys saved in the INTERRUPT structure
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEnableInterruptList (
   INTERRUPT  *interruptList,
   NUTInfo    *handle);
```

Parameters

interruptList

(IN) Points to the first element in an array of pointers to INTERRUPT structures (see "Remarks" below).

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function enables a list of interrupt keys defined by the user, or more commonly, retrieved by a call to **NWSSaveInterruptList**.

The *interruptList* parameter points to the first element in an array of pointers to INTERRUPT structures. The size of the array should not exceed the value MAXFUNCTIONS+1, where MAXFUNCTIONS is a value defined in NWSNUT.H. The final element in the array should be a NULL pointer or an INTERRUPT structure in which the *interruptProc* field is set to NULL.

The INTERRUPT structure is defined in NWSNUT.H as follows:

```
typedef struct INT_
```

```
{
   void   (*interruptProc)(void *handle);
   LONG   key;
} INTERRUPT;
```

Functions can be passed to NWSNUT by using **NWSEnableInterruptList**
.

See Also

NWSDisableAllInterruptKeys, **NWSEnableInterruptKey**, **NWSSaveInterruptList**

NWSEnablePortalCursor

Flags the cursor to be shown when the specified portal is current
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEnablePortalCursor (
   PCB  *portal);
```

Parameters

portal

(IN) Points to a NWSNUT portal control block.

Return Values

None

Remarks

This function enables the cursor of *portal*. The *portal* parameter can be obtained by calling **NWSGetPCB**.

See Also

NWSDisablePortalCursor

NWSEndWait

Removes the wait portal displayed by **NWSStartWait**
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSEndWait (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function removes a wait portal created by **NWSStartWait**.

See Also

NWSStartWait

NWSFillPortalZone

Fills the specified region of a NWSNUT portal with characters of the specified attribute

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSFillPortalZone (
   LONG   line,
   LONG   column,
   LONG   height,
   LONG   width,
   LONG   fillCharacter,
   LONG   fillAttribute,
   PCB   *portal);
```

Parameters

line

(IN) Specifies the top-most line of the portal to fill.

column

(IN) Specifies the left-most column of the portal to fill.

height

(IN) Specifies the height of the region to be filled.

width

(IN) Specifies the width of the region to be filled.

fillCharacter

(IN) Specifies the character to fill the region with.

fillAttribute

(IN) Screen attribute of *fillCharacter.*

portal

(IN) Points to a NWSNUT portal control block.

Return Values

None

Remarks

This function allows you to fill a specified region of a portal with characters of a specific screen attribute. The *portal* parameter can be obtained by calling **NWSGetPCB**.

The *fillAttribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

See Also

NWSFillPortalZoneAttribute

Example

NWSFillPortalZone

```
#include <nwsnut.h>
#include <string.h>

NUTInfo  *handle;

main ()
{
   LONG   portalNumber;
   PCB    pPtr;
   BYTE  *ptr;
   portalNumber = NWSCreatePortal(6, 20, 10, 40, 6, 38,
      SAVE, "Demonstration Portal", 0,
      DOUBLE, 0, CURSOR_ON, VIRTUAL, handle);
   NWSGetPCB (&pPtr, portalNumber, handle);
   NWSClearPortal (pPtr);
   NWSSelectPortal (portalNumber, handle);
   ptr = "Zone below filled with a character";
```

```
NWSShowPortalLine (2, 0, ptr, strlen (ptr),    pPtr);
NWSFillPortalZone (3, 0, 3, 38, '@', VINTENSE, pPtr);
NWSUpdatePortal (pPtr);            /* cause it to be displayed on th
NWSGetKey (&type, &value, handle);   /* wait for a key to be pressed
NWSDestroyPortal (portalNumber, handle);
```

NWSFillPortalZoneAttribute

Changes the video attribute of all characters in the specified region of a NWSNUT portal

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSFillPortalZoneAttribute (
   LONG   line,
   LONG   column,
   LONG   height,
   LONG   width,
   LONG   attribute,
   PCB   *portal);
```

Parameters

line

(IN) Specifies the top-most row of the portal zone.

column

(IN) Specfies the left-most column of the portal zone.

height

(IN) Specifies the height of region to be filled.

width

(IN) Specifies the width of region to be filled.

attribute

(IN) Specifies the screen attribute for the fill zone.

portal

(IN) Points to a NWSNUT portal control block.

Return Values

None

Remarks

This function allows you to fill a specified region of a portal with a specific screen attribute. The *portal* parameter can be obtained by calling **NWSGetPCB**.

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

See Also

NWSFillPortalZone

Example

NWSFillPortalZoneAttribute

```
#include <nwsnut.h>
#include <string.h>

NUTInfo  *handle;

main ()
{
   LONG   portalNumber;
   PCB    pPtr;
   BYTE  *ptr;

   portalNumber = NWSCreatePortal(6, 20, 10, 40, 6, 38,
      SAVE, "Demonstration Portal", 0,
      DOUBLE, 0, CURSOR_ON, VIRTUAL, handle);
   NWSGetPCB (&pPtr, portalNumber, handle);
   NWSClearPortal (pPtr);
   NWSSelectPortal (portalNumber, handle);
   ptr = "This line shows reverse video filling";
   NWSShowPortalLine (0, 0, ptr, strlen (ptr),    pPtr);
   NWSFillPortalZoneAttribute (0, 0, 1, 38, VREVERSE, pPtr);
   NWSUpdatePortal (pPtr);        /* cause it to be displayed on the sc
```

```
   NWSGetKey (&type, &value, handle);
                                                    /* wait for a key to be pre
   NWSDestroyPortal (portalNumber, handle);
}
```

NWSFree

Frees memory allocated by **NWSAlloc**
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSFree (
   void     *address,
   NUTInfo  *handle);
```

Parameters

address

(IN) Points to the address of memory to free.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function frees memory allocated by **NWSAlloc** at *address*.

See Also

NWSAlloc

NWSGetADisk

Prompts the user to insert the specified floppy disk into the disk drive
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSGetADisk (
   BYTE     *volName,
   BYTE     *prompt,
   NUTInfo  *handle);
```

Parameters

volName

(IN) Specifies the floppy volume name.

prompt

(IN) Specifies the prompt to display.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

`A'	(0x41)	Successful.
	(0xFF)	Unsuccessful.

Remarks

This function prompts the user to insert a specific floppy disk into the disk drive.

NWSGetDefaultCompare

Obtains the current routine for comparing list elements
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetDefaultCompare (
   NUTInfo  *handle,
   int       (**defaultCompareFunction)(
      LIST  *el1,
      LIST  *el2));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

defaultCompareFunction

(OUT) Receives the current default compare function.

Return Values

None

Remarks

This function returns the current default compare function in the *defaultCompareFunction* parameter. To specify a new default compare function, call **NWSSetDefaultCompare**.

See Also

NWSSetDefaultCompare, **NWSSortList**

NWSGetFieldFunctionPtr

Obtains the routines associated with the specified field in a form
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetFieldFunctionPtr (
   FIELD  *fp,
   void    (**Format) (
      FIELD*,
      BYTE *text,
      LONG),
   LONG    (**Control) (
      FIELD *,
      int,
      int *,
      NUTInfo *),
   int     (**Verify) (
      FIELD *,
      BYTE *,
      NUTInfo *),
   void    (**Release) (
      FIELD *),
   void    (**Entry) (
      FIELD *,
      void *,
      NUTInfo *),
   void    (**customDataRelease) (
      void *,
      NUTInfo *));
```

Parameters

fp

(IN) Points to the field for which to return information.

Format

(OUT) Receives the formatting routine for *fp*.

Control

(OUT) Receives the control routine for *fp*.

Verify

(OUT) Receives the verify routine for *fp*.

Release

(OUT) Receives the memory release routine for *fp*.

Entry

(OUT) Receives the routine to be called for all entries in the form.

customDataRelease

(OUT) Receives the routine to release memory for releasing memory allocated for custom data for *fp*.

Return Values

None

Remarks

To set the routines for the field specified by the *fp* parameter, call **NWSSetFieldFunctionPtr**.

See Also

NWSSetFieldFunctionPtr

NWSGetHandleCustomData

Obtains the custom data and custom data release function that is held in the NUTInfo structure

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetHandleCustomData (
   NUTInfo  *handle,
   void     *customData,
   void      (*customDataRelease) (
      void     *theData,
      NUTInfo  *handle));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

customData

(OUT) Receives the custom data held in the NUTInfo structure.

customDataRelease

(OUT) Receives the custom data release function held in the NUTInfo structure.

Return Values

None

Remarks

This function returns the contents of the*customData* and *customDataRelease* fields of the NUTInfo structure. To define these values, call **NWSSetHandleCustomData**.

See Also

NWSSetHandleCustomData

NWSGetKey

Reads one key from the keyboard buffer
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetKey (
   LONG     *type,
   BYTE     *value,
   NUTInfo  *handle);
```

Parameters

type

(OUT) Receives the retrieved key type.

value

(OUT) Receives the retrieved key value.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function receives the key type and value of input from the keyboard.

The key type is the K_ constant as defined in NWSNUT.H under "keyboard constants". The *value* parameter receives the character associated with the key.

See Also

NWSKeyStatus, **NWSUngetKey**

NWSGetLineDrawCharacter

Gets a line drawing character
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSGetLineDrawCharacter (
   LONG   charIndex);
```

Parameters

charIndex

(IN) Specifies the index of the line drawing character (0 - 47).

Return Values

If successful, this function returns the line drawing character. Otherwise, 0 is returned.

Remarks

This function returns the line drawing character for the index *charIndex*. See the character and key constants defined in NWSNUT.H (F_H1 through F_BG4).

NWSGetList

Returns a structure that contains the pointers for the current list
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetList (
   LISTPTR  *listPtr,
   NUTInfo  *handle);
```

Parameters

listPtr

(IN/OUT) Specifies the address of a pointer that points at the current list item.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function returns the list pointer to the current list in the*listPtr* parameter.

See Also

NWSGetListHead, **NWSSetList**

NWSGetListHead

Returns the first element in the current list
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LIST *NWSGetListHead (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

This function returns the list element that is the head of the current list.

Remarks

This function returns a LIST structure defining the first element in the current list.

See Also

NWSGetList, **NWSGetListTail**

NWSGetListIndex

Returns the index of the specified list element
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSGetListIndex (
   LIST     *listElement,
   NUTInfo  *handle);
```

Parameters

listElement

(IN) Points to the element of the current list whose index is to be returned.

handle

(IN) Points to the NUTInfo structure containing state information allocated to the calling NLM.

Return Values

If successful, returns the index of the specified element. Otherwise, 0 is returned.

Remarks

List indexes are zero-based; the first element in the list has an index of 0.

See Also

NWSAlignChangedList, **NWSGetList**, **NWSGetListHead**, **NWSGetListTail**

NWSGetListNotifyProcedure

Obtains the routine to be called when a list element is selected
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetListNotifyProcedure (
   LIST  *el,
   void   (**entryProcedure) (
      LIST     *element,
      LONG      displayLine,
      NUTInfo  *handle));
```

Parameters

el

(IN) Points to the element for which to call the routine.

entryProcedure

(OUT) Receives the currently defined notify procedure.

Return Values

None

Remarks

This function is used to obtain the routine that is called when *el* is selected.

The *entryProcedure* parameter receives the routine that is called when *el* is selected. This routine is passed the following parameters:

element	The selected list element.
displayLine	The display line of the selected list element.
handle	The NUTInfo structure containing NWSNUT state

	information for your NLM.

See Also

NWSSetListNotifyProcedure

NWSGetListSortFunction

Returns a pointer to the currently set list sort function
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetListSortFunction (
   NUTInfo *handle,
   void     (**listSortFunction)(
      LIST     *head,
      LIST     *tail,
      NUTInfo  *handle));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

listSortFunction

(IN) Points to the pointer of the currently set list sort function.

Return Values

None

Remarks

This function allows you to obtain a customized list sort function previously set by **NWSSetListSortFunction**. The *listSortFunction* parameters *head* and *tail* designate the first and last links in the list.

See Also

NWSSetListSortFunction

NWSGetListTail

Returns the last element in the current list
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LIST *NWSGetListTail (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

This function returns the list element that is the tail of the current list.

Remarks

This function returns a LIST structure defining the last element in the current list.

See Also

NWSGetListHead

NWSGetMessage

Retrieves a message from the specified message buffer
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

BYTE *NWSGetMessage (
   LONG          message,
   MessageInfo  *messages);
```

Parameters

message

(IN) Specifies a message identifer: DYNAMIC_MESSAGE_ONE, DYNAMIC_MESSAGE_TWO, and so on, up to DYNAMIC_MESSAGE_FOURTEEN.

messages

(IN) Points to a buffer that contains the NWSNUT interface messages.

Return Values

This function returns a pointer to message text.

Remarks

This function returns a pointer to the message text associated with the message identifier *message*.

See Also

NWSSetDynamicMessage

NWSGetNUTVersion

Returns the version of NWSNUT currently loaded on the server
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetNUTVersion (
   LONG  *majorVersion,
   LONG  *minorVersion,
   LONG  *revision);
```

Parameters

majorVersion

(IN) Points to where to write the major version number.

minorVersion

(IN) Points to where to write the minor version number.

revision

(IN) Points to where to write the revision number.

Return Values

None

Remarks

NWSGetNUTVersion allows you or your NLM to find out which version of NWSNUT is currently running.

NWSGetPCB

Returns a pointer to the portal control block (PCB) for the specified portal
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSGetPCB (
   PCB      *pPcb,
   LONG      portalNumber,
   NUTInfo  *handle);
```

Parameters

pPcb

(OUT) Points to the PCB of the specified portal.

portalNumber

(IN) Specifies the portal index number of the portal.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function returns a pointer to the PCB structure in the *pPcb* parameter for the portal specified by *portalNumber*. The portal number is returned by **NWSCreatePortal** when the portal is created. The PCB structure is defined in NWSNUT.H.

NOTE: Fields in the PCB structure should not be changed directly. Call NWSNUT functions for creating and manipulating portals.

See Also

NWSCreatePortal

NWSGetScreenPalette

Returns the current screen palette
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSGetScreenPalette (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

This function returns the current screen palette.

Remarks

To change the screen palette, call **NWSSetScreenPalette**.

See Also

NWSSetScreenPalette

NWSGetSortCharacter

Returns the weighted value used for sorting a given character
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSGetSortCharacter (
   LONG character);
```

Parameters

character

(IN) Specifies the character for which to return a value.

Return Values

This function returns the weighted value used for sorting *character* in the current OS language.

Remarks

This function is used to determine the weighted values used for sorting characters used in the current OS language.

NWSInitForm

Initializes pointers for the current form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSInitForm (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function initializes the pointers in the first FIELD structure of a new form. After it is initialized, the form is built by appending various types of fields with "append" functions (for example, **NWSAppendMenuField**). Once all of the fields have been appended to the form, it is displayed by calling **NWSEditPortalForm**.

NOTE: Fields in the FIELD structure should not be changed directly. Call NWSNUT functions for building and manipulating forms.

See Also

NWSAppendBoolField, **NWSAppendCommentField**, **NWSAppendHexField**, **NWSAppendHotSpotField**, **NWSAppendIntegerField**, **NWSAppendMenuField**, **NWSAppendStringField**, **NWSAppendToForm**, **NWSAppendToMenuField**, **NWSAppendUnsignedIntegerField**, **NWSEditPortalForm**, **NWSInitMenuField**

Example

NWSInitForm

This example is taken from NDEMO.C, which can be found in the EXAMPLES directory. The message and help files have been included.

```
#include <stdio.h>
#include <conio.h>
#include <advanced.h>
#include <process.h>
#include <time.h>
#include <nwsnut.h>
#include <string.h>
#include "ndemo.hlh"         /*  help definitions  */
#include "ndemo.mlh"         /* message definitions  */

/* Global variables in this module */
NUTInfo  *handle;

/* prototypes for functions in this module */
LONG HotSpotAction (FIELD *fp, int selectKey, int *changedField,
                    NUTInfo *handle);
int  FormMenuAction(int option, void *parameter);

void main()
{
   int    menuChoice, myInteger = 600, myHexInteger = 0x2ffc, line;
   LONG   myOtherInteger = 900;
   BYTE   myBoolean, string[200];
   MFCONTROL  *mfctl;

   /* start NWSNUT here */
   /* create a form with various types of fields*/
   NWSInitForm (handle);
   line = 0;
   NWSAppendCommentField (line, 1, "Boolean Field:", handle);
   NWSAppendBoolField (line, 25, NORMAL_FIELD, &myBoolean, NULL, handle
   line += 2;
   NWSAppendCommentField (line, 1, "Integer Field:", handle);
   NWSAppendIntegerField (line, 25, NORMAL_FIELD, &myInteger, 0, 9999,
      NULL, handle);
   line += 2;
   NWSAppendCommentField (line, 1, "String Field:", handle);
   strcpy (string, "Data String");
   NWSAppendStringField (line, 25, 30, NORMAL_FIELD, string, "A..Za..z
      handle);
   line += 2;
   NWSAppendCommentField (line, 1, "Unsigned Integer Field:", handle);
   NWSAppendUnsignedIntegerField (line, 25, NORMAL_FIELD, &myOtherInteg
```

```
                                        0, 99999, NULL, handle);
   line += 2;
   NWSAppendCommentField (line, 1, "Hex Field:", handle);
   NWSAppendHexField (line, 25, NORMAL_FIELD, &myHexInteger, 0, 99999,
                      NULL, handle);
   line += 2;
   NWSAppendCommentField (line, 1, "Comment Field:", handle);
   NWSAppendCommentField (line, 25, "A comment", handle);
   line += 2;
   NWSAppendCommentField (line, 1, "Hot Spot Field:", handle);
   NWSAppendHotSpotField (line, 25, NORMAL_FIELD,
                          "Hot Field", HotSpotAction, handle);
   mfctl = NWSInitMenuField (FORM_MENU_HEADER, 10, 40, FormMenuAction,
                             handle);
   NWSAppendToMenuField (mfctl, MENU_TEXT_ONE, 1, handle);
   NWSAppendToMenuField (mfctl, MENU_TEXT_TWO, 2, handle);
   menuChoice = 1;             /* display the text for option one */
   line += 2;
   NWSAppendCommentField (line, 1, "Menu Field:", handle);
   NWSAppendMenuField (line, 25, NORMAL_FIELD, &menuChoice, mfctl,
                       NULL, handle);

   /* if no help is desired for the form, pass F_NO_HELP as the help pa
   NWSEditPortalForm (FORM_HEADER, 11, 40, 16, 50, F_NOVERIFY, FORM_HEL
      EXIT_FORM_MSG, handle);

   /* cleanup and discard this form */
   NWSDestroyForm (handle);

   /* restore NWSNUT here */
}

int  FormMenuAction(int option, void *parameter)
{
   parameter = parameter;                      /* for the compiler */

   /*
      do anything that might be needed by the selection of a given menu
      and the value returned indicates which data item is to be
      displayed in the menu field on the form
   */
   return option;
}
LONG HotSpotAction (FIELD *fp, int selectKey, int *changedField, NUTInf
{
   selectKey = selectKey;
   fp = fp;
   changedField = changedField;
   NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
      "This is your hot spot routine", &handle->messages);
   NWSAlert (0, 0, handle, DYNAMIC_MESSAGE_ONE);
   return K_RIGHT;                     /* send us to the next field */
```

```
}
```

NWSInitializeNut

Initializes the NUTInfo structure for use with the NLM
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

long NWSInitializeNut (
   LONG      utility,
   LONG      version,
   LONG      headerType,
   LONG      compatibilityType,
   BYTE     **messageTable,
   BYTE     *helpScreens,
   int       screenID,
   LONG      resourceTag,
   NUTInfo  **handle);
```

Parameters

utility

(IN) Specifies the message identifier for the name of your NLM.

version

(IN) Specifies the message identifier for the version of your NLM.

headerType

(IN) Specifies the header size.

compatibilityLevel

(IN) Specifies the NWSNUT.NLM revision level that is compatible with your NLM. This must be the NUT_REVISION_LEVEL as defined in NWSNUT.H.

messageTable

(IN) Specifies an optional pointer to message table containing program messages for your NLM (see Remarks below).

helpScreens

(IN) Specifies an optional pointer to help information for your NLM.

screenID

(IN) Specifies the screen to use for your NLM.

resourceTag

(IN) Specifies the resource tag to use when allocating memory.

handle

(IN/OUT) Points to a pointer to a NUTInfo structure allocated by the NWSNUT NLM library.

Return Values

0	Success
Nonzero	Failure

Remarks

This function must be called before any other NWSNUT function.

The *handle* parameter receives a pointer to a NUTInfo structure that contains NWSNUT context information for your NLM. This parameter is passed to other NWSNUT functions to maintain context.

The NWSNUT NLM library resolves the *messageTable* and the *helpScreens* from the NLM load definition structure if these parameters are NULL. This method of text string resolution allows NWSNUT NLM applications to be enabled for natural language support.

The following values can be specified in the *headerType* parameter:

NO_HEADER	No header
SMALL_HEADER	1-line header
NORMAL_HEADER	2-line header
LARGE_HEADER	3-line header

The *screenID* parameter is the screen handle returned by **CreateScreen**.

The *resourceTag* parameter is obtained by calling **AllocateResourceTag**.

See Using NWSNUT Interface: Example.

See Also

NWSRestoreNut

Example

NWSInitializeNut

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

/* Global variables in this module */
NUTInfo  *handle;
LONG      NLMHandle;
LONG      allocTag;
int       CLIBScreenID;

void main()
{
   LONG   ccode;

   /* get a handle for allocating a resource tag*/
   NLMHandle = GetNLMHandle();

   /* create a screen for displaying our information*/
   CLIBScreenID = CreateScreen("NUT Demo Screen", AUTO_DESTROY_SCREEN);
   if (!CLIBScreenID)
      return;

   /* allocate a resource tag to use for memory allocations */
   allocTag = AllocateResourceTag(NLMHandle, "NUT DEMO Alloc Tag",
      AllocSignature);
   if (!allocTag)
   {
      DestroyScreen(CLIBScreenID);
      return;
   }

   /*  initialize the screen interface */
   ccode = NWSInitializeNut(UTILITY_MSG, VERSION_100, NORMAL_HEADER, 0,
           0, 0, CLIBScreenID, allocTag, &handle);
   if (ccode)
   {
      DestroyScreen(CLIBScreenID);
      return;
   }
}
```

NWSInitList

Initializes the current list pointers
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSInitList (
   NUTInfo  *handle,
   void      (*freeRoutine) (
      void  *memoryPointer));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

freeRoutine

(IN) Points to a routine to be used to free memory allocated to be used in the current list.

Return Values

None

Remarks

If you have previously built a list, you must save or destroy it before calling **NWSInitList**. **NWSInitList** initializes a new LISTPTR structure, creating an empty list. The *freeRoutine* parameter specifies the free routine to be used for freeing memory allocated by your NLM and passed to NWSNUT by the **NWSAppendToList** function in the *otherInfo* parameter.

See Also

NWSAppendToList, **NWSDeleteFromList**, **NWSDeleteFromPortalList**, **NWSDestroyList**, **NWSGetList**, **NWSInsertInList**, **NWSInsertInPortalList**, **NWSList**, **NWSModifyInPortalList**,

NWSSaveList, NWSRestoreList, NWSSetList

Example

NWSInitList

This example is taken from NDEMO.C, which can be found in the EXAMPLES directory. The message and help files have been included.

```
#include <stdio.h>
#include <conio.h>
#include <advanced.h>
#include <process.h>
#include <nwsnut.h>
#include <string.h>
#include "ndemo.hlh"      /*  help definitions  */
#include "ndemo.mlh"      /*  message definitions  */

/*  Global variables in this module*/
NUTInfo  *handle;

/*  prototypes for functions in this module */
int ListAction (LONG keyPressed, LIST **elementSelected, LONG *itemNumb
      void *listParameter);

void main()
{
   /* start NWSNUT here */
   NWSInitList (handle, Free);
   NWSAppendToList (NWSGetMessage (LIST_ITEM_1, &(handle->messages)),
      (void *) 0, handle);
   NWSAppendToList (NWSGetMessage (LIST_ITEM_2, &(handle->messages)),
      (void *) 0, handle);
   NWSAppendToList (NWSGetMessage (LIST_ITEM_3, &(handle->messages)),
      (void *) 0, handle);
   NWSAppendToList (NWSGetMessage (LIST_ITEM_4, &(handle->messages)),
      (void *) 0, handle);
   NWSList (LIST_HEADER, 10, 40, 4,
      strlen (NWSGetMessage (LIST_HEADER, &(handle->messages))) + 4,
      M_ESCAPE | M_SELECT, NULL, handle, NULL, ListAction, 0);
   /* cleanup and discard this list */
   NWSDestroyList (handle);
   /* restore NWSNUT here */
}
int ListAction (LONG keyPressed, LIST **elementSelected, LONG *itemNumb
      void *listParameter)
{
   elementSelected = elementSelected;
   listParameter = listParameter;
   if (keyPressed == M_ESCAPE)
      return 1;
```

```
   switch ((*itemNumber) + 1)
   {
      case 1:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected item number 1", &handle->messages);
         break;

      case 2:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected item number 2", &handle->messages);
         break;

      case 3:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected item number 3", &handle->messages);
         break;

      case 4:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected item number 4", &handle->messages);
         break;

      default:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected another item", &handle->messages);
         break;
   }
   NWSAlert (0, 0, handle, DYNAMIC_MESSAGE_ONE);
   return -1;
}
```

NWSInitListPtr

Initializes a list that is appended to a form
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSInitListPtr (
   LISTPTR  *listPtr);
```

Parameters

listPtr

(IN) Specifies the list pointer to be initialized.

Return Values

None

Remarks

This function initializes a list that is appended to a form.

No memory is freed by this process, so this function should not be used to reinitialize a list pointer.

NWSInitMenu

Initializes pointers for the current menu
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSInitMenu (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function initializes an MFCONTROL structure for a menu.

To build a list of options for the menu, call **NWSAppendToMenu** for each option.

See Also

NWSAppendToMenu, **NWSMenu**

Example

NWSInitMenu

This example is taken from NDEMO.C, which can be found in the EXAMPLES directory. The message and help files have been included.

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

```
#include <stdio.h>
#include <conio.h>
#include <advanced.h>
#include <process.h>
#include <nwsnut.h>
#include "ndemo.hlh"      /*  help definitions  */
#include "ndemo.mlh"      /*  message definitions  */

/*  Global variables in this module*/
NUTInfo  *handle;
/* prototypes for functions in this module  */
int  MenuAction(int option, void *parameter);

void main()
{
   /* start NWSNUT here */
   /* create a menu */
   NWSInitMenu (handle);

   /*
      build the menu items on the fly. These could come from the
      message file just as easily.
   */
   NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
      "Menu Item 1       ", &handle->messages);
   NWSSetDynamicMessage(DYNAMIC_MESSAGE_TWO,
      "Menu Item 2       ", &handle->messages);
   NWSSetDynamicMessage(DYNAMIC_MESSAGE_THREE,
      "Menu Item 3       ", &handle->messages);
   NWSAppendToMenu (DYNAMIC_MESSAGE_ONE, 1, handle);
   NWSAppendToMenu (DYNAMIC_MESSAGE_TWO, 2, handle);
   NWSAppendToMenu (DYNAMIC_MESSAGE_THREE, 3, handle);
   NWSMenu (MENU_HEADER, 10, 40, NULL, MenuAction, handle, (void *)hand
   NWSDestroyMenu (handle);
   /* restore NWSNUT here */
}

int  MenuAction(int option, void *junk)
{
   option = option;            /* keep the compiler quiet */
   junk = junk;
   switch (option)
   {
      case 1:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected item number 1", &handle->messages);
         break;

      case 2:
         NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
         "You selected item number 2", &handle->messages);
         break;
```

```
            case 3:
               NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
               "You selected item number 3", &handle->messages);
               break;

            case 4:
               NWSSetDynamicMessage(DYNAMIC_MESSAGE_ONE,
               "You selected item number 4", &handle->messages);
               break;

            default:
               return 0;
         }
         NWSAlert (0, 0, handle, DYNAMIC_MESSAGE_ONE);
         return -1;
      }
```

NWSInitMenuField

Initializes a menu field for a form
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

MFCONTROL *NWSInitMenuField (
   LONG      headermsg,
   LONG      cLine,
   LONG      cCol,
   int       (*action) (
      int    option,
      void  *parameter),
   NUTInfo  *nutInfo,
   ...);
```

Parameters

headermsg

(IN) Specifies the message identifier of the menu's header text.

cLine

(IN) Specifies the screen line to center the menu on.

cCol

(IN) Specifies the screen column to center the menu on.

action

(IN) Specifies the routine to be called when a menu item is selected.

nutInfo

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

If successful, this function returns an MFCONTROL structure for the new menu.

Remarks

This function initializes a new MFCONTROL structure for a menu that is associated with a field in a form.

The *headermsg* parameter specifies the text to appear in a menu field on the form. When this field is selected, the menu appears.

Optional parameters for the action routine can be added to the end of the parameter list.

See Also

NWSAppendMenuField, **NWSAppendToMenuField**

Example

See the example for **NWSInitForm**.

NWSInsertInList

Inserts a new list element after the specified list element
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LIST *NWSInsertInList (
   BYTE     *text,
   BYTE     *otherInfo,
   LIST     *atElement,
   NUTInfo  *handle);
```

Parameters

text

(IN) Points to the text for the new list element.

otherInfo

(IN) Points to customized data to be associated with the new list element.

atElement

(IN) Points to a list element that marks the location before the new element.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

This function returns a pointer to the new list element.

Remarks

This function behaves like **NWSAppendToList**, except that the location of the new list element is specified by the *atElement* parameter.

The *atElement* parameter is given the LIST structure that was returned

when that element was created by **NWSAppendToList**.

See Also

NWSAppendToList, **NWSInsertInPortalList**, **NWSModifyInPortalList**

NWSInsertInPortalList

Inserts a new list element at the specified list location using a specified insertion procedure

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSInsertInPortalList (
   LIST     **currentElement,
   int      *currentLine,
   int       (*InsertProcedure) (
      BYTE  *text,
      void  **otherInfo,
      void  *parameters),
   int       (*FreeProcedure) (
      void  *otherInfo),
   NUTInfo  *handle,
   void     *parameters);
```

Parameters

currentElement

(IN) Points to the current list element. The new list element is inserted after it.

currentLine

(IN) Points to the current line within the list.

InsertProcedure

(IN) Points to a custom insertion procedure for inserting the new list element.

FreeProcedure

(IN) Points to a procedure to be used for freeing memory allocated by *InsertProcedure*.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Successful.
Nonzero	The value returned from the insert procedure when unsuccessful.

Remarks

This function inserts a new list element in a list currently displayed in a portal.

The *InsertProcedure* parameter is a custom routine written by the developer. *InsertProcedure* is passed the following parameters:

text	A 256-byte buffer which contains the text for the new element.
otherInfo	A pointer to a character pointer. This parameter shoud be set to NULL if the *otherInfo* field for the LIST structure of the new element is NULL. Otherwise, it should point to memory allocated to *otherInfo*.
parameters	Additional parameters for the insert procedure.

The *InsertProcedure* should return 0 if it successfully gets the *text* and *otherInfo* fields for the new element.

The *FreeProcedure* specifies a procedure to be used to free memory allocated to the list element. This function receives the same pointer to *otherInfo* as the *InsertProcedure*.

After insertion, the new element is realigned in the display area so that it is the currently highlighted element.

See Also

NWSAppendToList, **NWSInsertInList**, **NWSModifyInPortalList**

NWSIsAnyMarked

Returns a boolean value that indicates whether any elements of the current list are marked

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSIsAnyMarked (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Nothing is marked
1	One or more elements are marked

Remarks

This function returns the marking status of the current list. If **NWSIsAnyMarked** returns 1, 1 or more elements in the current list is marked.

See Also

NWSPopMarks, **NWSPushMarks**, **NWSUnmarkList**

NWSIsdigit

Returns a boolean value indicating whether the specified character is an ASCII number representation

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSIsdigit (
   BYTE   character);
```

Parameters

character

(IN) Specifies the ASCII value of the byte in question.

Return Values

1	*character* is a digit.
0	*character* is not a digit.

Remarks

This function returns 1 if *character* is an ASCII representation of a decimal digit.

See Also

NWSIsxdigit

NWSIsxdigit

Returns a boolean value indicating whether the specified character is an ASCII representation of a hexadecimal digit

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSIsxdigit (
   BYTE   character);
```

Parameters

character

(IN) Specifies the BYTE in question.

Return Values

1	*character* is a HEX digit (that is, `0'-`9' or `A'-`F')
0	*character* is not a HEX digit

Remarks

This function returns 1 if *character* is an ASCII representation of a hexadecimal digit.

See Also

NWSIsdigit

NWSKeyStatus

Returns a boolean value indicating whether a key is waiting in the NWSNUT screen keyboard buffer

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSKeyStatus (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

1	A key is waiting
0	No key is waiting

Remarks

This function returns 1 if a key is waiting in the keyboard buffer.

NWSList

Displays the current list and allows the user to mark, select and perform other LIST options on the current list

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSList (
   LONG      header,
   LONG      centerLine,
   LONG      centerColumn,
   LONG      height,
   LONG      width,
   LONG      validKeyFlags,
   LIST     **element,
   NUTInfo  *handle,
   LONG      (*format) (
      LIST     *element,
      LONG      skew,
      BYTE     *displayLine,
      LONG      width),
   int       (*action) (
      LONG      keyPressed,
      LIST     **elementSelected,
      LONG     *itemLineNumber,
      void     *actionParameter),
   void     *actionParameter);
```

Parameters

header

(IN) Specifies the message identifier for the header text.

centerLine

(IN) Specifies the row to center the portal on.

centerColumn

(IN) Specifies the column to center the portal on.

height

(IN) Specifies the height of the new portal.

width

(IN) Specifies the width of the new portal.

validKeyFlags

(IN) Specifies the action key mask showing valid action keys as defined in NWSNUT.H.

element

(IN/OUT) Specifies the element to highlight as default.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

format

(IN) Specifies an optional routine to format list elements.

action

(IN) Specifies an optional routine to be called after key is pressed.

actionParm

(IN) Specifies an optional parameter for the action routine.

Return Values

If an error occurs, M_ESCAPE is returned.

If *action* is not NULL, the value returned by the action procedure is returned.

If *action* is NULL, and no error occurs, the key value is returned indicating which key the user pressed, and *element* points at the item the user selected.

Remarks

This function displays a list and allows the users to manipulate it. If a wait portal is displayed, it is removed. Then the **NWSList** function calculates and draws the list portal. The user is then allowed to manipulate the list by scanning, inserting, deleting, modifying, or selecting. If the action routine returns -1, **NWSList** allows another choice. Otherwise, **NWSList** passes the value returned by the action routine to the calling procedure.

The *validKeyflags* parameter defines which action keys are valid for the list. The bits in the *validKeyFlags* parameter are defined in NWSNUT.H as follows:

M_ESCAPE	Escape key enabled.

M_INSERT	Insert key enabled.
M_DELETE	Delete key enabled.
M_MODIFY	Modify key enabled.
M_SELECT	Select key (Enter) enabled.
M_MDELETE	Delete key enabled for marked items.
M_CYCLE	Tab enabled.
M_MMODIFY	Modify key enabled for marked items.
M_MSELECT	Select key (Enter) enabled for marked items.
M_NO_SORT	Do not sort list.

Combinations of keys can be made ORed together to make them active.

The *element* parameter specifies the list element to highlight by default. If this parameter is NULL, the first element in the list is highlighted.

The *format* parameter specifies an optional routine for formatting list items. If no special formatting is required, *format* can be NULL. The following parameters are passed to the format routine:

element	The element to format.
skew	The horizontal skew value desired for the formatted elements.
displayLine	The line number in the portal where the element is displayed.
width	The width desired for the formatted elements.

The *action* parameter specifies an optional action routine. If no action routine is desired, *action* can be NULL. The following parameters are passed to the action routine:

keyPressed	The key pressed to initiate the action routine.
elementSelected	The element highlighted.
itemLineNumber	The line number of the highlighted element.

actionParameter	The same parameter that was passed to List.

The *actionParameter* parameter is an additional optional parameter to be passed to the action routine.

See Also

NWSAppendToList, **NWSDestroyList**, **NWSInitList**

Example

See the example for **NWSInitList**.

NWSMemmove

Copies bytes from one buffer to another buffer
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSMemmove (
   void  *dest,
   void  *source,
   int    len);
```

Parameters

dest

(OUT) Points to the destination address.

source

(IN) Points to the source address.

len

(IN) Specifies the number of bytes to move.

Return Values

None

Remarks

The two buffers can overlap.

NWSMenu

Allows the user to choose from the options in the current menu
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSMenu (
   LONG      header,
   LONG      centerLine,
   LONG      centerColumn,
   LIST     *defaultElement,
   int       (*action) (
      int       option,
      void     *parameter),
   NUTInfo  *handle,
   void     *actionParameter);
```

Parameters

header

(IN) Specifies the message identifier for header text.

centerLine

(IN) Specifies the center row for the menu portal.

centerColumn

(IN) Specifies the center column for the menu portal.

defaultElement

(IN/OUT) Specifies the element to highlight as the default selection.

action

(IN) Specifies an optional action procedure to be called when an item is selected.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

actionParameter

(IN) Specifies an optional parameter for the action procedure.

Return Values

If an error occurs -2 is returned.

If *action* is not NULL, then the value returned by the action procedure is returned.

If *action* is NULL, and no error occurs, either M_ESCAPE or the value assigned to the selected menu option (by **NWSAppendToMenu**) is returned indicating which key the user pressed, and *defaultElement* points at the item the user selected.

If the user presses the Escape key, -1 is returned.

Remarks

The *defaultElement* parameterspecifies which element is highlighted by default. If *defaultElement* is NULL, the first element in the menu is highlighted.

The *action* parameter specifies the action routine to be called when an item is selected. This routine is passed the following parameters:

option	The value assigned the menu option by **NWSAppendToMenu**.
parameter	The same parameter passes to **NWSMenu** as *actionParameter*..

An additional parameter can be passed to the action routine through the *actionParameter* parameter.

See Also

NWSAppendToMenu, **NWSDestroyMenu**, **NWSInitMenu**

Example

See the example for "**NWSInitMenu**".

NWSModifyInPortalList

Modifies the text field of an element
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSModifyInPortalList (
   LIST     **currentElement,
   int      *currentLine,
   int       (*ModifyProcedure) (
      BYTE     *text,
      void     *parameters),
   NUTInfo  *handle,
   void     *parameters);
```

Parameters

currentElement

(IN) Points to the highlighted element in the list.

currentLine

(IN) Points to the line of the current element.

ModifyProcedure

(IN) Points to the procedure to be used to modify the list.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Successful.
-1	The function failed to create the new element.
Other	The value returned by the modify procedure is returned.

Remarks

This function modifies the text field of a list element. Since the text size of a list element is fixed at the time the list element is created, a new element is created and the old element is deleted.

The *ModifyProcedure* modifies the text for the list element. This procedure is passed the following parameters:

text	A buffer with a copy of the current text for the list element.
parameters	Same parameter that was passed to **NWSModifyInPortalList**

The modify procedure should return 0 if the text is successfully modified. This causes **NWSModifyInPortalList** to create a new list element and distroy the old list element.

See Also

NWSAppendToList, **NWSInsertInList**, **NWSInsertInPortalList**

NWSPopHelpContext

Removes the last element from the help stack
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSPopHelpContext (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Success
Nonzero	Failure

Remarks

This function removes the last help context from the help stack.

See Also

NWSDisplayHelpScreen, **NWSPushHelpContext**

Example

NWSPopHelpContext

```
#include <nwsnut.h>
```

```
#include "myNLM.HLH"       /* includes definition for MY_FIRST_HELP*/

   NWSPushHelpContext (MY_FIRST_HELP, handle);
   /*
      from this point on, whenever the user presses F1, the
      help screen identified by MY_FIRST_HELP is displayed
   */
   NWSPopHelpContext (handle);
   /* the previous help is now in force. */
```

NWSPopList

Pops a set of list pointers from the list stack and makes them current
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSPopList (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	No more lists to pop.
1	List was popped.

Remarks

This function pops the next set of list pointers from the list stack and uses those pointers for the context of the current list.

See Also

NWSPushList, **NWSRestoreList**, **NWSSaveList**

NWSPopMarks

Pops the marked/unmarked status of all elements of the current list
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSPopMarks (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

Marks can be pushed a total of 31 times, or to a level 31 deep. However, no checking is done to determine the current level. It is possible to issue an unlimited number of pushes, but only the last 31 are retained. Therefore, it is possible to push the mark status of the list off the end of the "stack" and lose it. If the marked status of list elements has been lost, they is considered unmarked.

See Also

NWSPushMarks

NWSPositionCursor

Sets the position of the cursor relative to the entire screen.
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSPositionCursor (
   LONG                    line,
   LONG                    column,
   struct ScreenStruct   *screenID);
```

Parameters

line

(IN) Specifies the vertical position of the cursor.

column

(IN) Specifies the horizontal position of the cursor.

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None

Remarks

Values for *line* may be 0..24 to position the cursor on the screen; 25 hides the cursor.

Values for column may be from 0 to 79.

For the *screenID* parameter, use the*screenID* field in NUTInfo, as illustrated in the example following these remarks.

NWSPositionCursor can work in harmony with any of the related functions listed in the following "See Also" list.

See Also

NWSEnablePortalCursor, **NWSDisablePortalCursor**, **NWSPositionPortalCursor**

Example

NWSPositionCursor

```
ccode = NWSPositionCursor(12, 40, handle->screenID);
```

NWSPositionPortalCursor

Positions the cursor for a NWSNUT portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSPositionPortalCursor (
   LONG   line,
   LONG   column,
   PCB   *portal);
```

Parameters

line

(IN) Portal line where cursor is to be positioned.

column

(IN) Portal column where cursor is to be positioned.

portal

(IN) Points to a NWSNUT portal control block.

Return Values

None

Remarks

The *portal* parameter can be obtained by calling **NWSGetPCB**.

NWSPromptForPassword

Enables a console operator to input a password to an NLM, with optional forced verification

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSPromptForPassword (
   LONG      passwordHeader,
   LONG      centerLine,
   LONG      centerColumn,
   LONG      maxPasswordLen,
   BYTE     *passwordString,
   LONG      verifyPassword,
   NUTInfo  *handle);
```

Parameters

passwordHeader

(IN) Specifies the header string for the password box.

centerLine

(IN) Specifies the vertical center of the password portal.

centerColumn

(IN) Specifies the horizontal center of the password portal.

maxPasswordLen

(IN) Designated the maximum length of the password string.

passwordString

(IN) Points to a null terminated string to be used as the password.

verifyPassword

(IN) Enables or disables password verification.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

1	E_ESCAPE (User pressed <Escape>; password string is empty.)
2	E_SELECT (User pressed <Enter>; string contains valid password.)
0xFFFFFFFE	Unable to allocate memory for PCB, virtual screen, or save area.
0xFFFFFFFF	Maximum number of portals already defined.

Remarks

NWSPromptForPassword creates a password portal with a specifiable header. It also allows for optional password verification. The preset prompt for password entry is "Type the password:".

The string designated by *passwordHeader* displays a maximum of 40 characters in the password portal.

The *maxPasswordLen* parameter specifies the number of characters the function will accept for the password. Make sure the buffer is at least one byte longer than this value to accommodate the null byte.

The *centerLine* and *centerColumn* parameters specify the screen location of the password portal. 0, 0 centers the portal on the screen, excluding the NLM header. Other values passed to *centerLine* and *centerColumn* locate the center of the password portal relative to the top-most line below the NLM header and the left-most column of the screen. However, since these values designate the portal center, take into account the dimensions of the password portal itself.

Passing a zero to *verifyPassword* keeps the function from displaying a confirmation portal. Nonzero enables confirmation. The header for the confirmation portal "Retype the password for verification" and the prompt "Retype the password" are both preset.

See Also

NWSAppendPasswordField

NWSPushHelpContext

Saves the help context onto the help stack, making it the current help context (it is displayed when <F1> is pressed

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSPushHelpContext (
   LONG      helpContext,
   NUTInfo  *handle);
```

Parameters

helpContext

(IN) Specifies the help context.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Success.
Nonzero	Unable to push help context.

Remarks

This function makes *helpContext* the current help context and saves it onto the help stack. If the user presses <F1>, the help message associated with *helpContext* is displayed.

The help context is a help message identifier that is associated with a help message. The help context (help identifiers) is stored in an .HLH file and the messages are stored in an .HLP file.

Up to MAXHELP help contexts can be saved to the help stack.

See Also

NWSDisplayHelpScreen, NWSPopHelpContext

Example

NWSPushHelpContext

```
#include <nwsnut.h>
#include "myNLM.HLH"       /* includes definition for MY_FIRST_HELP*/

   NWSPushHelpContext (MY_FIRST_HELP, handle);
   /*
      from this point on, whenever the user presses F1, the
      help screen identified by MY_FIRST_HELP is displayed
   */
   NWSPopHelpContext (handle);
   /* the previous help is now in force. */
```

NWSPushList

Pushes the current list pointers onto the list stack
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSPushList (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	No room on list stack.
1	List was pushed.

Remarks

This function saves the current list to the list stack. Up to MAXLISTS lists can be saved.

See Also

NWSPopList, **NWSRestoreList**, **NWSSaveList**

NWSPushMarks

Pushes the marked/unmarked status of all elements of the current list
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSPushMarks (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

Marks can be pushed a total of 31 times, or to a level 31 deep. However, no checking is done to determine the current level. It is possible to issue an unlimited number of pushes, but only the last 31 are retained. Therefore, it is possible to push the mark status of the list off the end of the "stack" and lose it.

See Also

NWSPopMarks

NWSRemovePreHelp

Removes the current pre help portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSRemovePreHelp (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function reverses the effect of **NWSDisplayPreHelp**.

See Also

NWSDisplayPreHelp

NWSRestoreDisplay

Identical to **NWSInitDisplay** except that it does not return anything. The function merely clears the screen.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSRestoreDisplay (
   struct ScreenStruct  *screenID);
```

Parameters

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

Non

NWSRestoreList

Makes the specified list the current list
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSRestoreList (
   LONG      listIndex,
   NUTInfo  *handle);
```

Parameters

listIndex

(IN) Specifies the index into the save stack of the list to be made current.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Index out of range
1	Successful

Remarks

This function makes the list specified by *listIndex* the current list. The *listIndex* parameter is the number assigned to the list when **NWSSaveList** was called.

See Also

NWSPopList, **NWSPushList**, **NWSSaveList**

NWSRestoreNut

Cleans up any resources allocated by the NWSNUT library on behalf of the calling client NLM

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSRestoreNut (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function should be called whenever an NLM that has been using NWSNUT is unloaded or whenever the NLM is finished using NWSNUT. This allows NWSNUT to release any memory resources it has allocated on behalf of the client NLM, and take care of any other cleanup.

See Also

NWSInitializeNut

NWSRestoreZone

Restores data in a buffer to the screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSRestoreZone (
   LONG                  line,
   LONG                  column,
   LONG                  height,
   LONG                  width,
   BYTE                 *buffer,
   struct ScreenStruct  *screenID);
```

Parameters

line

(IN) Specifies the top-most line of the zone, relative to the entire screen (range 0 to 24).

column

(IN) Specifies the left-most column of the zone relative to the entire screen (range 0 to 79).

height

(IN) Specifies the height of the zone in lines.

width

(IN) Specifies the width of the zone in columns.

buffer

(IN) Points to the buffer from which the specified screen area is to be filled.

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None

Remarks

NWSRestoreZone restores the data from a buffer to the screen. Data can be saved to the buffer by using **NWSSaveZone**.

The size of the buffer to which buffer points should be the zone's height times width times two.

For the *screenID* parameter, use the*screenID* field in NUTInfo, as illustrated in **NWSPositionCursor**.

See Also

NWSSaveZone, **NWSScrollZone**

NWSSaveFunctionKeyList

Saves the current list of enabled function key
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSaveFunctionKeyList (
   BYTE     *keyList,
   NUTInfo  *handle);
```

Parameters

keyList

(IN) Points to a byte array in which to store the function key list.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function saves saves the enabled function keys into *keyList*.

See Also

NWSDisableFunctionKey, **NWSEnableFunctionKey**, **NWSEnableFunctionKeyList**

NWSSaveInterruptList

Saves the context of the current interrupt keys
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSaveInterruptList (
   INTERRUPT  *interruptList,
   NUTInfo    *handle);
```

Parameters

interruptList

(IN/OUT) Points to the first element in an array of pointers to INTERRUPT structures (see "Remarks" below).

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

The *interruptList* parameter points to an array of pointers to INTERRUPT structures. The array should be of size MAXFUNCTINOS+1, where MAXFUNCTIONS is a value defined in NWSNUT.H.
NWSSaveInterruptList saves information about all currently enabled interrupt keys into this buffer.

See Also

NWSDisableAllInterruptKeys, **NWSEnableInterruptKey**, **NWSEnableInterruptList**

NWSSaveList

Saves the current list into the specified slot in the save stack
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSSaveList (
   LONG      listIndex,
   NUTInfo  *handle);
```

Parameters

listIndex

(IN) Specifies the index into the save stack.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	The *listIndex* parameter is out of range.
1	Successful.

Remarks

This function saves the list into the specified slot in the *saveStack* field of the NUTInfo structure. The *listIndex* parameter specifies the slot.

NOTE: Take care that you do not save another list to the same slot, because the first list is overwritten.

See Also

NWSPopList, **NWSPushList**, **NWSRestoreList**

NWSSaveZone

Saves a defined area on the screen to a buffer.
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSaveZone (
   LONG                  line,
   LONG                  column,
   LONG                  height,
   LONG                  width,
   BYTE                 *buffer,
   struct ScreenStruct  *screenID);
```

Parameters

line

(IN) Specifies the top-most line of the zone, relative to the entire screen (range of 0 to 24).

column

(IN) Specifies the left-most column of the zone, relative to the entire screen (range of 0 to 79).

height

(IN) Specifies the height of the zone in lines.

width

(IN) Specifies the width of the zone in columns.

buffer

(IN) Points to the buffer in which the specified screen area is to be saved.

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None

Remarks

NWSSaveZone saves an defined area of the screen to a buffer. The data in the buffer can be restored to the screen using **NWSRestoreZone**.

The buffer to which *buffer* points should be of the size height times width times two, relative to the zone to be saved.

For the *screenID* parameter, use the*screenID* field in NUTInfo, as illustrated in **NWSPositionCursor**.

See Also

NWSRestoreZone, **NWSScrollZone**

NWSScreenSize

Returns the maximum number of display lines and columns available on the server screens

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSScreenSize (
   LONG  *maxLines,
   LONG  *maxColumns);
```

Parameters

maxLines

(OUT) Points to a LONG variable that receives the maximum number of lines for the server screen.

maxColumns

(OUT) Points to a LONG variable that receives the maximum number of columns for the server screen.

Return Values

None

Remarks

This function can be used to determine the maximum portal size that can be displayed.

NWSScrollPortalZone

Scrolls a region in a NWSNUT portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSScrollPortalZone (
   LONG   line,
   LONG   column,
   LONG   height,
   LONG   width,
   LONG   attribute,
   LONG   count,
   LONG   direction,
   PCB   *portal);
```

Parameters

line

(IN) Specifies the top-most row of the portal to scroll.

column

(IN) Specifies the left-most column of the portal to scroll.

height

(IN) Specifies the height of the scroll region.

width

(IN) Specifies the width of the scroll region.

attribute

(IN) Specifies the attribute to fill in vacated region.

count

(IN) Specifies the number of lines to scroll.

direction

(IN) Specifies the scroll direction (V_UP or V_DOWN).

portal

(IN) Points the portal control block of the portal to scroll.

Return Values

None

Remarks

This function scrolls the display area of a portal up or down the number of lines specified by *count*. If the *directFlag* field of the *portal* parameter is DIRECT, the real screen is scrolled. If the *directFlag* field is VIRTUAL, the virtual display area of the portal is scrolled. To copy the virtual screen to the physical screen, use **NWSUpdatePortal**.

IMPORTANT: As a virtual screen is scrolled, the previous data in the virtual screen is overwritten.

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

NWSScrollZone

Allows a console operator to scroll the contents of a defined zone in a screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSScrollZone (
   LONG                  line,
   LONG                  column,
   LONG                  height,
   LONG                  width,
   LONG                  attribute,
   LONG                  count,
   LONG                  direction,
   struct ScreenStruct  *screenID);
```

Parameters

line

(IN) Specifies the top-most line of the zone.

column

(IN) Specifies the left-most column of the zone.

height

(IN) Specifies the height of the zone.

width

(IN) Specifies the width of the zone in columns.

attribute

(IN) Specifies the screen attribute for the new line(s) resulting from the scroll.

count

(IN) Specifies the number of lines to scroll.

direction

(IN) Specifies the scroll direction (V_UP or V_DOWN).

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None

Remarks

NWSScrollZone scrolls the contents of a screen zone. That is, it overwrites the lines of a defined screen area in a designated direction (up or down) with the lines from the direction of the scroll, one or more lines at a time. It also thus provides blank lines at the top or bottom of the zone.

The *count* parameter specifies the number of lines to be cleared, and the *direction* parameter specifies whether the blank lines will appear at the top or the bottom of the zone. (V_UP puts the blanks at the bottom, V_DOWN at the top.)

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

For the *screenID* parameter, use the*screenID* field in NUTInfo, as illustrated in **NWSPositionCursor**.

See Also

NWSSaveZone, **NWSRestoreZone**

NWSSelectPortal

Selects a portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSelectPortal (
   LONG      portalNumber,
   NUTInfo  *handle);
```

Parameters

portalNumber

(IN) Specifies the portal index of the new active portal.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function selects a portal, making it the active portal, bringing it to the front, and highlighting its border.

The *portalNumber* parameter is the portal index number returned by **NWSCreatePortal**.

See Also

NWSCreatePortal, **NWSDeselectPortal**

NWSSetDefaultCompare

Specifies a routine for sorting list elements
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetDefaultCompare (
   NUTInfo  *handle,
   int       (*defaultCompareFunction) (
      LIST     *el1,
      LIST     *el2));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

defaultCompareFunction

(IN) Points to the new compare routine.

Return Values

None

Remarks

This function is used to specify a custom compare routine to be used for comparing list elements. This function is stored in the *defaultCompareFunction* field of the NUTInfo structure. To obtain the current compare function, call **NWSGetDefaultCompare**.

The default compare function is passed the following parameters:

el1	The list element to compare with *el2*.
el2	The list element to compare with *el1*.

The return values for the default compare function should be:

-1	If *el1* < *el2*
0	If *el1* = *el2*
1	If *el1* > *el2*

See Also

NWSGetDefaultCompare

NWSSetDynamicMessage

Stores dynamic messages in the NWSNUT interface message table
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetDynamicMessage (
   LONG          message,
   BYTE         *text,
   MessageInfo  *messages);
```

Parameters

message

(IN) Specifies the message (DYNAMIC_MESSAGE_ONE through DYNAMIC_MESSAGE_FOURTEEN).

text

(IN) Points to the actual message.

messages

(IN) Points to the MessageInfo structure for the program.

Return Values

None

Remarks

This function does not copy the message text, but sets the appropriate pointer in the MessageInfo structure to point at the text.

The dynamic message fields in this structure are initially set to NO_MESSAGE. Specify a message number from DYNAMIC_MESSAGE_ONE through DYNAMIC_MESSAGE_FOURTEEN in the *message* parameter to set the corresponding dynamic message field in this structure. The *text* parameter specifies the message text to be pointed to by the dynamic message field.

See Using NWSNUT Interface: Example.

NWSSetErrorLabelDisplayFlag

Sets the *displayErrorLabel* field of the NUTInfo structure
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetErrorLabelDisplayFlag (
   LONG      flag,
   NUTInfo  *handle);
```

Parameters

flag

(IN) Specifies whether to display NLM version information for error messages.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function sets the *displayErrorLabel* field of the NUTInfo structure to the value of *flag*. If *flag* is 1 (the default value), **NWSDisplayErrorCondition** and **NWSDisplayErrorText** display NLM version information. If *flag* is set to 0, the version information is not displayed by the error functions.

See Also

NWSDisplayErrorCondition, **NWSDisplayErrorText**

NWSSetFieldFunctionPtr

Changes functions associated with a field in a form
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetFieldFunctionPtr (
   FIELD  *fp,
   void    (*Format) (
      FIELD*,
      BYTE *   text,
      LONG),
   LONG    (*Control) (
      FIELD *,
      int,
      int *,
      NUTInfo *),
   int     (*Verify) (
      FIELD *,
      BYTE *,
      NUTInfo *),
   void    (*Release) (
      FIELD *),
   void    (*Entry) (
      FIELD *,
      void *,
      NUTInfo *),
   void    (*customDataRelease) (
      void *,
      NUTInfo *));
```

Parameters

fp

(IN) Points to the field to associate the routines with.

Format

(IN) Specifies the format routine for the field. If no routine is wanted, specify NULL.

Control

(IN) Specifies the control routine for the field. If no routine is wanted, specify NULL.

Verify

(IN) Specifies the verify routine for the field. If no routine is wanted, specify NULL.

Release

(IN) Specifies the memory release routine for the field. If no routine is wanted, specify NULL.

Entry

(IN) Specifies a routine to be called for all fields in the form. If no routine is wanted, specify NULL.

customDataRelease

(IN) Specifies a routine to be used to release memory allocated for custom data associated with the field. If no routine is wanted, specify NULL.

Return Values

None

Remarks

This function allows the developer to specify routines associated with a field. When NULL is specified for a routine, the routine retains its former value. If you want to delete a routine, assign it to 0.

The following lists the routines that can be specified and the parameters that are passed to the routines:

Routine	**Parameters**	
Format	FIELD *	The field to format.
	BYTE **text*	The text in the field.
	LONG	The length of text.
Control	FIELD	The selected field.
	int	The action key hit.
	int	Indicates whether the field is changed.
	NUTInfo	A pointer to a NUTInfo structure that contains state information allocated to the calling NLM.
Verify	FIELD	The field to verify.
	BYTE	Data to verify.

	NUTInfo	A pointer to a NUTInfo structure that contains state information allocated to the calling NLM.
Release	FIELD	The field to release.
Entry	FIELD	The selected field.
	void	Data to be passed to the fieldEntry routine.
	NUTInfo	A pointer to a NUTInfo structure that contains state information allocated to the calling NLM.
customDataRelease	FIELD	The field to release custom data for.
	NUTInfo	A pointer to a NUTInfo structure that contains state information allocated to the calling NLM.

See Also

NWSGetFieldFunctionPtr, NWSInitForm

NWSSetFormRepaintFlag

Sets a flag that causes the form to be repainted, showing changes made to the form but not yet reflected on the screen.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetFormRepaintFlag (
   LONG      value,
   NUTInfo  *handle);
```

Parameters

value

(IN) Sets the *redisplayFormFlag* field of the NUTInfo structure containing state information allocated to the calling NLM. Pass TRUE or FALSE.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

NWSSetFormRepaintFlag allows a form to be redisplayed after you have made changes such as adjusting the length of a field, changing the text in a field (for example a prompt field), and so forth. Although these changes may have been made in the form's code, such changes do not appear on the screen unless the form is explicitly repainted. Calling **NWSSetFormRepaintFlag** and setting the *value* parameter to TRUE causes the form to be redisplayed and reflects the changes.

For example, a comment field may prompt the user to enter a password, then change to thank the user after the password has been entered. **NWSSetFormRepaintFlag** enables the change on the comment field to

appear on the screen. As this example illustrates, such changes take place in an intermediate function that has gained temporary control, but that will return control to one of the functions listed in the following "See Also" section.

See Also

NWSEditForm, **NWSEditPortalForm**, **NWSEditPortalFormField**

NWSSetHandleCustomData

Sets the custom data and custom data release function that is held in the NUTInfo structure

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetHandleCustomData (
   NUTInfo  *handle,
   void     **customData,
   void      (**customDataRelease) (
      void     *theData,
      NUTInfo  *handle));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

customData

(IN) Specifies the custom data to be held in the NUTInfo structure.

customDataRelease

(IN) Specifies the custom data release function to be held in the NUTInfo structure.

Return Values

None

Remarks

This function sets the the *customData* and *customDataRelease* fields of the NUTInfo structure. This allows you to define customized data for your NWSNUT NLM. To obtain the values of *customData* and *customDataRelease*, call **NWSGetHandleCustomData**.

If the memory was not allocated, pass NULL for *customDataRelease*.

See Also

NWSGetHandleCustomData

NWSSetList

Changes the current list to point to the new list information
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetList (
   LISTPTR  *listPtr,
   NUTInfo  *handle);
```

Parameters

listPtr

(IN) Specifies the new list pointers to be made current.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function makes the list specified by *listPtr* the current list. The *listPtr* parameter can be obtained from **NWSGetList**.

See Also

NWSGetList

NWSSetListNotifyProcedure

Sets a routine to be called when the specified list entry is selected
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetListNotifyProcedure (
   LIST  *el,
   void   (*entryProcedure) (
      LIST     *element,
      LONG      displayLine,
      NUTInfo  *handle));
```

Parameters

el

(IN) Points to the list element for which to call the entry procedure.

entryProcedure

(IN) Specifies the routine to be called when *el* is selected.

Return Values

None

Remarks

This function allows the developer to specify a routine to be called when the list element *el* is selected. To obtain the current routine, call **NWSGetListNotifyProcedure**.

If NULL is specified for *entryProcedure,* the routine is deleted.

The following parameters are passed to the *entryProcedure*:

element	The selected list element.
displayLine	The display line of the selected list element.

handle	A pointer to a NUTInfo structure that contains state information allocated to the calling NLM.

See Also

NWSGetListNotifyProcedure

NWSSetListSortFunction

Specifies a custom list sorting function
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetListSortFunction (
   NUTInfo  *handle
   void      (*listSortFunction)(
      LIST     *head,
      LIST     *tail,
      NUTInfo  *handle));
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

listSortFunction

(IN) Points to a customized list sort function.

Return Values

None

Remarks

This function allows you to use a customized list sort function in place of the default list sort function. The **listSortFunction** parameters *head* and *tail* designate the first and last links in the list.

NWSNUT stores the pointer to the customized sort function inNUTInfo. *listSortFunction*.

See Also

NWSGetListSortFunction

NWSSetScreenPalette

Changes the screen palette
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSetScreenPalette (
   LONG      newPalette,
   NUTInfo  *handle);
```

Parameters

newPalette

(IN) Specifies the palette to change the screen to.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

The following bits are defined for the *palette* parameter:

NORMAL_PALETTE
INIT_PALETTE
HELP_PALETTE
ERROR_PALETTE
WARNING_PALETTE
OTHER_PALETTE

NWSShowLine

Displays a text string at a specified screen location
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSShowLine (
   LONG                  line,
   LONG                  column,
   BYTE                 *text,
   LONG                  length,
   struct ScreenStruct  *screenID);
```

Parameters

line

(IN) Specifies the line on which the text is to be shown.

column

(IN) Specifies the column at which the text is to begin.

text

(IN) Points to the text to be shown.

length

(IN) Specifies the screen display length in which the string pointed to by *text* appears.

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None

Remarks

NWSShowLine displays a line at a specified location on a specified screen. You must also specify the length of the display in which the text

line appears.

The *length* parameter specifies the length of the display area. That length may be shorter than the string to which *text* points, in which case not all of the string appears on the screen. However, *length* may also be longer than the string, in which case the string and the contents of the remaining bytes appear. That is, *length* is not null aware.

For the *screenID* parameter, use the *screenID* field in NUTInfo, as illustrated in **NWSPositionCursor**.

NWSShowLine does not alter the attribute bytes. To specify an attribute in the line to be shown, call **NWSShowLineAttribute**.

See Also

NWSShowLineAttribute, **NWSShowPortalLine**

NWSShowLineAttribute

Identical to **NWSShowLine**, but also allows screen attribute specification
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSShowLineAttribute (
   LONG                  line,
   LONG                  column,
   BYTE                 *text,
   LONG                  length,
   LONG                  attribute,
   struct ScreenStruct  *screenID);
```

Parameters

line

(IN) Specifies the line on which the text is to be shown.

column

(IN) Specifies the column at which the text is to begin.

text

(IN) Points to the text to be shown.

length

(IN) Specifies the screen display length in which the string pointed to by *text* appears.

attribute

(IN) Specifies the screen attribute for the text.

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None

Remarks

NWSShowLineAttribute is identical to **NWSShowLine**, except that **NWSShowLineAttribute** also allows for specification of the screen attribute for the string to which *text* points.

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

For the *screenID* parameter, use the *screenID* field in NUTInfo, as illustrated in **NWSPositionCursor**.

See Also

NWSShowLine

NWSShowPortalLine

Places screen output in the specified portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSShowPortalLine (
   LONG   line,
   LONG   column,
   BYTE  *text,
   LONG    length,
   PCB   *portal);
```

Parameters

line

(IN) Specifies the top-most line within the portal that the text is to occupy.

column

(IN) Specifies the left-most column within the portal that the text is to occupy.

text

(IN) Points to the text to be shown in the portal.

length

(IN) Specifies the length of the string to which*text* points.

portal

(IN) Points to a NWSNUT portal control block.

Return Values

None

Remarks

This function places a text line within a portal. To display the changes on

the screen, call **NWSUpdatePortal**.

The *portal* parameter is the portal control block returned by **NWSGetPCB**.

See Also

NWSDisplayTextInPortal, **NWSDisplayTextJustifiedInPortal**, **NWSShowPortalLineAttribute**, **NWSUpdatePortal**

NWSShowPortalLineAttribute

Places screen output with a specified screen attribute in the specified portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSShowPortalLineAttribute (
   LONG   line,
   LONG   column,
   BYTE  *text,
   LONG   attribute,
   LONG   length,
   PCB   *portal);
```

Parameters

line

(IN) Specifies the top-most line within the portal that the text is to occupy.

column

(IN) Specifies the left-most column within the portal that the text is to occupy.

text

(IN) Specifies the text to be shown in the portal.

attribute

(IN) Specifies the screen attribute for text display.

length

(IN) Specifies the length of *text*.

portal

(IN) Points to a NWSNUT portal control block.

Return Values

None

Remarks

This function places text with a specified screen attribute within a portal. To display the changes on the screen, call **NWSUpdatePortal**.

The *portal* parameter is the portal control block returned by **NWSGetPCB**.

The *attribute* parameter can have the following values:

VNORMAL	Normal video
VINTENSE	Intense video
VREVERSE	Reverse video
VBLINK	Blinking, normal video
VIBLINK	Blinking, intense video
VRBLINK	Blinking, reverse video

See Also

NWSDisplayTextInPortal, **NWSDisplayTextJustifiedInPortal**, **NWSShowPortalLine**, **NWSUpdatePortal**

NWSSortList

Sorts the current list alphabetically
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSSortList (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function sorts the current list using the default compare function in the NUTInfo structure. The initial default is alphabetical. A custom sorting routine can be specified by calling **NWSSetDefaultCompare**.

When the sort is complete, the head and tail of the list have been updated to reflect the new list order.

See Also

NWSGetDefaultCompare, **NWSSetDefaultCompare**

NWSStartWait

Displays a portal with a <please wait> message on the NWSNUT screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSStartWait (
   LONG      centerLine,
   LONG      centerColumn,
   NUTInfo  *handle);
```

Parameters

centerLine

(IN) Specifies the center row of the display portal.

centerColumn

(IN) Specifies the center column of the display portal.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

To destroy this portal, call **NWSEndWait**. Keys are not disabled by this function.

See Also

NWSEndWait

NWSStrcat

Appends a copy of one string to the end of another
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSStrcat (
   BYTE  *string,
   BYTE  *newStuff);
```

Parameters

string

(IN/OUT) Points to the string to concatenate to.

newStuff

(IN) Points to the string to append to *string*.

Return Values

The new length of the string is returned.

Remarks

The *string* parameter is modified so that it contains the string resulting from the concatenation.

NWSToupper

Returns the uppercase value of the specified byte
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

BYTE NWSToupper (
   BYTE   character);
```

Parameters

character

(IN) Specifies the byte to change to uppercase.

Return Values

An uppercase version of *character*.

Remarks

This function uses the server character tables, and therefore changes to uppercase in the language of the server. Passing either the first or second half of a DBCS character to this routine yields unpredictable results.

NWSTrace

Displays an information portal on the screen and waits for an escape key
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSTrace (
   NUTInfo  *handle,
   BYTE     *message,
   ...);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

message

(IN) Points to the message identifier of the message to display in trace portal.

Return Values

(0xFFFFFFFF)	Error
(0xFF)	*pauseFlag* != 0 and escape key was hit
(0xFE)	*pauseFlag* != 0 and F7 key was hit

If *pauseFlag* == 0, then the portal index is returned.

Remarks

This function is useful for debugging purposes.

Optional parameters can be added to the end of the parameter list as required by *message* (for example, %d or %s).

NWSUngetKey

Inserts a key into the keyboard type-ahead buffer
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

LONG NWSUngetKey (
   LONG      type,
   LONG      value,
   NUTInfo  *handle);
```

Parameters

type

(IN) Specifies the key type.

value

(IN) Specifies the key value.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

-1	Keyboard buffer is full
0	Key was inserted

Remarks

This function reverses the effect of **NWSGetKey**.

See Also

NWSGetKey, **NWSKeyStatus**

NWSUnmarkList

Removes the mark status from the current list
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSUnmarkList (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This removes the marked status from all items in a list.

See Also

NWSIsAnyMarked, **NWSPopMarks**, **NWSPushMarks**

NWSUpdatePortal

Updates the specified portal's screen display to show changes made to the portal since its creation or last update

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSUpdatePortal (
   PCB  *portal);
```

Parameters

portal

(IN) Specifies the PCB of the portal.

Return Values

None

Remarks

This function redisplays a portal to reflect changes since its creation or last update. The *portal* parameter can be obtained by calling **NWSGetPCB**.

NWSViewText

Displays text within a portal
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSViewText (
   LONG      centerLine,
   LONG      centerColumn,
   LONG      height,
   LONG      width,
   LONG      headerNumber,
   BYTE     *textBuffer,
   LONG      maxBufferLength,
   NUTInfo  *handle);
```

Parameters

centerLine

(IN) Specifies the screen line on which to center the portal.

centerColumn

(IN) Specifies the screen column on which to center the portal.

height

(IN) Specifies the height of the portal.

width

(IN) Specifies the width of the portal.

headerNumber

(IN) Specifies the message identifier for the portal header.

textBuffer

(IN) Points to the buffer containing the text to display.

maxBufferLength

(IN) Specifies the maximum length of the text.

handle

(IN) Points to a NUTInfo structure that contains state information

allocated to the calling NLM.

Return Values

This function returns zero if successful.

Remarks

This function displays *textBuffer* in a portal. The user can position the cursor, but cannot edit the text.

See Also

NWSDisplayInformation, **NWSDisplayInformationInPortal**, **NWSDisplayTextInPortal**, **NWSDisplayTextJustifiedInPortal**, **NWSEditText**

NWSViewTextWithScrollBars

Identical to **NWSEditTextWithScrollBars**, except that the displayed text can only be read, not edited.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

Platform: NLM

Service: NWSNUT

SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSViewTextWithScrollBars (
   LONG      centerLine,
   LONG      centerColumn,
   LONG      height,
   LONG      width,
   LONG      headerNumber,
   BYTE     *textBuffer,
   LONG      maxBufferLength,
   LONG      scrollBarFlag,
   NUTInfo  *handle);
```

Parameters

centerLine

(IN) Specifies the screen line to center the portal on.

centerColumn

(IN) Specifies the screen column to center the portal on.

height

(IN) Specifies the height of the portal in lines.

width

(IN) Specifies the width of the portal in columns.

headerNumber

(IN) Specifies the message identifier for the portal header.

textBuffer

(IN) Points to the buffer containing the text to display.

maxBufferLength

(IN) Specifies the maximum length of the text.

scrollBarFlag

(IN) Specifies the presence and operation of the scroll bars.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	Successful

Remarks

NWSViewTextWithScrollBars allows a console operator to view text that is horizontally and vertically larger than the portal. The function scrolls in both directions, but it neither displays a cursor nor allows editing.

The *centerLine* and *centerColumn* parameters specify the screen location of the portal. 0, 0 centers the portal on the screen, excluding the NLM header. Other values passed to *centerLine* and *centerColumn* locate the center of the portal relative to the top-most line below the NLM header and the left-most column of the screen. However, since these values designate the portal center, they must also take into account the dimensions of the portal itself.

The *scrollBarFlag* parameter uses a combination of two sets of flags. The first set specifies which scroll bar are to appear, and the second set specifies the conditions of their appearance.

The first set contains only two flags that can be specified separately or ORed together:

SHOW_VERTICAL_SCROLL_BAR	Vertical scroll bar appears along the portal right edge.
SHOW_HORIZONTAL_SCROLL_BAR	Horizontal scroll bar appears along the portal bottom.

The second set contains two flags, only one of which can be ORed at one time with either or both of the flags in the previous set:

CONSTANT_SCROLL_BARS	Scroll bars appear constantly.
TEXT_SENSITIVE_SCROLL_BARS	Scroll bars appear only when the text exceeds the portal boundaries vertically or horizontally.

See Also

NWSDisplayInformation, **NWSDisplayInformationInPortal**, **NWSDisplayTextInPortal**, **NWSDisplayTextJustifiedInPortal**, **NWSEditText**, **NWSViewText**

NWSWaitForEscape

Waits for the escape key to be pressed
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

void NWSWaitForEscape (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

None

Remarks

This function waits for the escape key to be pressed before going to the next instruction.

See Also

NWSWaitForEscapeOrCancel

NWSWaitForEscapeOrCancel

Waits for the escape or cancel (**F7**) key to be pressed
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSWaitForEscapeOrCancel (
   NUTInfo  *handle);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

0	The escape key was pressed.
1	The cancel key was pressed.

Remarks

This function waits for the escape or cancel key to be pressed before going to the next instruction.

See Also

NWSWaitForEscape

NWSWaitForKeyAndValue

Waits until the user presses one of the keys in a specified set.
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12, 4.x
Platform: NLM
Service: NWSNUT
SMP Aware: No

Syntax

```
#include <nwsnut.h>

int NWSWaitForKeyAndValue (
   NUTInfo  *handle,
   LONG      nKeys,
   LONG      keyType[],
   LONG      keyValue[]);
```

Parameters

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

nKeys

(IN) Specifies the size of the *keyType* and *keyValue* arrays.

keyType

(IN) Specifies an array of key types defined in the NWSNUT.H file.

keyValue

(IN) Specifies an array of values for keys designated by *keyType* if *keyType* is set to K_NORMAL.

Return Values

Index into the *keyType* array or the *keyValue* array of the key pressed by the user.

Remarks

NWSWaitForKeyAndValue allows an NLM to wait until the user presses a key in a specified set. The *keyType* parameter specifies an array of key types, and the *keyValue* specifies an array associated values.

WARNING: Do not set the value of *nKeys* to be larger than the number of elements in each (not both) of the *keyType* and *keyValue* arrays, or the server will abend.

To use an ASCII defined alphanumeric character or symbol from the keyboard as the key for which your NLM waits, specify K_NORMAL as an element in the *keyType* array and the ASCII value of the desired key in a corresponding position as an element in the *keyValue* array. The following example initializes to resume operation when the user presses any lower case letter from **a** to **e**:

```
LONG count=5;/ * number of members in keyType and keyValue */
LONG keyType[]={K_NORMAL,K_NORMAL,K_NORMAL,K_NORMAL,K_NORMAL};
LONG keyValue[]={97,98,99,100,101};/ * ASCII values for a through
```

NWSNUT.H defines a number of keys (K_UP, K_DOWN, and so forth) that can also be used with **NWSWaitForKeyAndValue**. To use any of these---**excluding function keys**---specify the K_ constant as an element in the *keyType* array and zero as an element in a corresponding position in the *keyValue* array. The following example initializes to accept **Enter**, **Esc**, **Left-arrow**, or **Right-arrow**:

```
LONG count=4;
LONG keyType[]={K_SELECT,K_ESCAPE,K_LEFT,K_RIGHT};
LONG keyValue[]={0,0,0,0};
```

To use a function key, specify one of the function constants (K_F2, K_F3, K_SF4, and so on) from NWSNUT.H as an element in the *keyType* array 1 in a corresponding position as an element in the *keyValue* array. The following example initializes to accept **F2**, **F3**, or **F4**:

```
LONG count=3;
LONG keyType[]={K_F2,K_F3,K_4};
LONG keyValue[]={1,1,1};
```

NOTE: Do not specify F1 to resume because F1 is frequently used for help.

It is also possible to combine the methods illustrated above, provided the choices are aligned in their respective arrays. The following example initializes to accept **Esc**, **F3**, or lower case **e**:

```
LONG count=3;
LONG keyType[]={K_ESCAPE,K_F3,K_NORMAL};
LONG keyValue[]={0,1,101};
```

See Also

NWSWaitForEscape, **NWSWaitForEscapeOrCancel**

NWSNUT: Structures

FIELD

Defines a form field
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct fielddef {
   LIST              *element;
   LONG               fieldFlags;
   LONG               fieldLine;
   LONG               fieldColumn;
   LONG               fieldWidth;
   LONG               fieldAttribute;
   int                fieldActivateKeys;
   void               (*fieldFormat)(struct fielddef *field,
                        BYTE *text, LONG buffLen);
   LONG               (*fieldControl)(struct fielddef *field,
                        int selectKey, int *fieldChanged,
                        NUTInfo *handle);
   int                (*fieldVerify)(struct fielddef *field,
                        BYTE *data, NUTInfo *handle);
   void               (*fieldRelease)(struct fielddef *field);
   BYTE               *fieldData;
   BYTE               *fieldXtra;
   int                fieldHelp;
   struct fielddef    *fieldAbove;
   struct fielddef    *fieldBelow;
   struct fielddef    *fieldLeft;
   struct fielddef    *fieldRight;
   struct fielddef    *fieldPrev;
   struct fielddef    *fieldNext;
   void               (*fieldEntry)(struct fielddef *field,
                        void *fieldData, NUTInfo *handle);
   void               *customData;
   void               (*customDataRelease)(
                        void *fieldCustomData,
                        NUTInfo *handle);
   NUTInfo            *nutInfo;
} FIELD;
```

Fields

NOTE: Fields in this structure should not be changed directly. Use NWSNUT functions for building and manipulating forms.

element

Contains NWSNUT-internal information.

fieldFlags

Set with various "Append" functions (such as **NWSAppendBoolField**) and can have the following values:

Value	Meaning
NORMAL_FIELD	Normal, editable field.
LOCKED_FIELD	Nonaccessible field.
SECURE_FIELD	Noneditable field.
REQURED_FIELD	Verify field on form exit.
HIDDEN_FIELD	Hidden fields are not seen by the user. These fields are also locked.
PROMPT_FIELD	Prompt fields cannot be selected by the user. These fields are also locked.
ULOCKED_FIELD	Field locked by user.
FORM_DESELECT	Causes form deselection before action and verify routines are called.
NO_FORM_DESELECT	Form is not deselected before action and verify routines are called.
DEFAULT_FORMAT	Normal field-controlled justification.
RIGHT_FORMAT	Right justification.
LEFT_FORMAT	Left justification.
CENTER_FORMAT	Centered.

fieldLine

Contains the portal line on which the form field is located. This field is set when the field is appended to the form by the various "Append" functions.

fieldColumn

Contains the portal column on which the field is located. This field is set when the field is appended to the form by the various "Append" functions.

fieldWidth

Contains the maximum width of the form field. This field is set when the field is appended to the form by the various "Append" functions.

fieldAttribute

ontains the display attribute for the form field. This can be set when appending a custom field with **NWSAppendToForm**.

fieldActivateKeys

ontains the keys that activate the form field. This field can be set when

appending a custom field with **NWSAppendToForm**.

fieldFormat

Contains the routine used to format the form field.

fieldControl

Contains the routine that is called when the form field is selected.

fieldVerify

Contains the routine that is called when the form field is selected.

fieldRelease

Contains the routine called to release memory allocated for*fieldData* and *fieldExtra*.

fieldData

Contains a pointer to data associated with the form field. This pointer can be set when a custom field is appended to a form with **NWSAppendToForm**.

fieldXtra

Contains additional control information associated with the form field. This field can be set when a custom field is appended to a form with **NWSAppendToForm**.

fieldHelp

Contains the help context for the form field. You can specify the help context when the form is displayed by **NWSEditPortalForm** or **NWSEditPortalFormField**. A help context can be specified for a field when it is created by **NWSAppendToForm**.

fieldAbove

Contains NWSNUT-internal positioning information.

fieldBelow

Contains NWSNUT-internal positioning information.

fieldLeft

Contains NWSNUT-internal positioning information.

fieldRight

Contains NWSNUT-internal positioning information.

fieldPrev

Contains NWSNUT-internal positioning information.

filedNext

Contains NWSNUT-internal positioning information.

fieldEntry

Contains a routine to be called upon entry to each field in the form.

customData

Contains user-defined data to be attached to the form field.

customDataRelease

Contains a routine to release data in *customData*. The parameters match **NWSFree** so that **NWSAlloc** can be used to allocate memory for *customData*, further guaranteeing that memory is freed.

nutInfo

Contains NWSNUT context information.

Remarks

The position of the field is set by the "Append" function used to create the field. The *nutInfo*, *element*, *fieldAbove*, *fieldBelow*, *fieldLeft*, *fieldRight*, *fieldPrev*, and *fieldNext* fields are NWSNUT internal and should not be modified directly.

The following fields contain information about what routines are used for the form field:

fieldFormat

fieldControl

fieldVerify

fieldRelease

customDataRelease

fieldEntry

To obtain the routines for a field, call **NWSGetFieldFunctionPtr**; to change the routines, call **NWSSetFieldFunctionPtr**. These fields can also be set when a custom field is added to a form by **NWSAppendToForm**.

INTERRUPT

Defines an interrupt key
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct INT_ {
   void   (*interruptProc)(void *handle);
   LONG   key;
} INTERRUPT;
```

Fields

interruptProc

Points to the interupt routine that you want to use for the specified key.

key

Contains the interrupt key value.

Remarks

This structure is used by **NWSEnableInterruptList**.

LIST

Defines a list element
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct LIST_STRUCT {
   struct LIST_STRUCT   *prev;
   struct LIST_STRUCT   *next;
   void                 *otherInfo;
   LONG                 marked;
   LONG                 flags;
   void                 (*entryProcedure)(
      struct LIST_STRUCT   *listElement,
      LONG   displayLine,
      void   *NUTInfoStructure);
   LONG                 extra;
   BYTE                 text[1];
} LIST;
```

Fields

NOTE: Fields in the LIST structure should not be changed directly. Call NWSNUT functions for building and manipulating lists.

prev

Points to the previous list element in the list. If*prev* is NULL, the element is the first element in the list.

next

Points to the next list element in the list. If*next* is NULL, the element is the first element in the list.

otherInfo

Contains developer-defined data set by calling **NWSAppendToList**.

marked

Indicates whether the item has been marked for future actions. The user presses F5 to mark the current (highlighted) list item. To remove marks from all items in a list, call**NWSUnmarkList**. To determine whether any items in a list have been marked, call **NWSIsAnyMarked**.

flags

Reserved for NWSNUT.

entryProcedure

Contains a routine to call if this list item is selected. This field can be obtained by calling **NWSGetListNotifyProcedure** and set by **NWSSetListNotifyProcedure**.

extra

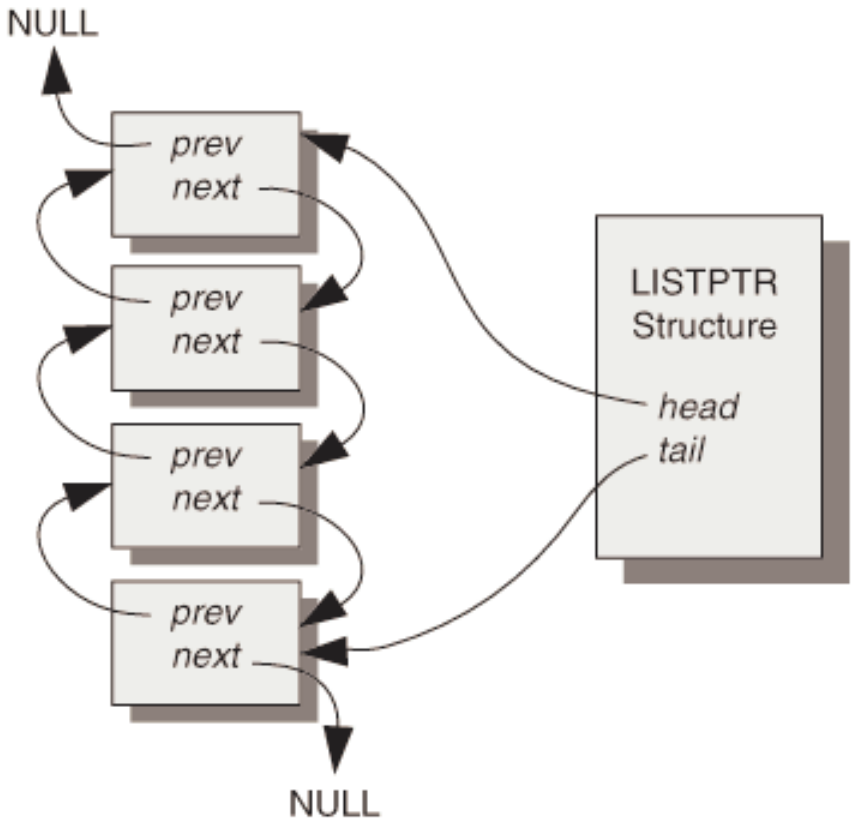
Reserved for NWSNUT.

text

Reserved for NWSNUT.

Remarks

Note that each list item points to both the previous (*prev*) and following (*next*) list items, linking the list items together. The first list item points to NULL for its previous item, whereas the last list item points to NULL for its following list item.The following illustrates the relationship of list items within a list, and their relationship to the LISTPTR structure associated with the list:



LISTPTR

Defines a list
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct LP_ {
   void   *head;
   void   *tail;
   int    (*sortProc)();
   void   (*freeProcedure)(void *memoryPointer);
} LISTPTR;
```

Fields

NOTE: Fields in this structure should not be changed directly. Use NWSNUT functions for building and manipulating lists.

head

Points to the first list element.

tail

Points to the last list element.

sortProc

Points to a procedure for sorting list items.

freeProcedure

Points to a procedure for freeing the list. Set this field with **NWSInitList**.

MessageInfo

Contains messages that can be changed dynamically as your application runs

Service: NWSNUT

Defined In: nwsnut.h

Structure

```
typedef struct MI_ {
   BYTE   *dynamicMessageOne;
   BYTE   *dynamicMessageTwo;
   BYTE   *dynamicMessageThree;
   BYTE   *dynamicMessageFour;
   BYTE   *dynamicMessageFive;
   BYTE   *dynamicMessageSix;
   BYTE   *dynamicMessageSeven;
   BYTE   *dynamicMessageEight;
   BYTE   *dynamicMessageNine;
   BYTE   *dynamicMessageTen;
   BYTE   *dynamicMessageEleven;
   BYTE   *dynamicMessageTwelve;
   BYTE   *dynamicMessageThirteen;
   BYTE   *dynamicMessageFourteen;
   LONG   messageCount;
   BYTE   **programMesgTable;
} MessageInfo;
```

Fields

dynamicMessageOne through *dynamicMessageFourteen*

Point to the dynamic messages. These messages are set by calling **NWSSetDynamicMessage**.

messageCount

programMesgTable

MFCONTROL

Defines a menu
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct MFC_ {
   LONG      headernum;
   LONG      centerLine;
   LONG      centerColumn;
   LONG      maxoptlen;
   int       (*action)(int option, void *parameter);
   LONG      arg1;
   LONG      arg2;
   LONG      arg3;
   LONG      arg4;
   LONG      arg5;
   LONG      arg6;
   LISTPTR   menuhead;
   NUTInfo   *nutInfo;
} MFCONTROL;
```

Fields

NOTE: Fields in this structure should not be changed directly. Use NWSNUT functions for building and manipulating menus.

headernum

Contains the message identifier for the menu header (set by **NWSMenu**).

centerLine

Contains the screen line the menu is centered on (set by **NWSMenu**).

centerCoumn

Contains the screen column the menu is centered on (set by **NWSMenu**).

maxoptlen

action

Contains the routine to be called when an option is selected (set by **NWSMenu**).

arg1

NWSNUT internal; cannot be set.

arg2

NWSNUT internal; cannot be set.

arg3

NWSNUT internal; cannot be set.

arg4

NWSNUT internal; cannot be set.

arg5

NWSNUT internal; cannot be set.

arg6

NWSNUT internal; cannot be set.

menuhead

Contains the list pointer structure for the menu list; cannot be set.

nutInfo

Contains NWSNUT context information; cannot be set.

NUTInfo

Contains NWSNUT context information
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct NUTInfo_ {
   PCB             *portal[MAXPORTALS];
   LONG            currentPortal;
   LONG            headerHeight;
   LONG            waitFlag;
   LISTPTR         listStack[MAXLISTS];
   LISTPTR         saveStack[SAVELISTS];
   LONG            nextAvailList;
   LIST            *head, *tail;
   int             (*defaultCompareFunction)(LIST *el1,
                     LIST *el2);
   void            (*freeProcedure)(void *memoryPointer);
   void            (*interruptTable)[MAXFUNCTIONS];
   LONG            functionKeyStatus[MAXACTIONS];
   MessageInfo     messages;
   LONG            helpContextStack[MAXHELP];
   LONG            currentPreHelpMessage;
   int             freeHelpSlot;
   LONG            redisplayFormFlag;
   LONG            preHelpPortal;
   short           helpActive;
   short           errorDisplayActive;
   LONG            helpPortal;
   LONG            waitPortal;
   LONG            errorPortal;
   void            *resourceTag;
   void            *screenID;
   BYTE            *helpScreens;
   int             helpOffset;
   LONG            helpHelp;
   void            *allocChain;
   LONG            version;
   LONG            MyNutInfoStuff[10];
   LONG            moduleHandle;
   void            *customData;
   void            (*customDataRelease)(void *theData,
                     struct NUTInfo_ *thisStructure);
   LONG            displayErrorLabel;
} NUTInfo;
```

Fields

portal

NWSNUT internal; cannot be set

currentPortal

NWSNUT internal; cannot be set

headerHeight

NWSNUT internal; cannot be set

waitFlag

NWSNUT internal; cannot be set

listStack

NWSNUT internal; cannot be set

saveStack

Points to a stack of lists saved with**NWSSaveList**. To restore a saved list, call **NWSRestoreList**.

nextAvailList

NWSNUT internal; cannot be set

head

NWSNUT internal; cannot be set

tail

NWSNUT internal; cannot be set

defaultCompareFunction

Points to a default compare function used by NWSNUT to sort lists (that is, the routine that is used by **NWSSortList**). This function can be obtained by calling **NWSGetDefaultCompare**, and set by calling **NWSSetDefaultCompare**.

freeProcedure

NWSNUT internal; cannot be set

interruptTable

Points to the interrupt procedures enabled by **NWSEnableInterruptKey**. A list of enabled interrupt keys can be saved by calling **NWSSaveInterruptList**. A list of interrupt keys can be activated by calling **NWSEnableInterruptList**.

functionKeyStatus

Indicates whether function keys are enabled. The following functions control function key status:

NWSDisableAllFunctionKeys

NWSDisableFunctionKey

NWSEnableAllFunctionKeys

NWSEnableFunctionKey

NWSEnableFunctionKeyList

NWSSaveFunctionKeyList

messages

Contains the MessageInfo structure that holds message information for your NLM. **NWSInitializeNut** sets the *programMesgTable* field of MessageInfo to the value of the *messageTable* parameter. If *messageTable* is NULL, the message table is loaded in the current NLM language of the OS. For a further information about the MessageInfo structure, see **NWSSetDynamicMessage**.

helpContextStack

Saves help context for your NLM. To save a context to the stack, call **NWSPushHelpContext**; to remove a context from the stack, call **NWSPopHelpContext**. The last help context pushed onto the help stack is displayed when **F1** is pressed.

currentPreHelpMessage

Contains the identifier set by **NWSDisplayPreHelp**.

freeHelpSlot

NWSNUT internal; cannot be set

redisplayFormFlag

NWSNUT internal; cannot be set

preHelpPortal

NWSNUT internal; cannot be set

helpActive

NWSNUT internal; cannot be set

errorDisplayActive

NWSNUT internal; cannot be set

helpPortal

NWSNUT internal; cannot be set

waitPortal

NWSNUT internal; cannot be set

errorPortal

NWSNUT internal; cannot be set

resourceTag

Set by **NWSInitializeNut** to the value of the *resourceTag* parameter.

screenID

NWSNUT internal; cannot be set

helpScreens

Points to the help file for your NLM. It is set by **NWSInitializeNut** to the value of the *helpScreens* parameter. If *helpScreens* is NULL, the help file is loaded in the current NLM language of the OS.

helpOffset

NWSNUT internal; cannot be set

helpHelp

NWSNUT internal; cannot be set

allocChain

NWSNUT internal; cannot be set

version

Set by **NWSInitializeNut** to CURRENT_NUT_VERSION, as defined in NWSNUT.H.

MyNutInfoStuff

NWSNUT internal; cannot be set

NWSNUT internal; cannot be set

moduleHandle

NWSNUT internal; cannot be set

customData

Contains custom data to be passed within theNUTInfo structure. To obtain this field, call **NWSGetHandleCustomData**; to set it, call **NWSSetHandleCustomData**.

customDataRelease

Points to the function used to release memory allocated to*customData*. To obtain this field, call **NWSGetHandleCustomData**; to set it, call **NWSSetHandleCustomData**.

displayErrorLabel

Indicates whether **NWSDisplayErrorCondition** and **NWSDisplayErrorText** displays NLM version information. The default value for this field is 1, causing version information to be displayed. To hide version information, call **NWSSetErrorLabelDisplayFlag**. This function can be called again to reset the flag to its original value.

PCB

Defines a portal
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct PCB_ {
   LONG   frameLine;
   LONG   frameColumn;
   LONG   frameHeight;
   LONG   frameWidth;
   LONG   virtualHeight;
   LONG   virtualWidth;
   LONG   cursorState;
   LONG   borderType;
   LONG   borderAttribute;
   LONG   saveFlag;
   LONG   directFlag;
   LONG   headerAttribute;
   LONG   portalLine;
   LONG   portalColumn;
   LONG   portalHeight;
   LONG   portalWidth;
   LONG   virtualLine;
   LONG   virtualColumn;
   LONG   cursorLine;
   LONG   cursorColumn;
   LONG   firstUpdateFlag;
   BYTE   *headerText;
   BYTE   *headerText2;
   BYTE   *virtualScreen;
   BYTE   *saveScreen;
   void   *screenID;
   struct NUTInfo_   *nutInfo;
   LONG   sequenceNumber;
   LONG   markForReposition;
   LONG   DBCSAction;
   LONG   borderPalette;
      /* scroll bar oriented stuff */
   LONG   showScrollBars;
   LONG   lastLine;
   LONG   longestLineLen;
   LONG   verticalScroll;
   LONG   horizontalScroll;
   LONG   oldVertical;
   LONG   oldHorizontal;
} PCB;
```

Fields

NOTE: Fields in this structure should not be changed directly. Use NWSNUT functions for creating and manipulating portals.

frameLine

Contains the top-most line of the portal. Set when the portal is created.

frameColumn

Contains the left-most column of the portal. Set when the portal is created.

frameHeight

Contains the frame height in lines. Set when the portal is created.

frameWidth

Contains the frame width in columns. Set when the portal is created.

virtualHeight

Contains the virtual height of the portal in lines. Set by calling **NWSCreatePortal**.

virtualWidth

Contains the virtual width of the portal in columns. Set by calling **NWSCreatePortal**.

cursorState

Set when the cursor is enabled (see **NWSDisablePortalCursor** and **NWSEnablePortalCursor**).

borderType

Contains the type of the portal border. Set by calling **NWSCreatePortal**.

borderAttribute

Contains the attribute for the portal border. Set by calling **NWSCreatePortal**.

saveFlag

Indicates whether the screen area beneath the portal has been saved. If this flag is set to SAVE, the screen area is redisplayed when the portal is destroyed. This flag is set by **NWSCreatePortal**.

directFlag

Indicates whether to write to the physical or virtual display area of the portal. If set to DIRECT, NWSNUT writes to the physical display area (defined by *portalHeight*, *portalWidth*, *portalLine* and *portalColumn*). If set to VIRTUAL, NWSNUT writes to the virtual display area (defined by *virtualHeight*, *virtualWidth*, *virtualLine*, and *virtualColumn*). This flag is set by **NWSCreatePortal**.

headerAttribute

Contains the screen attribute for the header. Set by calling **NWSCreatePortal**.

portalLine

Contains the top-most line of the physical display area within the frame (the area that can be written to).

portalColumn

Contains the left-most column of the physical display area within the frame (the area that can be written to).

portalHeight

Contains the number of lines in the physical display area within the frame (the area that can be written to).

portalWidth

Contains the number of columns in the physical display area within the frame (the area that can be written to).

virtualLine

Contains the top-most line of the virtual portal. Set by calling **NWSCreatePortal**.

virtualColumn

Contains the left-most column of the virtual portal. Set by calling **NWSCreatePortal**.

cursorLine

Contains the line that the cursor is on. The cursor position can be specified by calling **NWSPositionPortalCursor**.

cursorColumn

Contains the column that the cursor is on. The cursor position can be specified by calling **NWSPositionPortalCursor**.

firstUpdateFlag

NWSNUT internal. Cannot be set

headerText

Points to the text for the header. Set by calling **NWSCreatePortal**.

headerText2

NWSNUT internal. Cannot be set

virtualScreen

Contains the data written to the virtual display area. This field is set by the various functions that create portals.

saveScreen

Points to the screen area that has been saved if *saveFlag* has been set.

screenID

NWSNUT internal. Cannot be set.

nutInfo

Contains NWSNUT context information.

sequenceNumber

NWSNUT internal. Cannot be set.

markForReposition

NWSNUT internal. Cannot be set.

DBCSAction

NWSNUT internal. Cannot be set.

borderPalette

Contains the palette used for the border. Set by calling **NWSCreatePortal**.

showScrollBars

NWSNUT internal. Cannot be set.

lastLine

NWSNUT internal. Cannot be set.

longestLineLen

NWSNUT internal. Cannot be set.

verticalScroll

NWSNUT internal. Cannot be set.

horizontalScroll

NWSNUT internal. Cannot be set.

oldVertical

NWSNUT internal. Cannot be set.

oldHorizontal

NWSNUT internal. Cannot be set.

PROCERROR

Associates a return value with an error message
Service: NWSNUT
Defined In: nwsnut.h

Structure

```
typedef struct PCERR_ {
   int   ccodeReturned;
   int   errorMessageNumber;
} PROCERROR;
```

Fields

ccodeReturned

Contains the completion code for the error.

errorMessageNumber

Contains the message number for the error.

Remarks

This structure is used by **NWSDisplayErrorCondition**.

Screen Handling

Screen Handling: Guides

Screen Handling: Concept Guide

In the NetWare® 3.x and 4.x environments, Screen Handling Services enable you to manage the special features of the server's logical screens. These special features include the ability to create new screens with a variety of different characteristics, manipulate screen cursors, specify which screen is to be used when screen I/O is performed, and so on. These functions supplement the device I/O functions, first-level operating system I/O functions, and second-level stream I/O functions, all of which can perform screen I/O.

General Information

Screen Types

Screen Handling Function List

Creating/Destroying Screens

Creating Screens

Screen Names

Screen Attributes

Initial Screen Attribute Settings

Changing Screen Attributes

Type-Ahead and Command History Buffers

Destroying Screens

Screen I/O

Performing Screen I/O

Additional Links

Screen Handling: Functions

Screen Handling: Concepts

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.

Parent Topic: Screen Attributes

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.

Parent Topic: Creating Screens

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

Parent Topic: Screen Attributes

Creating Screens

CreateScreen 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.

Related Topics

Screen Names

Screen Attributes

Initial Screen Attribute Settings

Changing Screen Attributes

Type-Ahead and Command History Buffers

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.

Parent Topic: Screen Attributes

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.

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.

Parent Topic: Creating Screens

Performing Screen I/O

In the NetWare 4.x 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.

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.

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.

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.

Parent Topic: Screen Attributes

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)
DONT_CHECK_CTRL_CHARS (see Control-Character Checking)
POP_UP_SCREEN (see Popup Screens)
UNCOUPLED_CURSORS (see Cursor Coupling)

Parent Topic: Creating Screens

Screen Handling Function List

CheckIfScreenDisplayed

Checks to see if the screen is active.

clrscr

Disables the cursor and clears the current screen.

ConsolePrintf

Writes a message to the System Console Screen.

CopyFromScreenMemory

Copies a rectangular region from screen memory.

CopyToScreenMemory

Copies a rectangular region into screen memory.

CreateScreen

Opens a new screen.

DestroyScreen

Closes a screen.

DisplayInputCursor

Enables the input cursor for the current screen.

DisplayScreen

Displays the specified screen.

DropPopUpScreen

Redisplays the screen that the popup screen covered.

GetCurrentScreen

Retrieves the screen handle of the current screen.

GetCursorCouplingMode

Returns whether cursor coupling is enabled or disabled for the current screen.

GetCursorShape

Returns the start and end scan line for the cursor.

GetCursorSize

Returns the maximum size of the cursor.

GetPositionOfOutputCursor

Returns the output cursor's current row and column position for the current screen.

__GetScreenID

Returns the screen ID for a screen handle.

GetScreenInfo

Returns the screen handle associated with the specified screen.

GetSizeOfScreen

Returns the number of rows and columns of the current screen.

gotoxy

Positions the output cursor on the current screen.

HideInputCursor

Disables the input cursor for the current screen.

IsColorMonitor

Determines if a color monitor is being used.

PressAnyKeyToContinue

Writes the message <Press any key to continue> to the current screen.

PressEscapeToQuit

Writes the message <Press Escape to continue> to the current screen.

ScanScreens

Returns the screen handle (a C Library structure) associated with the specified screen.

ScrollScreenRegionDown

Scrolls down a portion of the current screen (a set of contiguous rows).

ScrollScreenRegionUp

Scrolls up a portion of the current screen (a set of contiguous rows).

SetAutoScreenDestructionMode

Enables or disables autoscreen destruction for the current screen.

SetCtrlCharCheckMode

Enables or disables control-character checking for the current screen.

SetCurrentScreen

Sets the current screen.

SetCursorCouplingMode

Enables or disables input and output cursor coupling for the current screen.

SetCursorShape

Sets the shape of the cursor.

SetInputAtOutputCursorPosition

Sets the input cursor position to the output cursor position.

SetOutputAtInputCursorPosition

Sets the output cursor position to the input cursor position.

SetPositionOfInputCursor

Sets the input cursor position on the current screen.

SetScreenRegionAttribute

Sets the display adapter attribute bytes for a region of the current screen (contiguous set of rows).

wherex

Returns the input cursor horizontal position.

wherey

Returns the input cursor vertical position.

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.

Parent Topic: Creating Screens

Screen Types

Multiple screens can exist on a server running NetWare 3.x or 4.x. 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.

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.

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.

Parent Topic: Creating Screens

Screen Handling: Functions

CheckIfScreenDisplayed

Checks whether a screen is active
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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:

22	(0x16)	EBADHNDL	Bad screen handle was passed in.

Remarks

The active screen is the screen currently being displayed on the server

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, **GetCurrentScreen**

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

SMP Aware: No

Service: Screen Handling

Syntax

```
#include <nwconio.h>

void clrscr (void);
```

Return Values

None

If an error occurs, *errno* is set to:

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

Example

clrscr

```
#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 the System Console Screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

void ConsolePrintf (
   const char   *format,
   ...);
```

Parameters

format

(IN) Points to the format control string.

Return Values

None

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.

See Also

printf

CopyFromScreenMemory

Copies a rectangular region from screen memory
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

void CopyFromScreenMemory (
   int    height,
   int    width,
   BYTE   *Rect,
   int    beg_x,
   int    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) Receives 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.

Return Values

None

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 the number of columns on the screen (currently always 80).

beg_y + *height* is less than 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 less than 0 or 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

CopyToScreenMemory

Copies a rectangular region into screen memory
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

void CopyToScreenMemory (
   int    height,
   int    width,
   BYTE   *Rect,
   int    beg_x,
   int    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) Receives 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.

Return Values

None

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 the number of columns on the screen (currently always 80).

beg_y + *height* is less than 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 less than 0 or 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

CreateScreen

Creates a new screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int CreateScreen (
   char  *screenName,
   BYTE   attributes);
```

Parameters

screenName

(IN) Receives 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:

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.
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 Loading an NLM from an NLM: Example.

See Also

DestroyScreen, **DisplayScreen**, **GetCurrentScreen**, **__GetScreenID**, **GetScreenInfo**, **ScanScreens**, **SetCurrentScreen**

Example

CreateScreen

```
#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
SMP Aware: No
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

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

DisplayInputCursor

Enables the input cursor for the current screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int DisplayInputCursor (void);
```

Return Values

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

DisplayScreen

Displays the specified screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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

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, **DestroyScreen**

DropPopUpScreen

Redisplays the screen that the popup screen covered
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int DropPopUpScreen (
   int   screenHandle);
```

Parameters

screenHandle

(IN) Specifies the screen handle of the popup screen.

Return Values

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, **DisplayScreen**

GetCurrentScreen

Returns the screen handle of the current screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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:

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

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

SMP Aware: No

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:

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

Example

GetCursorCouplingMode

```
#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
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

WORD GetCursorShape (
   BYTE   *startline,
   BYTE   *endline);
```

Parameters

startline

(OUT) Receives the first cursor scan line.

endline

(OUT) Receives 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, **SetCursorShape**

Example

GetCursorShape

```
#include <nwconio.h>
WORD   scanline;
BYTE   startline;
BYTE   endline;
```

```
BYTE   endline;
scanline = GetCursorShape (&startline, & endline);
```

GetCursorSize

Returns the cursor size
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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

Example

GetCursorSize

```
#include <nwconio.h>
WORD   scanline;
BYTE   firstline;
BYTE   lastline;
```

```
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

SMP Aware: No

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int GetPositionOfOutputCursor (
   WORD   *row,
   WORD   *column);
```

Parameters

row

(OUT) Receives the row on which the cursor is positioned (first row is 0).

column

(OUT) Receives the column on which the cursor is positioned (first column is 0).

Return Values

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

gotoxy

__GetScreenID

Returns the screen ID for a screen handle
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
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 (such as **NWSInitializeNut**).

See Also

CreateScreen, **GetScreenInfo**, **ScanScreens**

GetScreenInfo

Returns the screen handle associated with the specified screen (appeared in the NetWare 3.11 patch as **MapScreenIDToHandle**)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 4.x

SMP Aware: No

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) Specifies the name of the screen. Names of nonC Library screens are also returned (for example, MONITOR.NLM's screen).

attrib

(OUT) Specifies 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.

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, **__GetScreenID**, **ScanScreens**

GetSizeOfScreen

Returns the number of rows and columns of the current screen.
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int GetSizeOfScreen (
   WORD   *height,
   WORD   *width);
```

Parameters

height

(OUT) Receives the number of rows in the screen (first column is 0)

width

(OUT) Receives the number of columns in the screen (first column is 0)

Return Values

	(0x00)	ESUCCESS

Remarks

This function returns the size of the current screen. Currently all screens are 25x80.

See Also

DisplayScreen

gotoxy

Positions the output cursor on the current screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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:

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

Example

gotoxy

```
#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
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int HideInputCursor (void);
```

Return Values

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, GetPositionOfOutputCursor, SetPositionOfInputCursor

IsColorMonitor

Determines whether a color monitor is being used
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: Yes
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.

MapScreenIDToHandle

See **GetScreenInfo**

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
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int PressAnyKeyToContinue (void);
```

Return Values

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, **PressEscapeToQuit**

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

SMP Aware: No

Service: Screen Handling

Syntax

```
#include <nwconio.h>

int PressEscapeToQuit (void);
```

Return Values

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, **PressAnyKeyToContinue**

RingTheBell

Causes the console speaker to beep
Local Servers: blocking
Remote Servers: N/A
Classification: 3.12 and above, 4.x
SMP Aware: No
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

RingTheBell

```
#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, 4.x

SMP Aware: No

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) Specifies the name of the screen.

attrib

(OUT) Specifies 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, MapScreenIDToHandle

Example

ScanScreens

```
/*
   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 sc
      }
```

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
SMP Aware: No
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

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

CopyToScreenMemory, SetScreenRegionAttribute

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
SMP Aware: No
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

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

CopyToScreenMemory, SetScreenRegionAttribute

SetAutoScreenDestructionMode

Enables or disables auto-screen destruction for the current screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE SetAutoScreenDestructionMode (
   BYTE   newMode);
```

Parameters

newMode

(IN) 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:

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
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE SetCtrlCharCheckMode (
   BYTE   newMode);
```

Parameters

newMode

(IN) 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:

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
SMP Aware: No
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

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 Loading an NLM from an NLM: Example.

See Also

CreateScreen, **DestroyScreen**, **DisplayScreen**

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
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

BYTE SetCursorCouplingMode (
   BYTE   newMode);
```

Parameters

newMode

(IN) 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:

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

SetCursorShape

Sets the cursor shape
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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

SetInputAtOutputCursorPosition

Sets the input cursor position to the output cursor position
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetInputAtOutputCursorPosition (void);
```

Return Values

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, **GetPositionOfOutputCursor**, **gotoxy**, **HideInputCursor**, **SetOutputAtInputCursorPosition**, **wherex**, **wherey**

SetOutputAtInputCursorPosition

Sets the output cursor position to the input cursor position
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Screen Handling

Syntax

```
#include <nwconio.h>

int SetOutputAtInputCursorPosition (void);
```

Return Values

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, **gotoxy**, **SetInputAtOutputCursorPosition**, **wherex**, **wherey**

SetPositionOfInputCursor

Sets the position of the input cursor on the current screen
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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

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, **HideInputCursor**

SetScreenAreaAttribute

Sets the display adapter attribute bytes for an area of the current screen
Local Servers: nonblocking
Remote Servers: N/A
Classification: 4.x
SMP Aware: No
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, **SetScreenRegionAttribute**

Example

SetScreenAreaAttribute

```
#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
SMP Aware: No
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, **SetScreenRegionAttribute**

Example

SetScreenCharacterAttribute

```
#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

SMP Aware: No

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

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

wherex

Returns the horizontal position of the input cursor
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
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:

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, **wherey**

Example

wherex

```
#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
SMP Aware: No
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:

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, **wherex**

Example

wherey

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

main()
{
```

```
    printf("%d,%d\r\n",wherex(),wherey());
    getch();
}
```

SMP

SMP: Guides

SMP: General Guide

SMP: Concepts

Current NetWare SMP Scalability

NetWare SMP is designed to scale well. All SMP functions and many of the other functions in this SDK take full advantage of multiple processors. However, the current NetWare core OS and the major associated resources such as the file system presently run only on the primary processor (Processor 0). As a result, any function that requires I/O processing currently marshals to Processor 0, then migrates back to the processor from which the function was called.

Processor-intensive functions, such as those used for mathematical computations, scale very well with NetWare SMP because their work can be divided between or among processors. On the other hand, functions that make frequent or extensive I/O calls are largely limited to Processor 0 by the core OS and closely associated resources. As a result, you might not presently see significant overall improvement in performance by making use of multiprocessing functions.

Recommendation: Writing now to NetWare SMP is a good idea, despite the limitations explained above.

Work is presently underway to develop an SMP enabled NetWare OS and central associated resources, especially the file system. When these products are released, applications written with current SMP functionality will immediately realize significant improvements in performance.

Keep in mind also that the performance of the entire system---hardware, software, and workload---has more to do with the efficiency of the underlying software than with the quantity of hardware alone. NetWare on a single processor already shows superior performance in some ways to a major competitor running on as many as four processors. When key components are fully SMP enabled, you might want your applications to take full advantage of enhanced NetWare performance. If you write to NetWare SMP now, required modifications then will be minimal, but resulting increases in perfomance will be immediate and impressive.

Support: Support for developers writing SMP applications is limited to the API functions and their functionality. Publications that address development of efficient and scalable SMP applications are available in the marketplace, but consultation on these topics is in not within the scope of Novell developer support.

Parent Topic: Introduction to SMP

General Requirements for Synchronization Objects

In shared-memory multiprocessors each processor directly addresses all memory available on the system. This uniform access requires an efficient synchronization mechanism to guarantee consistency of the shared data. The synchronization mechanism described here is based on the abstraction of a lock control structure and its associated identifier. The lock control structure, typically allocated from the OS, is a collection of data defining precise semantics of each instance within the specified lock type.

The synchronization API has been designed to provide certain basic capabilities. The user-level synchronization library includes spin locks, mutex locks (spin-dropping-to-sleep type), read/write locks, condition variables, and barriers (spin-dropping-to-sleep type). The user-level synchronization mechanisms include

Spin Locks
Mutex Locks
Recursive Mutex Locks
Read-Write Locks
Condition Variables
Barriers

Getting Started with SMP

SMP is implemented as a series of NLM applications. When loaded on a NetWare server, these NLM applications create a generic SMP environment scalable to up to 32 processors. Special features in NetWare 4.1 allow the SMP modules to be loaded and unloaded dynamically from the server console without interrupting server operations or performance. However, platforms must support dynamic interrupt distribution and recovery.

SMP modules are provided on a CD. Follow the instructions that accompany the CD to install SMP modules onto a server. You must also install the PSM.NLM supplied by your SMP hardware vendor and copy it to the DOS partition on which SERVER.EXE resides.

Loading SMP Modules onto an Operating Server

IMPORTANT: Loading must proceed in the order listed here.

Load the following required NLM applications onto an operating server running NetWare 4.1 or higher:

1. >PSM<.NLM, supplied by hardware vendor

2. SMP.NLM
3. SMP drivers---type `load smpdriver all`
4. LAN drivers

SMP.NLM contains debugger routines to help you debug applications.

Parent Topic: Introduction to SMP

Introduction to SMP

NetWare® Symmetric Multiprocessing (SMP) was developed to provide support for multithreaded memory management and concurrent use of multiple processors in an environment compatible with native NetWare.

Software developers can use the SMP environment to create NLM™ applications that are scalable to up to 32 processors. SMP provides the functionality needed to develop multiprocessor NetWare services, multiprocessor communications and management products, and multiprocessor utilities.

Related Topics

Current NetWare SMP Scalability

Getting Started with SMP

SMP Function List

Overview of SMP Modules

SMP Features That Support Existing NetWare Functionality

Marshalling

Marshalling is a term that refers to where a function is processed. If a marshalling function is called on processor 1 or higher, the thread associated with the function marshals---goes to sleep and migrates to processor 0---before the function is processed. Following processing the thread goes to sleep again and migrates back to the processor from which it was called before continuing operation. Marshalling can detract from both the speed and the scalability of a program because of the migration time. The level of detraction is dependent upon the caching conditions on processor 0.

All functions in the *NLM Specific Function Reference* are labeled for marshalling. This information is listed just below the blocking and classification headings, under the heading "**SMP Aware:**". Functions marked "No" marshal, and follow the pattern of migration described above.

Functions marked "Yes" run on the processor from which they are called.

Overview of SMP Modules

The following NLM/OS components comprise the NetWare SMP environment:

NetWare 4.1 (SMP monitoring is included in native NetWare)
SMP.NLM
PSM.NLM (Platform Specific Module; Vendor Supplied Component)
MPDRIVER.NLM
CLIB.NLM
MSM.NLM
SMPMON.NLM (Developer tool)
MONITOR.NLM

Each of these NLM applications is briefly described below.

NetWare 4.1

This is the core software for the NetWare OS.

SMP.NLM

This module contains the NetWare Symmetric MultiProcessing Kernel, SMP Application Programming Interfaces, and debugging software provided by Novell®.

PSM.NLM and MPDRIVER.NLM

The platform-specific module is a vendor-supplied component that allows NetWare SMP to abstract the SMP hardware interface. The PSM interface defines hardware support for MP interrupt hardware, processor activation and deactivation, and other platform-specific functions, such as support for multibus architectures.

The PSM module allows NetWare SMP to use a generic MPDRIVER.NLM across all SMP hardware platforms. The MPDRIVER.NLM must be loaded into NetWare as many times as there are processors in the SMP server. The easiest and most secure way to do this is to specify the `all` switch by typing `load mpdriver all`.

CLIB.NLM

This module provides the SMP Multithreaded Library support needed to run CLIB in the SMP environment. In addition, CLIB.NLM exports SMP functions for migration and locking. It also provides a stub for each SMP function. On non-SMP platforms these stubs simply return for void functions; otherwise they return EOK or NULL. When you load CLIB.NLM, the following NLM applications also load unless they are already loaded: NIT.NLM, NLMLIB.NLM, FPSM.NLM, REQUESTR.NLM, and THREADS.NLM.

MSM.NLM

This module provides SMP LAN driver support for the SMP environment.

Parent Topic: Introduction to SMP

SMP Barrier Functions

Barrier is a synchronization mechanism which guarantees that all participating threads have reached a specified point in their execution before any are allowed to proceed.

```
barrierwait(N)
  {
    count = count + 1;
    if(count == N) then
    {
      Reset count;
         Notify all threads waiting on the barrier;
    }
  else
      Have the calling thread wait at the barrier;
  }
```

An arriving thread increments the barrier variable. Upon execution of the barrier by the Nth thread, all N threads are ready to resume. Otherwise, the resulting value is less than N, the thread polls the barrier flag which is set by the last thread to reach the barrier. The second step in a barrier, notifying all threads that the barrier has been reached is called global event notification.

Thus event notification will dominate the latency of the barrier from the point where the threads wait at the barrier. If the hardware does not provide an appropriate support, the implementation can entail significant run-time overhead. This overhead increases linearly, or for the best implementation logarithmically, with the number of threads synchronizing at the barrier.

One barrier implementation may use a scheme where threads arriving at a barrier are put to sleep until the last thread arrives. This method avoids the extra bus/network traffic of polling a barrier flag, but incurs a potentially high overhead of enqueuing a thread on the barrier. Here, the cost of the barrier synchronization is mainly because of context switched for the threads that must be installed. Sleep-barriers can be used for less-synchronized computations (that is, non-uniform amount of work assigned to each thread at fan-out time).

Often, the choice of spinning or sleeping cannot be made at compile time because uncertainty in execution times of processes. For those cases, barriers will be implemented in such a way that they will appear to be blocking, however, for optimization the implementation may choose to spin for some amount of time before blocking. NetWare SMP does not support

nonblocking barriers. All SMP barriers are the blocking type.

SMP Condition Functions

A condition variable is a user-level synchronization mechanism used to communicate information between cooperating threads. It makes possible for a thread to suspend its execution while waiting for an event. For instance, the consumer in a producer-consumer algorithm might need to wait for the producer.

A condition variable is always associated with some shared variables protected by a mutex and a predicate based on those shared variables. A thread acquires the mutex (that is, enters a critical section) and evaluates the predicate to see if it should call wait on condition to suspend its execution. To avoid the familiar wakeup-waiting race with **NWSMPCondSignal** or **NWSMPCondBroadcast**, this call must automatically release the mutex (that is, end the critical section) and suspend execution of the thread.

A thread that changes the shared variables so that the predicate might be satisfied, acquires the mutex, sets the condition (changes the shared variables to satisfy the condition), releases the mutex, and calls **NWSMPCondSignal** or **NWSMPCondBroadcast**. Those two operations allow suspended threads to resume execution and re-acquire the mutex. The **NWSMPCondSignal** is used to unblock a single thread sleeping on this condition; **NWSMPCondBroadcast**, releases all threads waiting for the condition. Both these operations are no-ops if there are no threads waiting on the condition. If **NWSMPCondSignal** is used, all threads waiting on the condition variable should be waiting for satisfaction of the same predicate. If **NWSMPCondBroadcast** is always used instead, the predicate for different waiting threads need not all be the same. When a thread returns from **NWSMPCondWait** it has reacquired the mutex and is in a new critical section. It re-evaluates the predicate whether to proceed or to call **NWSMPCondWait** again. It is up to the thread waiting on the condition to decide, based on the predicate, whether to proceed. That decision is not a responsibility of the signaling thread, as it is with semaphores or other similar mechanisms. That's because the predicate may become false before a waiting thread resumes execution (some other thread may enter a critical section first and invalidate the rededicate). Therefore, return from **NWSMPCondWait** is only a hint that must be confirmed, and the waiting thread must be prepared to test the predicate and perhaps call **NWSMPCondWait** again.

SMP Features That Support Existing NetWare Functionality

Special hooks and dormant tables in NetWare 4.1 allow the SMP modules to be loaded and unloaded dynamically from the server console without interrupting server operations and performance, except with platforms that

interrupting server operations and performance, except with platforms that do not support dynamic interrupt distribution and recovery. All Intel APIC based systems support dynamic interrupt schemes, as do several other platforms.

All NLM applications run virtually unmodified within the SMP environment. For an NLM to take advantage of SMP, the NLM must be written with the new scheduling functions provided in the SMP Kernel. All of the CLIB and OS functions are still available to NLM applications.

NetWare SMP supports the following feature set, which enhances both the performance and scalability of the NetWare environment:

SMP aware LAN drivers

SMP CLIB support for NLM applications

SMP kernel and API for SMP NLM applications

SMP Direct File System for database support

In addition, NetWare SMP supports the following previously existing services:

All existing NLM applications

All existing LAN drivers

All existing disk drivers

All Novell core OS products except DOMAIN.NLM.

Parent Topic: Introduction to SMP

SMP Function List

The SMP Services provide the functions needed to work with NetWare in a multiprocessing environment. The SMP functions are summarized in the following groups:

Table auto. Thread Management Functions

Function Name	Job
NWSMPIsAvailable	Returns whether SMP is loaded and enabled on the server
NWSMPThreadToMP	Migrates thread to the MP scheduler.
NWSMPThreadToNetWare	Migrates thread to the NetWare scheduler

Table auto. User-Level Synchronization Functions

Function Name	Job
Spin Lock	
NWSMPSpinAlloc	Allocates a spin lock
NWSMPSpinLock	Locks a spin lock
NWSMPSpinTryLock	Conditions locking
NWSMPSpinUnlock	Releases a spin lock
NWSMPSpinDestroy	Destroys a spin lock
Mutex Lock	
NWSMPMutexSleep Alloc	Allocates a sleep mutex lock
NWSMPMutexLock	Locks a mutex lock
NWSMPMutexTryLoc k	Conditions locking lock
NWSMPMutexUnloc k	Unlocks a mutex lock
NWSMPMutexDestro y	Destroys a mutex lock
Recursive Mutex Lock	
NWSMPRMutexAlloc	Allocates a recursive mutex lock
NWSMPRMutexLock	Locks a recursive mutex lock
NWSMPRMutexTryL ock	Conditions locking lock
NWSMPRMutexUnlo ck	Unlocks a recursive mutex lock
NWSMPRMutexDest roy	Destroys a recursive mutex lock
NWSMPRMutexOwn er	Defines the owner of the rmutex lock
Read-Write Lock	
NWSMPRWLockAllo c	Allocates a read-write lock
NWSMPRWReadLoc k	Acquires a read lock
NWSMPRWWriteLoc k	Acquires a write lock
NWSMPRWTryRead Lock	Conditions acquiring a read lock
NWSMPRWTryWrite	Conditions acquiring a write lock

Lock	
NWSMPRWUnlock	Unlocks a read-write lock
NWSMPRWLockDestroy	Destroys a read-write lock
Condition Variable Lock	
NWSMPCondAlloc	Allocates a condition variable lock
NWSMPCondSignal	Signals a condition variable lock
NWSMPCondBroadcast	Broadcasts a condition lock
NWSMPCondWait	Waits on a condition lock
NWSMPCondDestroy	Destroys a condition variable lock
Barrier Lock	
NWSMPBarrierAlloc	Allocates a barrier lock
NWSMPBarrierWait	Blocks at a barrier lock
NWSMPBarrierDestroy	Destroys a barrier lock

The following functions provide semaphore capability and are SMP enabled:

Table auto. Local Semaphore Functions

Function Name	Job
CloseLocalSemaphore	Closes a local semaphore
ExamineLocalSemaphore	Returns the current value of a local semaphore
OpenLocalSemaphore	Allocates a local semaphore and gives the NLM access to it
SignalLocalSemaphore	Increments the semaphore value of the specified semaphore
TimedWaitOnLocalSemaphore	Waits on a local semaphore until it is signalled or the specified time-out elapses
WaitOnLocalSemaphore	Decrements the semaphore value of the specified semaphore

Parent Topic: Introduction to SMP

SMP Mutex Functions

The mutex locks described here provide the basic mutual exclusion

functionality. A mutex, or mutual exclusion lock, guarantees that no two threads of control can access a shared variable simultaneously. If the mutual exclusion lock is found unavailable, the mutex operation can either suspend the caller, or spin until the lock is free. With careful design, spinning can be useful on a multiprocessor if expected wait time is small enough. Generally, the lock duration should be small; otherwise it may indicate a design problem.

Alternatively, the mutex operation can appear to be blocking; however, the mutex spins for some amount of time before blocking. However, since the implementation details define its precise semantics the latter alternative will satisfy only the basic semantics of the mutual exclusion.

Since the precise behavior can vary between implementations, such a general specification may not always satisfy an application designer. These functions provide basic functionality on which other synchronization functions can be built, and the mutex lock described in this document is the preferred function to use as a base.

SMP Program Examples

Spin Lock Examples

The following examples illustrate typical usages of spin locks. For clarity, error processing has been omitted. The typical age of a spin lock can be illustrated by the following example.

In the best case a spin lock on a uniprocessor will waste thread's quantum slowing down the owner of the lock; in the worst case it will deadlock (monopolize) the processor.

```
#include <nwsmp.h>
   spin_t lock;    /* spin lock */

main()
  {
  lock=NWSMPSpinAlloc("name");  /* allocate spin lock */
      ...
  NWSMPSpinLock(lock);  /* lock critical region */
      ...
   /* Critical region processing. */
      ...
  NWSMPSpinUnlock(lock);  /* unlock critical region */
      ...
  NWSMPSpinDestroy(lock);  /* destroy spin lock */
  }
```

While **NWSMPSpinLock** returns with the lock acquired, the spin-waiting for the lock may not always be the right choice for an application. In some

cases an application may instead use the operations that make a single attempt to acquire the lock. This can be illustrated with **NWSMPSpinTryLock** by the following example:

```
#include <nwsmp.h>
    spin_t lock;    /* spin lock */

main()
  {
   lock=NWSMPSpinAlloc("name");   /* allocate spin lock */
      ...
    if(NWSMPSpinTryLock(lock) == 0) {  /* a single attempt lock critic
      /* Lock is acquired; Critical region processing. */
        ...
   NWSMPSpinUnlock(lock);    /* unlock critical region */
     }
   else {
      ...
      /* Lock could not be acquired; Do something else. */
      ...
      }
      ...
   NWSMPSpinDestroy(lock);  /* destroy spin lock */
 }
```

Mutex Examples

The typical uses for mutex locks are illustrated as follows. For clarity, error processing has been omitted. The typical usage of a mutex can be illustrated by the following example:

```
#include <nwsmp.h>
   mutex_t  mutex;    /* mutex */

main()
  {
   mutex=NWSMPMutexSleepAlloc("name");   /* allocate mutex */
      ...
   NWSMPMutexLock(mutex);    /* lock critical region */
    /* Critical region processing. */
   NWSMPMutexUnlock(mutex);  /* unlock critical region */
      ...
   NWSMPMutexDestroy(mutex);  /* destroy mutex  */
  }
```

Similarly, conditional locking of a mutex can be illustrated by the following example:

```
#include <nwsmp.h>
   mutex_t  mutex;    /* mutex handle  */

main()
```

```
  {

mutex=NWSMPMutexSleepAlloc("name");   /* allocate mutex */
      ...
    if(NWSMPMutexTryLock(mutex) == 0) {
      /* Mutex has been acquired */
   /* Do critical region processing. */
     NWSMPMutexUnlock(mutex);  /* unlock critical region */
   }
   else {
      /* Mutex could not be acquired. */
   /* Do something else. */
   }
     ...
    NWSMPMutexDestroy(mutex);  /* destroy mutex  */
 }
```

Read-Write Lock Examples

The typical uses for read-write locks are illustrated as follows. For clarity, error processing has been omitted. A simple sequence of events with the read-write locks can be illustrated by the following pseudo code:

```
#include <nwsmp.h>

main()
  {
   rwlock_t rwlock;    /* read-write lock  */
    /* READER-1 - allocates the lock and acquires it for reading */
   rwlock=NWSMPRWLockAlloc("name");
   NWSMPRWReadLock(rwlock);  /* acquire read lock  */
     /* reader 1 is reading */     ...
     /* READER-2 - acquires the lock for reading without blocking */
   NWSMPRWReadLock(rwlock); /* acquire read lock */
     /* both reader 1 and reader 2 are reading */
       ...
     /* WRITER - blocks trying to acquire the lock (readers hold the lo
   NWSMPRWWriteLock(rwlock);  /* attempt to acquire write lock */
     /* READER-1 - releases the lock */
   NWSMPRWUnlock(rwlock);    /* release the lock  */
     /* READER-2 - releases the lock */
   NWSMPRWUnlock(rwlock);    /* release the lock  */
     /* WRITER - acquires the lock and is allowed to continue writing *
      ...
   NWSMPRWLockDestroy(rwlock);
  }
```

Condition Example

The typical uses of condition variables are illustrated as follows. For clarity,

error processing has been omitted. The example involves a simple consumer-producer problem, where some threads put data on a queue for other threads to remove it. It is accomplished with three threads:

(1) creates a condition variable;

(2) the producer puts data on the queue Q (sets the condition) and signals the occurrence of the condition, and

(3) the consumer waits on a condition and takes data out of the queue Q. Alternatively, to wake up all threads waiting on the condition, the producer could broadcast the occurrence the condition with **NWSMPCondBroadcast**.

Note the non-determinism built into condition variables as the broadcast/signal of the condition is only a hint that the predicate might be true, therefore the thread must always recheck the predicate after being awakened on a condition.

```
#include <nwsmp.h>
   cond_t cond;    /* condition variable */
   mutex_t mutex;    /* mutex */

main()
  {
    /* Thread 1 - allocate a condition variable and a mutex */
   cond=NWSMPCondAlloc("name");
   mutex=NWSMPMutexSleepAlloc("name");   /* create mutex */
   _____
     /* Thread 2 - producer - puts data on queue Q and signals the cond
   NWSMPMutexLock(mutex);
   putdata(Q, data);  /* put data on the queue */
   NWSMPCondSignal(cond);
   NWSMPMutexUnlock(mutex));
   _____
     /* Thread 3 - consumer - waits on condition and gets data from que
   NWSMPMutexLock(mutex);
     while(queue_is_empty(Q)) {
       NWSMPCondWait(cond, mutex); /* thread suspended on condition */
     }
   getdata(Q, data);  /* remove data from the queue */
   NWSMPMutexUnlock(mutex);  /* unlock the mutex */
      . . .
   NWSMPCondDestroy(cond);
   NWSMPMutexDestroy(mutex);
  }
```

Barrier Example

The typical usage of barriers is illustrated as follows. For clarity error processing has been omitted. In this example three threads participate in the barrier. The first creates the barrier and waits for others. Once the last thread reaches the barrier all of them can resume execution.

```
#include <nwsmp.h>
```

```
    barrier_t barrier;    /* barrier  */

main()
 {
    int count = 3;    /* number of threads participating */
     /* Thread 1 initializes the barrier and waits for others to arrive
    barrier=NWSMPBarrierAlloc(count, "name");
       ...
    NWSMPBarrierWait(barrier); /* wait for others to arrive */
       ...
     /* All three threads at the barrier, continue processing. */
        ...
    _____
     /* Thread 2 calls the barrier, and waits for others. */
       ...
    NWSMPBarrierWait(barrier); /* wait for others to arrive */
     /* All three threads at the barrier, continue processing. */
        ...
     _____
     /* Thread 3 calls the barrier, and waits for others. */
       ...
    NWSMPBarrierWait(barrier); /* wait for others to arrive */
     /* All three threads at the barrier, continue processing. */
        ...
    NWSMPBarrierDestroy(barrier);
  }
```

SMP Read-Write Lock Functions

Counting semaphores work well with a single class of resource consumers. If, however, one class called "reader" references a resource without modifying its contents, and the other class called "writers" makes modifications, then another synchronization mechanism can be used. This mechanism is known as multiple reader / single writer locking or a read-write lock. Semantically, the class of readers excludes the class of writers, and any writer excludes both the class of readers and the other members of the class of writers. The interface does not provide lock upgrades (from read to write), and lock downgrades (from write to read) required for two phase locking. These operations are not required since users can provide their own locking strategy to handle this problem (for instance, through an auxiliary lock). This allowed us to separate the interfaces of the read-write locks and optimize their implementation. As in the case of mutex locks, the read-write locks will appear to be blocking, however, for optimization the implementation may choose to spin for some amount of time before blocking.

Semaphores described in the previous section are adequate to solve a wide variety of synchronization problems by guaranteeing exclusive access to the shared resources. They are intended however, to operate only with a single class of the resource consumers. A common synchronization problem occurs

when more than one class of the resource consumers is identified and allowed to access a shared resource. Each class may use its own definition of access semantics to the resource it controls.

The P(), V() operations must commute to avoid the race condition between a waiter that is being blocked and the owner of the lock attempting to unblock the waiter. For instance, one class may allow its members concurrent access to a resource, while excluding members of other classes. Note that in a general case, there is no limitation on a number of defined classes, as there is no limitation on type of semantics that can be associated with such a construct. The most common example presenting this construct is the "read-write" problem ensuring that the class of "readers" excludes the class of "writers," and that any member of writers excludes both the class of readers and the other members of the class of writers.

Although the problem is easy to understand, and typically expressed in the lock() and unlock() operations, the proposed solutions are quite complex. The reason for this complexity is that while semaphores are well suited to inhibiting other threads within a single class, they cannot directly be used by one class to inhibit other classes. The two common approaches to read-write locking are: (1) reader priority, and (2) writer priority. NetWare SMP uses writer priority.

The goal of the reader priority solution is to ensure minimum delay for readers. This can be described by the following pseudo code:

```
READERS
     P(mutex);
    readers = readers + 1;
    if readers = 1 then P(w);
    V(mutex);
       reading
      . . .
     P(mutex);
    readers = readers - 1;
    if readers = 0 then V(w);
    V(mutex);
     WRITERS
     P(extra);
    P(w);
       writing
      . . .
     V(w);
    V(extra);
```

By nesting writer code within P/V pair of an additional binary semaphore extra, the readers will be given priority over the writers. While a writer is in its writing region all other writers are queued on extra, not on writer. Thus when w is released the waiting reader, if any, will be awakened. The solution ensures that readers exclude writers and that a writer inhibits other writers with extra before contending with readers. The problem with this solution is that it can potentially starve the writers. A similar problem exists with the writer priority solution. In this case the readers can potentially be

starved.

An additional problem here is that even the application developer would not always be able to make the right choice. That is because some applications behavior, aside from its design characteristics, will depend, in some loose sense, on the external environment (that hardware configuration, usage profile, etc.) and irrespective of the selected solution can potentially starve either readers, or writers. An alternative, and the simplest solution is FIFO ordering, where the access to a resource is provided to all readers on the queue up to the first encountered writer. An apparent weakness of this approach is that it will not always provide as much of parallelism as either reader or writer priority. It does however, simplify the problem and avoids both readers and writers starvation. The thread selection algorithm may further depend on a particular scheduling policy. For those reasons, under time-sharing policy the read-write locks described in the next section employs FIFO ordering as the algorithm for the thread selection. Under other scheduling policies the thread selection algorithm maybe policy specific.

SMP Recursive Mutex Functions

A mutex can also be defined as recursive: if the rmutex is already locked by the calling thread, the recursive depth is incremented and control is returned to the caller, as if the lock had just been acquired.

SMP Semaphore Functions

CLib semaphores that map to SMP functionality are aimed at processes that do not share memory. Thus by definition, they do not perform as the synchronization functions covered in this section, so duplication of semaphore operations is not a concern. The parameter *count* supplied during allocation defines the initial count of resources protected by the semaphore.

A semaphore is a user-level synchronization mechanism that sleeps in the NetWare SMP environment. Sleeping semaphores are particularly well suited for mutual exclusion and event synchronization and are included in the set of user- level synchronization operations. Unlike spin locks, sleeping semaphores are not wasteful of processor cycles while a thread is waiting, but the higher level functionality that they provide usually results in a lower efficiency. Sleeping semaphores are used to block-wait for an event, or when a critical section is long.

Since blocking on a semaphore involves context switch, much stronger support from the operating system is required. The system must provide a mechanism for detecting when a condition is met, and means of selecting among the waiting processes (e.g., from an associated queue). Like spin locks, there are several variations of semaphores; however, functionality of

the blocking operation is the same. Counting semaphores limited to 0 and 1 are called binary semaphores. Those initialized to n are general or counting semaphores and are useful when multiple instances of a shared resource are present. When a semaphore has a value greater than 0 it is open; otherwise it is closed. The value of the semaphore after the increment/decrement operation specifies the action of the semaphore function.

Thus assuming an initial value of one for a semaphore, the P() operation inhibits all threads except the first from continuing, and the V() operation releases a single thread (if one is waiting) from its inhibited condition.

Two operations on semaphores P() and V() are defined as follows:

```
P(S)
 {
   S = S - 1;
   if( S < 0) {
     block( Delay the calling thread and append it to the semaphore que
   }
 }
```

The P() operation attempts to acquire a resource under the semaphore control by decrementing the value of the semaphore. If the resulting value is less than 0, the caller is suspended, and added to the queue of threads sleeping on this semaphore else, it acquired the semaphore resource and the caller is allowed to continue. If a thread blocks, it will be implicitly awarded the resource when it is re-activated.

```
V(S)
 {
 S = S + 1;
 if( S < 0) {
   unblock(Remove a thread from the semaphore queue and append it to th
  }
 }
```

The V() operation releases the lock by incrementing the value of the semaphore. If the resulting value is less than or equal to 0, it removes the next thread from the semaphore queue and appends it to the ready queue. The resource is implicitly awarded to the newly-activated thread. These P() and V() operations are sometimes provided as multiprocessor instructions or microcode routines, or operating system calls to the process manager. In either case, they must commute in the sense that V() wakes up exactly one process and that the sequence `P(); V()' has the same net effect as "V(); P()".

SMP Spin Lock Functions

The spin lock is a particular implementation of a mutex. Spin locks should be used very sparingly because applications seldom have asynchronous processes to unlock the thread waiting on the spin. The interfaces described

here are intended to provide the basic functionality and the minimum complexity. Spin locks can be used only when there is a guarantee that a thread will not be preempted or blocked while holding a spin lock. They are exclusively intended for the applications demanding high efficiency on a multiprocessor and should never be used on uniprocessor systems.

The simplest user-level synchronization mechanism is a spin-mutex called spin lock. It is an atomically modified boolean value that is set to 1 when the lock is held and reset to 0 when the lock is free. When a thread requests the lock that is already held by another thread, it spins in a loop (`busy-waiting') testing whether the lock has become available.

Such spinning wastes processor cycles and can slow processors doing useful work, including the one holding the lock, by consuming communication bandwidth. Since spin locks are the basic building blocks for other locking mechanisms efficiency, as opposed to fairness is the rule for their implementation. For this reason, spin locks are often implemented with machine instructions with minimal state.

The two basic spin lock operations are Lock and Unlock as illustrated below:

```
Lock:

 spinlock(S)
 {
   while(TEST&SET(S) == 1)
   continue;
 }
```

Set the lock variable value S to 1 if and only if its original value is 0, otherwise spin until the holder of the lock releases it (resets it to 0).

```
Unlock:

 spinunlock(S)
 {
   S = 0;
 }
```

Reset the lock value to 0, allowing spinners to acquire the lock.

The major assumption here regarding spin locks is that they are never held across sleep/blocking operations. This makes sense given that they are considered to be critical resources that must be accessed frequently for a short period of time. It will be the responsibility of each application to unlock all spin locks before calling sleep or sleep-blocking on a synchronization operation.

Spin-waiting wastes computing resources and its precise impact on the system behavior varies along several architectural dimensions: how processors are connected to memory, whether or not each processor has a hardware-managed coherent private cache, and if so, the coherence protocol. If spin-waiting is consuming communication bandwidth it can slow processors doing useful work, including the one holding the lock.

If two or more threads are spinning on a lock, it may cause a lock contention, which if high enough may cause the addition of processors to produce no net gain in effective processing power. Lock contention depends on the length of time threads hold each lock and the rate at which they obtain each lock. Independent studies reviewed spin locks algorithms and their trade-offs in terms of memory requirements, impact on communication bandwidth, uncontested latency, and contribution to the critical section time.

Locks that may encounter contention should be protected with some form of collision avoidance (for instance, adaptive backoff). Adaptive backoff is recommended for general use. Even though spin-waiting is wasteful, spin locks are useful when sleep is not permitted. Using spin locks on a uniprocessor can lead to serious side-effects including deadlocks, especially when the critical section is small, so that the expected spin on CPU is less costly than blocking and resuming the process. In the case of NetWare, deadlocks on a uniprocessor or multiprocessor configuration with spin locks will generate a server abend.

Since a wide spread perception of a spin lock as a very fast synchronization function may frequently lead to its misuse and exposure of unaware users to serious side-effects. The use of spin locks carries the explicit warning of possible serious side-effects. Mutex lock is the preferred interface of mutual exclusion. For most input/output and communications applications, spin locks provide the best solution. Interrupt paths should always use spin locks.

Synchronization Granularity and Fairness

The most important attribute of any synchronization protocol is an efficient implementation which determines the level of granularity that such a protocol can be used. Since usually on a multiprocessor, a busy wait implementation of a synchronization operation is more efficient than its sleep wait equivalent, the busy wait can be used with finer grain of parallelisms.

NetWare SMP implements synchronization fairness as FIFO (meaning first requested, first obtained) or FIFO with designated priority. For some functions, fairness is not an issue. Spin locks, semaphore locks, and mutex locks use FIFO fairness. Read/write locks use FIFO fairness, with writers having priority over readers. Fairness does not apply to condition variables or barriers.

Synchronization Performance

The paramount concern for synchronization operations is efficiency. Even applications with sufficient parallelism can fail to achieve substantial

speed-up because of the synchronization overhead.

The performance of a tightly coupled shared-memory multiprocessor system may be highly dependent on the amount of overhead incurred due to synchronization.

To minimize the run-time overhead the synchronization operations issue system calls only when absolutely necessary (for example, when blocking).

The delay between when a lock is released and when it is reacquired by a thread must also be minimized. Such a delay, termed the lock latency, is important to applications with high synchronization rate with little contention.

Thread Migration in SMP

Threads must be created under the context of CLib by way of the existing **BeginThread** and **BeginThreadGroup** functions. For appropriate information, see Thread and NLM Code Development.

For threads to take advantage of SMP in NetWare, following creation, threads should call **NWThreadToMP** from the running thread. Any thread that calls this function migrates into the context of the SMP kernel. Calling **NWThreadToNetWare** schedules an SMP thread back to the NetWare kernel. Threads should generally migrate to MP and stay there until termination. Migrating threads back to NetWare results in a heavier performance penalty than migrating threads to MP.

The thread migration functions allow application programs to move threads between the SMP and the native NetWare environments without converting or altering the data. This functionality enables native NetWare modules such as CLib to process portions of a large job, then move threads in a suspended or sleep mode over to SMP where threads can be quickly processed by multiple processors. No action is taken upon threads yielded into the new environment during migration. It is as if they went to sleep in one environment and awakened in another.

Threads and SMP

Effectively extending parallel execution beyond uniprocessing requires a mechanism to cooperate easily in sharing memory and other resources. This extension mechanism is typically called a thread or thread of control and is the basic unit of processor utilization. A thread is an abstract concept of execution in a shared address space---a sequence of instructions executed as an independent entity, scheduled and synchronized by kernel software.

A major objective for using threads is to enhance performance. Algorithms used in multi-processing, parallel computing, and distributed operating systems are often more simply expressed as threads of control that share

address space and file descriptors. Threads are a convenient way to process coarse- to medium-grained parallel algorithms. The SMP API is the interface that manages, schedules, migrates, and synchronizes threads on a multiprocessor platform.

Threads and Synchronization in SMP

NetWare SMP provides a generic, shared memory, multi-processor environment for the NetWare distributed operating system. SMP gives users the ability to execute multiple threads simultaneously on multiple processors within a single address space. Threads are discussed in the following paragraphs.

Related Topics

Threads and SMP

User Level Synchronization

General Requirements for Synchronization Objects

Synchronization Granularity and Fairness

Synchronization Performance

Marshalling

User Level Synchronization

Synchronization serves the dual purpose of ensuring the mutually exclusive access to shared variables and enforcing the correct sequencing of threads.

The first form of synchronization, mutual exclusion, is implemented by some form of a mutual exclusion lock---mutex lock for short---and provides atomic access to shared resources. The second form, event synchronization, is used to signal/wait the occurrence of (possibly state-less) events.

Although individual synchronization functions are more naturally used as one of the two forms of synchronization, some functions can be used as either mutexes or events. For example, a semaphore initialized to `1' can act as a mutex if the operation decrementing the semaphore value (P() operation) is used to acquire the exclusive access to a shared object, and the operation incrementing the semaphore value (V() operation) is used to release the mutex. Similarly, a semaphore initialized to `0' can act as an event if the P() operation is used to await an event, while the V() operation is used to signal the event. Uniprocessors typically handle mutual exclusion using disabling interrupts at appropriate times to avoid concurrent (strictly speaking reentrant) access to the shared data. However, this method is not sufficient in a multiprocessing environment, since multiple activities may be

accessing and modifying shared data concurrently. Access must be controlled by a set of synchronization protocols.

The synchronization protocols are implemented in two basic ways: 1) busy wait and 2) sleep wait. Synchronization operations can be optimized by creating hybrids---called adaptive wait---of these two basic protocols.

If the wait for the exclusive access to a shared object is expected to be short, the function will busy wait; that is, it will continue to run by repeatedly checking the value of the synchronization variables until they reach the desired state. Busy wait can slow processors doing useful work, including the one operating on the shared variable, and may lead to memory contentions known as hot spots. The benefit of busy wait is that any change to the synchronization variable is detected quickly, so the overall latency before the lock is captured is short.

If the wait for the exclusive access to a shared object is expected to be long, the caller can be switched out of the processor and sleep wait on a queue, allowing other activities to run on that processor until a certain condition is true. This synchronization protocol requires much stronger system support than busy waiting. The system must provide a mechanism for detecting when a condition is met, and the means of selecting among the waiting processes (for instance, from an associated queue). In the hybrid of busy wait to sleep wait, adaptive wait, the caller busy waits for a while before sleep waiting for a condition; when the condition occurs the sleeping caller wakes up, rechecks the condition and proceeds.

Memory contentions can be dealt with either at the software level by changing the locking granularity, or at the hardware level by combining, in which several requests for the same variable can be combined into a single request. Combining requests reduces communication traffic and reduces memory accesses. Since combined requests can themselves be combined, any number of concurrent memory references to the same location can be satisfied in the time required for one central memory access. This permits contention-free implementation of many synchronization protocols. Unfortunately, combining is expensive, due to the actual implementation costs and the performance penalty for requests that don't use the combining feature.

SMP: Functions

NWSMPBarrierAlloc

Allocates and initializes a barrier
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

barrier_t  NWSMPBarrierAlloc (
   nuint32  count,
   pnstr    name);
```

Parameters

count

(IN) Specifies the number of threads that must rendezvous at the barrier.

name

(IN) Points to a text string containing the name of the barrier object.

Return Values

On success, **NWSMPBarrierAlloc** returns a barrier. Otherwise, the function fails and returns NULL.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPBarrierAlloc allocates a barrier to a known state. Thereafter, it can be used any number of times to synchronize execution of the count threads.

See Also

NWSMPBarrierDestroy, **NWSMPBarrierWait**

NWSMPBarrierDestroy

Destroys a barrier
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPBarrierDestroy (
   barrier_t   barrier);
```

Parameters

barrier

(IN) Points to the barrier.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	An attempt to destroy *barrier* was made while the barrier was referenced by other threads.
0xFFFF	EINVAL	Bad parameter

WARNING: The EBUSY error return cannot be made fully reliable; application code must ensure that no thread attempt to destroy the barrier while it is in use.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPBarrierDestroy destroys *barrier*. This includes invalidating *barrier* and freeing any associated dynamically allocated resources.

See Also

NWSMPBarrierWait

NWSMPBarrierWait

Blocks the calling thread at a barrier
Local Servers: blocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPBarrierWait (
   barrier_t   barrier);
```

Parameters

barrier

(IN) Points to the barrier.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPBarrierWait provides a simple coordination mechanism for threads. The caller is blocked at the barrier until all count threads reach the barrier. (Count was defined with **NWSMPBarrierAlloc**.). When the last thread reaches the barrier, all count blocked threads are released from the barrier and are allowed to resume execution. The barrier is automatically reinitialized after the waiting threads are released.

See Also

NWSMPBarrierDestroy

NWSMPCondAlloc

Allocates a condition variable to a known state
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

cond_t  NWSMPCondAlloc (
   pnstr   name);
```

Parameters

name

(IN) Points to a text string containing the name of the condition object.

Return Values

On success, **NWSMPCondAlloc** returns a condition variable. Otherwise, the function fails and returns NULL.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPCondAlloc allocates a condition variable to a known state. Once created, the condition can be used any number of times.

See Also

NWSMPCondBroadcast, **NWSMPCondDestroy**, **NWSMPCondSignal**, **NWSMPCondWait**

NWSMPCondBroadcast

Wakes up all the threads waiting on a condition variable
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPCondBroadcast (
   cond_t   cond);
```

Parameters

cond

(IN) Points to the condition variable.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPCondBroadcast wakes up all the threads waiting on the condition *cond*. This function wakes up all threads if more than one thread is waiting.

NWSMPCondBroadcast has no effect if there are no threads waiting on the indicated condition.

NWSMPCondBroadcast will typically be more efficient if the associated mutex used by waiters is held across the call.

See Also

NWSMPCondDestroy, NWSMPCondSignal, NWSMPCondWait

NWSMPCondDestroy

Destroys a condition variable
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPCondDestroy (
   cond_t   cond);
```

Parameters

cond

(IN) Points to the condition variable.

Return Values

0x0000	EOK	Success

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPCondDestroy destroys the condition variable pointed to by*cond*. This includes invalidating *cond* and freeing any associated dynamically allocated resources.

See Also

NWSMPCondBroadcast, **NWSMPCondSignal**, **NWSMPCondWait**

NWSMPCondSignal

Wakes up a single thread waiting on a condition variable
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPCondSignal (
   cond_t   cond);
```

Parameters

cond

(IN) Points to the condition variable.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPCondSignal wakes up a single thread, if one exists, waiting on the condition *cond*. **NWSMPCondSignal** has no effect if there are no threads waiting on the indicated condition.

NWSMPCondSignal will typically be more efficient if the associated mutex used by waiters is held across the call.

See Also

NWSMPCondBroadcast, **NWSMPCondDestroy**, **NWSMPCondWait**

NWSMPCondWait

Waits on a condition
Local Servers: blocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPCondWait (
   cond_t    cond,
   mutex_t   mutex);
```

Parameters

cond

(IN) Points to the condition variable.

mutex

(IN) Points to the mutex.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPCondWait allows the caller to wait for the occurrence of a condition at the condition variable pointed to by *cond*.

The mutual exclusion variable [mutex] pointed to by *mutex* must be locked by the calling thread upon entry to **NWSMPCondWait**, otherwise the behavior will be unpredictable.

NWSMPCondWait automatically releases the mutex, causes the calling thread to wait on the condition variable *cond*, and when the condition is

signaled or the wait is interrupted reacquires the mutex and returns to the caller.

Since a potential race condition exists between the time the condition is signaled and the mutex is relocked, the calling thread always check the indicated condition upon return. The calling thread is allowed to resume execution when the condition is signaled or broadcast, or when interrupted.

The logical condition should be checked on return, as a turn may not have been caused by a change in the condition.

See Also

NWSMPCondBroadcast, **NWSMPCondDestroy**, **NWSMPCondSignal**

NWSMPIsAvailable

Checks to see whether SMP is loaded and enabled on the server
Local Servers: TBD
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <mwsmp.h>

nuint32 NWSMPIsAvailable (void);
```

Return Values

0x00000001	TRUE	SMP is loaded and enable
0x00000000	FALSE	SMP is not loaded or not enabled

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPIsAvailable checks to determine whether SMP is loaded and enabled to the server. If SMP is both loaded and enabled, this function returns TRUE. If SMP is either not loaded or loaded but not enabled, this function returns FALSE.

See Also

NWSMPThreadToMP

NWSMPMutexDestroy

Destroys a mutex lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPMutexDestroy (
   mutex_t   mutex);
```

Parameters

mutex
(IN) Points to the mutex lock handle.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	An attempt to destroy *mutex* was made while the mutex was locked by another thread.
0xFFFF	EINVAL	Bad parameter

WARNING: The EBUSY error return cannot be made fully reliable; application code must ensure there are no thread attempt to destroy the mutex while it is in use.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPMutexDestroy destroys the mutex pointed to by *mutex*. This includes invalidating the mutex and freeing any associated implementation-allocated dynamic resources.

Any user-allocated dynamic storage is unaffected by **NWSMPMutexDestroy** and must be explicitly released by the program.

See Also

NWSMPMutexLock, NWSMPMutexTryLock, NWSMPMutexUnlock

NWSMPMutexLock

Locks a mutex lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPMutexLock (
   mutex_t   mutex);
```

Parameters

mutex
(IN) Points to the mutex.

Return Values

On successful completion, **NWSMPMutexLock** leaves the mutex in the locked state and returns zero to the caller. Otherwise, an appropriate error indicator value is returned.

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPMutexLock is used to acquire the mutex pointed to by *mutex*.

See Also

NWSMPMutexDestroy, **NWSMPMutexTryLock**, **NWSMPMutexUnlock**

NWSMPMutexSleepAlloc

Allocates and initializes a mutex lock
Local Servers: blocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

mutex_t  NWSMPMutexSleepAlloc (
   pnstr   name);
```

Parameters

name

(IN) Points to a text string containing the name of the mutex object.

Return Values

On success **NWSMPMutexSleepAlloc** returns a mutex object. Otherwise the function returns NULL.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPMutexSleepAlloc allocates and initializes a mutex in the unlocked state. Once allocated, the mutex can be used any number of times.

Operations on locks allocated and initialized with **NWSMPMutexSleepAlloc** are not recursive---a thread will block itself if it attempts to reacquire a mutex lock that it already has acquired.

See Also

NWSMPMutexDestroy, **NWSMPMutexLock**, **NWSMPMutexTryLock**, **NWSMPMutexUnlock**

NWSMPMutexTryLock

Conditionally locks a mutex lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPMutexTryLock (
   mutex_t   mutex);
```

Parameters

mutex
(IN) Points to the mutex.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	The mutex was already locked by another thread at entry.
0xFFFF	EINVAL	Bad parameter

Remark

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPMutexTryLock attempts once to acquire the mutual exclusion lock [mutex] pointed to by *mutex*.

If the lock is already locked upon entry, **NWSMPMutexTryLock** immediately returns to the caller without acquiring the lock.

For consistency, locks acquired with **NWSMPMutexTryLock** should be released with **NWSMPMutexUnlock**.

See Also

NWSMPMutexDestroy, NWSMPMutexLock, NWSMPMutexUnlock

NWSMPMutexUnlock

Erases the lock applied by **NWSMPMutexLock** and **NWSMPMutexTryLock**

Local Servers: nonblocking

Remote Servers: N/A

SMP Aware: Yes

Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPMutexUnlock (
   mutex_t   mutex);
```

Parameters

mutex

(IN) Points to the mutex.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPMutexUnlock unlocks the mutex pointed to by *mutex*. The locks acquired with **NWSMPMutexLock** and **NWSMPMutexTryLock** should be released with **NWSMPMutexUnlock**.

If there is one or more threads waiting for the mutex when **NWSMPMutexUnlock** is called, at least one waiting thread is allowed to attempt acquisition of the mutex.

The requirement that at least one waiting thread is allowed to attempt acquisition is sufficient if no interruption takes place in **NWSMPMutexLock** code.

See Also

NWSMPMutexDestroy, NWSMPMutexLock, NWSMPMutexTryLock

NWSMPRMutexAlloc

Allocates an initializes a recursive mutex lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

rmutex_t  NWSMPRMutexAlloc (
   pnstr   name);
```

Parameters

name

(IN) Points to a text string containing the name of the rmutex object.

Return Values

On success **NWSMPRMutexAlloc** returns an rmutex. Otherwise the function returns NULL.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRMutexAlloc allocates a recursive mutex and initializes it to the unlocked state.

All operations on locks allocated and initialized with **NWSMPRMutexAlloc** are recursive.

See Also

NWSMPRMutexLock, **NWSMPRMutexTryLock**, **NWSMPRMutexUnlock**, **NWSMPRMutexDestroy**

NWSMPRMutexDestroy

Destroys a recursive mutex lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRMutexDestroy (
   rmutex_t   rmutex);
```

Parameters

rmutex

(IN) Points to the rmutex.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	An attempt to destroy rmutex was made while rmutex was locked by another thread.
0xFFFF	EINVAL	Bad parameter

WARNING: The EBUSY error return cannot be made fully reliable; application code must ensure that no thread attempt to destroy the rmutex while it is in use.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPMutexDestroy destroys the recursive mutex *rmutex*. This includes freeing any dynamically allocated resources associated with rmutex and invalidates rmutex.

See Also

NWSMPRMutexLock, NWSMPRMutexUnlock, NWSMPMutexDestroy

NWSMPRMutexLock

Locks a recursive mutex lock
Local Servers: blocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRMutexLock (
   rmutex_t   rmutex);
```

Parameters

rmutex

(IN) Points to the rmutex.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRMutexLock, similarly to **NWSMPMutexLock**, acquires the recursive recursive mutex pointed to by *rmutex*.

If the recursive mutex is already locked by another thread upon entry, the calling thread is blocked until the lock becomes available.

If the recursive mutex is already held by the calling thread, the recursive depth is incremented and control is returned to the caller, just as if the lock had just been acquired.

The locks acquired with **NWSMPRMutexLock** should be released with **NWSMPRMutexUnlock**.

See Also

NWSMPRMutexTryLock, NWSMPRMutexUnlock, NWSMPRMutexDestroy

NWSMPRMutexOwner

Defines the thread (resource) that is the owner (controller) of the mutex
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

thread_desc_t NWSMPRMutexOwner (
   rmutex_t   rmutex);
```

Parameters

rmutex

(IN) Points to the rmutex.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRMutexOwner defines the thread (resource) that has ownership or control of the rmutex. Separations on rmutex locks are not recursive.

The thread_desc_t data type specifies the ID of the owner thread.

See Also

NWSMPRMutexLock, **NWSMPRMutexUnlock**, **NWSMPRMutexTryLock**, **NWSMPRMutexDestroy**

NWSMPRMutexTryLock

Conditionally locks a recursive mutex lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRMutexTryLock (
   rmutex_t  rmutex);
```

Parameters

rmutex

(IN) Points to the rmutex.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRMutexTryLock, is used to conditionally acquire the recursive mutex pointed to by *rmutex*.

If the recursive mutex is already held by another thread, **NWSMPRMutexTryLock** immediately returns to the caller without acquiring the lock. If the recursive mutex is already held by the calling thread, the recursive depth is incremented and control is returned to the caller, exactly as if the lock had just been acquired.

The locks acquired with **NWSMPRMutexTryLock** should be released with **NWSMPRMutexUnlock**.

See Also

NWSMPRMutexLock, **NWSMPRMutexUnlock**, **NWSMPRMutexDestroy**

NWSMPRMutexUnlock

Releases recursive locks acquired by **NWSMPRMutexLock**
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRMutexUnlock (
   rmutex_t   rmutex);
```

Parameters

rmutex

(IN) Points to the rmutex.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRMutexUnlock releases the rmutex applied by a previous call to **NWSMPRMutexLock**. The operation checks the identity of the caller and if the caller is the current owner of the rmutex it checks the depth count. If the depth count is greater than 0, it decrements the count and returns to the caller without releasing the mutex. When the depths count is found to be 0, the mutex is released.

If there is at least one thread waiting for mutex when the mutex is unlocked, at least one waiting thread is called to attempt acquisition of the mutex. **NWSMPRMutexUnlock** fails when applied to a rmutex not locked previously by the caller.

See Also

NWSMPRMutexLock, NWSMPRMutexTryLock, NWSMPRMutexDestroy

NWSMPRWLockAlloc

Allocates a read-write lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

rwlock_t  NWSMPRWLockAlloc (
   pnstr   name);
```

Parameters

name

(IN) Points to a text string containing the name of the read-write lock object.

Return Values

On success **NWSMPRWLockAlloc** returns a read-write lock. Otherwise, the function fails and returns NULL.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWLockAlloc allocates a read-write lock.

These locks are to be used where distinguishing type of the access will enhance concurrence. Once allocated, the lock can be used any number of times.

Operations on read-write locks are not recursive---a thread will block itself if it attempts to reacquire a lock that it already has acquired.

See Also

NWSMPRWLockDestroy, **NWSMPRWReadLock**, **NWSMPRWTryReadLock**, **NWSMPRWTryWriteLock**, **NWSMPRWUnlock**, **NWSMPRWWriteLock**

NWSMPRWLockDestroy

Destroys a read-write lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRWLockDestroy (
   rwlock_t   lock);
```

Parameters

lock

(IN) Points to the lock.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	An attempt to destroy the lock was made while the lock was locked or referenced by another thread.
0xFFFF	EINVAL	Bad parameter

WARNING: The EBUSY error return cannot be made fully reliable; application code must ensure that no thread attempt to destroy the read-write lock while it is in use.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWLockDestroy destroys the read-write lock pointed to by*lock*. This includes invalidating the lock and freeing any associated dynamically allocated resources.

See Also

NWSMPRWReadLock, NWSMPRWTryReadLock, NWSMPRWTryWriteLock, NWSMPRWUnlock, NWSMPRWWriteLock

NWSMPRWReadLock

Acquires a read-write lock in read mode
Local Servers: blocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRWReadLock (
   rwlock_t   lock);
```

Parameters

lock

(IN) Points to the lock.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWReadLock acquires the read-write lock pointed to by *lock* in read mode.

If the lock is currently held by another reader and there are no writers waiting, the thread proceeds by incrementing the reader count. However, if a writer holds the lock, the caller blocks. If a reader holds the lock the caller blocks only if doing so would cause the reader to block behind a waiting writer.

NWSMPRWReadLock is particularly useful for controlling access to the resources that are frequently read, but infrequently updated.

See Also

NWSMPRWLockDestroy, **NWSMPRWTryReadLock**, **NWSMPRWTryWriteLock**, **NWSMPRWUnlock**, **NWSMPRWWriteLock**

NWSMPRWTryReadLock

Conditionally acquires a read-write lock in read mode
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRWTryReadLock (
   rwlock_t   lock);
```

Parameters

lock

(IN) Points to the lock.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	The read-write lock pointed to by *lock* is already locked.
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWTryReadLock makes a single attempt to acquire the read-write lock pointed to by *lock* in read mode. If the lock cannot be acquired, **NWSMPRWTryReadLock** immediately returns to the caller.

If the lock is currently held by another reader and there are no writers waiting, **NWSMPRWTryReadLock** increments the reader count and returns success. If the lock is currently held by a writer, or there are writers waiting and the thread would have to block, **NWSMPRWTryReadLock** immediately returns to the requester with a failed.

NWSMPRWTryReadLock is particularly useful for controlling access to the resources that are frequently read, but infrequently updated.

The locks acquired with **NWSMPRWTryReadLock** should be released with **NWSMPRWUnlock**.

See Also

NWSMPRWLockDestroy, **NWSMPRWReadLock**, **NWSMPRWTryWriteLock**, **NWSMPRWUnlock**, **NWSMPRWWriteLock**

NWSMPRWTryWriteLock

Conditionally acquires a read-write lock in write mode
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRWTryWriteLock (
   rwlock_t   lock);
```

Parameters

lock

(IN) Points to the lock.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	The read-write lock pointed to by *lock* is already locked.
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWTryWriteLock makes a single attempt to acquire the read-write lock in write mode pointed to by *lock*. If the lock cannot be acquired, **NWSMPRWTryWriteLock** immediately returns to the caller.

The parameter *lock* points to the read-write lock on which the write lock is to be applied. A write request succeeds only when there are no other readers or writers already holding the lock.

The locks acquired with **NWSMPRWTryWriteLock** should be released with **NWSMPRWUnlock**.

See Also

NWSMPRWLockDestroy, **NWSMPRWReadLock**, **NWSMPRWTryReadLock**, **NWSMPRWUnlock**, **NWSMPRWWriteLock**

NWSMPRWUnlock

Releases a read-write lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRWUnlock (
   rwlock_t   lock);
```

Parameters

lock

(IN) Points to the lock.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	Lock is held currently by a reader of a writer
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWUnlock releases a lock acquired by a previous call to **NWSMPRWReadLock**, **NWSMPRWWriteLock**, **NWSMPRWTryReadLock**, or **NWSMPRWTryWriteLock**. It can determine the type of the lock that is being released by the current state of the *lock*. When the request is made by a reader, the reader count is decremented and if still more readers claim the lock, the lock is not released. The dequeuing can proceed only when there are no more readers holding the lock.

Should the unlock leave the lock released, the first waiting thread is

activated. If the thread activated is a reader, all subsequent readers are activated (up to the next writer or end of queue) and the count of readers holding the lock is changed to reflect this. If the thread activated is a writer, no other threads are activated and the lock is marked as being held by a writer. Writer starvation is prevented because writers always have higher priority than readers during process wake-up.

See Also

NWSMPRWLockDestroy, **NWSMPRWReadLock**, **NWSMPRWTryReadLock**, **NWSMPRWTryWriteLock**, **NWSMPRWWriteLock**

NWSMPRWWriteLock

Acquires a read-write lock in write mode
Local Servers: blocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPRWWriteLock (
   rwlock_t   lock);
```

Parameters

lock

(IN) Points to the read-write lock.

Return Values

0x0000	EOK	Success
0x0002	EDEADLOCK	Deadlock
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPRWWriteLock acquires the read-write lock in write mode pointed to by *lock*.

The request succeeds only when the lock is free of another write lock and of all read locks. The request will wait for any current readers to finish before acquiring the lock. All other write and read requests for the lock remain blocked until the lock is available again.

See Also

NWSMPRWLockDestroy, NWSMPRWReadLock, NWSMPRWTryReadLock, NWSMPRWTryWriteLock, NWSMPRWUnlock

NWSMPSpinAlloc

Allocates a spin lock and initializes it to the unlocked state
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

spin_t  NWSMPSpinAlloc (
   pnstr   name);
```

Parameters

name

(IN) Points to a text string containing the name of the spin lock object.

Return Values

On success this function returns a spin lock. Otherwise, it fails and returns NULL.

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPSpinAlloc allocates a spin lock in the unlocked state. Once allocated, the spin lock can be used any number of times. All operations on locks allocated with **NWSMPSpinAlloc** are nonrecursive.

See Also

NWSMPSpinLock, **NWSMPSpinTryLock**, **NWSMPSpinUnlock**, **NWSMPSpinDestroy**

NWSMPSpinDestroy

Destroys the lock, including invalidating the lock and freeing any associated dynamically allocated resources

Local Servers: nonblocking

Remote Servers: N/A

SMP Aware: Yes

Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPSpinDestroy (
   spin_t   lock);
```

Parameters

lock

(IN) Points to the spin lock.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	An attempt to destroy a spin lock was made while the spin lock was locked by another thread.
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

_spin_destory destroys the lock, including invalidating the lock and freeing any associated dynamically allocated resources.

See Also

NWSMPSpinLock, **NWSMPSpinTryLock**, **NWSMPSpinUnlock**

NWSMPSpinLock

Acquires a spin lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPSpinLock (
   spin_t  lock);
```

Parameters

lock

(IN) Points to the spin lock.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPSpinLock acquires a spin lock. The parameter *lock* points to the spin lock on which the lock is to be applied. If the lock is already locked, the calling thread loops until the lock can be acquired. In general, this operation is used when the resources are exclusively held for such short durations that releasing the processor via a context switch may not be optimal. The locks acquired with **NWSMPSpinLock** should be released with **NWSMPSpinUnlock**.

See Also

NWSMPSpinTryLock, **NWSMPSpinUnlock**, **NWSMPSpinDestroy**

NWSMPSpinTryLock

Makes a single attempt to acquire a spin lock
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPSpinTryLock (
   spin_t   lock);
```

Parameters

lock

(IN) Points to the spin lock.

Return Values

0x0000	EOK	Success
0x0001	EBUSY	Failed to get the lock
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPSpinTryLock makes a single attempt to acquire the spin lock pointed to by *lock*. If the spin lock is already locked, the option immediately returns to the caller without acquiring the lock.

In general, **NWSMPSpinTryLock**, like **NWSMPSpinLock**, is used when the resources are exclusively held for such short durations that releasing the processor via a context switch may not be optimal. The locks acquired with **NWSMPSpinTryLock** are released with **NWSMPSpinUnlock**.

See Also

NWSMPSpinLock, NWSMPSpinUnlock, NWSMPSpinDestroy

NWSMPSpinUnlock

Releases locks acquired with **NWSMPSpinLock** and **NWSMPSpinTryLock**
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

nuint32 NWSMPSpinUnlock (
   spin_t   lock);
```

Parameters

lock

(IN) Points to the spin lock.

Return Values

0x0000	EOK	Success
0xFFFF	EINVAL	Bad parameter

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

A spin lock acquired with **NWSMPSpinLock** and **NWSMPSpinTryLock** should be released with **NWSMPSpinUnlock**.

See Also

NWSMPSpinLock, **NWSMPSpinTryLock**, **NWSMPSpinDestroy**

NWSMPThreadToMP

Migrates the thread to the MP scheduler
Local Servers: TBD
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

void NWSMPThreadToMP (void);
```

Return Values

None

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPThreadToMP should be used only to initially force threads into the SMP environment.

NWSMPThreadToMP causes a significant performance overhead.

NWSMPThreadToMP causes the calling thread to switch environment context from uniprocessing under native NetWare to multiprocess in the SMP environment. No action is taken if the thread yields into the new environment. That is, the calling thread's execution is halted and the thread after awakening is placed in a runnable state in the SMP context.

The criteria for choosing a thread to run after the calling thread has yielded are not specified. It is possible for the calling thread to be rescheduled immediately, even if other runnable threads exist.

NWSMPThreadToMP should be viewed as a hint from the caller to the system, indicating that the caller has reached a point at which it is convenient to yield the processor to other threads.

NWSMPThreadToMP is used by multithreaded applications which need to exercise control over their scheduling. No privilege or special permission is required for **NWSMPThreadToMP**.

See Also

NWSMPThreadToNetWare

NWSMPThreadToNetWare

Migrates the thread to the NetWare scheduler
Local Servers: TBD
Remote Servers: N/A
SMP Aware: Yes
Service: SMP

Syntax

```
#include <nwsmp.h>

void NWSMPThreadToNetWare (void);
```

Return Values

None

Remarks

NOTE: For current expectations about NetWare SMP scaling, see Current NetWare SMP Scalability.

NWSMPThreadToNetWare should be used only to explicitly force threads into the NetWare environment for performance related tuning.

NWSMPThreadToNetWare causes a significant performance overhead.

NWSMPThreadToNetWare causes the calling thread to switch environment context from multiprocessing under SMP to uniprocess in the native NetWare environment. No action is taken if the thread yields into the new environment. That is, the calling thread's execution is halted and the thread after awakening is placed in a runnable state in the native NetWare context.

The criteria for choosing a thread to run after the calling thread has yielded are not specified. It is possible for the calling thread to be rescheduled immediately, even if other runnable threads exist.

NWSMPThreadToNetWare should be viewed as a hint from the caller to the system, indicating that the caller has reached a point at which it is convenient to yield the processor to other threads.

NWSMPThreadToNetWare is used by multithreaded applications which need to exercise control over their scheduling. No privilege or special permission is required for **NWSMPThreadToNetWare**.

See Also

NWSMPThreadToMP

String Conversion

String Conversion: Functions

ASCIIZToLenStr

Converts an ASCIIZ (NULL-terminated) string to a length-preceded string
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <nwstring.h>

int ASCIIZToLenStr (
   char   *destStr,
   char   *srcStr);
```

Parameters

destStr

(OUT) Specifies the destination string.

srcStr

(IN) Specifies the source string in ASCIIZ format.

Return Values

0	(0x00)	ESUCCESS	
1	(0x01)	ERR_STRING_EXCEEDS_LENGTH	Longer than 255 characters.

Remarks

ASCIIZToLenStr converts an ASCIIZ (NULL-terminated) string to a length-preceded string. A length-preceded string has the length of the string in the first byte, followed by the characters of the string.

The *destStr* and *srcStr* parameters might not point at the same string.

ASCIIZToMaxLenStr

Converts an ASCIIZ (NULL-terminated) string to a length-preceded string that is not longer than the specified maximum length

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

SMP Aware: Yes

Service: String Conversion

Syntax

```
#include <nwstring.h>

int ASCIIZToMaxLenStr (
   char   *lenString,
   char   *ASCIIZstring,
   int    maximumLength);
```

Parameters

lenString

(OUT) Specifies the destination (length-preceded) string.

ASCIIZstring

(IN) Specifies the source string in ASCIIZ format.

maximumLength

(IN) Specifies the maximum number of characters to place in the new string.

Return Values

-1	EFAILURE	The ASCIIZ string was longer than the *mamimumLength* .
0	ESUCCES	

Remarks

A length-preceded string has the length of the string in the first byte, followed by the characters in the string; it cannot exceed 255 characters. If the *ASCIIZstring* string is longer that 255 characters, this function returns EFAILURE, and *lenString* contains *maximumLength* characters. The

remaining characters of the *ASCIIZstring* are not copied to *lenString*.

Since length-preceded strings only have one byte to store the size of the string, the maximum size of the string is 255. Passing a maximum size larger than 255 produces unpredictable results.

Example

ASCIIZToMaxLenStr

```
#include <nwstring.h>
#include <errno.h>

main()
{
   char srcString[256];
   char destString[256];
   int ccode;
   int maxSize;
   strcpy(srcString,"This is the message");
   maxSize = 100;
   ccode = ASCIIZToMaxLenStr(destString, srcString, maxSize);
   if(ccode == ESUCCESS)
      printf("The string fit\n");
   else
      printf("The string was too long to fit in the allotted space.\n")
}
```

atof

Converts a string to double representation
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

double atof (
   const char   *ptr);
```

Parameters

ptr
(IN) Points to the string to be converted.

Return Values

atof returns the converted value. A value of 0 is returned when the input string cannot be converted. When an error has occurred,*errno* is set

Remarks

The **atof** function converts the string pointed to by*ptr* to double representation.

See Also

sscanf, **strtod**

Example

atof

```
#include <stdlib.h>

main ()
{
   double x;
   x=atof("3.1415926");
```

```
}
```

atoi

Converts a string to integer representation
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

int atoi (
   const char   *ptr);
```

Parameters

ptr
(IN) Points to the string to be converted.

Return Values

atoi returns the converted value.

Remarks

The **atoi** function converts the string pointed to by *ptr* to int representation.

See Also

sscanf, **strtoi**, **strtol**

Example

atoi

```
#include <stdlib.h>

main ()
{
   int x;
   x=atoi("-289");
}
```

atol

Converts a string to long integer representation
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

long int atol (
   const char   *ptr);
```

Parameters

ptr
(IN) Points to the string to be converted.

Return Values

atol returns the converted value.

Remarks

The **atol** function converts the string pointed to by *ptr* to long integer representation.

See Also

sscanf, **strtol**

Example

atol

```
#include <stdlib.h>

main ()
{
   long int x;
   x = atol( "-289" );
}
```

ecvt

Converts a floating-point number into a character string
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

char *ecvt (
   double   value,
   int      ndigits,
   int      *dec,
   int      *sign);
```

Parameters

value

(IN) Specifies the value to be converted into a string.

ndigits

(IN) Specifies the desired total number of significant digits.

dec

(OUT) Receives a pointer to the decimal point position relative to the first digit.

sign

(OUT) Receives a pointer to a positive or negative sign:

0 = Positive
Nonzero = Negative

Return Values

The **ecvt** function returns a pointer to a static buffer containing the converted string of digits. Note, both **ecvt** and **fcvt** use the same static buffer, except if you are using CLIB V 4.11 or above.

Remarks

The parameter *ndigits* specifies the number of significant digits desired. The converted number is rounded to *ndigits* of precision.

The character string contains only digits and is terminated by a NULL character. The integer pointed to by *dec* is filled in with a value indicating the position of the decimal point relative to the start of the string of digits. A zero or negative value indicates that the decimal point lies to the left of the first digit. The integer pointed to by *sign* contains 0 if the number is positive, and nonzero if the number is negative.

See Also

fcvt, **gcvt**, **printf**

Example

ecvt

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

main()
{
   char   str;
   int    dec, sign;
   str = ecvt( 123.456789, 6, &dec, &sign );
   printf( "str=%s,  dec=%d, sign=%d\n", str,dec, sign );
}
produces the following:
str=123457, dec=3, sign=0
```

fcvt

Converts the floating-point number value into a character string
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

char *fcvt (
   double   value,
   int      ndigits,
   int      *dec,
   int      *sign);
```

Parameters

value

(IN) Specifies the value to be converted into a string.

ndigits

(IN) Specifies the desired total number of significant digits to the right of the decimal point.

dec

(OUT) Receives a pointer to the decimal point position relative to the first digit.

sign

(OUT) Receives a pointer to a positive or negative sign: 0 = Positive; Nonzero = Negative

Return Values

The **fcvt** function returns a pointer to a static buffer containing the converted string of digits. Note, both **ecvt** and **fcvt** use the same static buffer, except if you are using CLIB V 4.11 or above.

Remarks

The difference between the **fcvt** and **ecvt** functions is that the parameter *ndigits* for the **fcvt** function specifies the number of digits desired to the right of the decimal point. The converted number is rounded to this

position.

The character string contains only digits, and it is terminated by a NULL character. The integer pointed to by *dec* is filled in with a value indicating the position of the decimal point relative to the start of the string of digits. A zero or negative value indicates that the decimal point lies to the left of the first digit. The integer pointed to by *sign* contains 0 if the number is positive, and nonzero if the number is negative.

See Also

ecvt, **gcvt**, **printf**

Example

fcvt

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

main()
{
   char *str;
   int  dec, sign;
   str=fcvt( -123.456789, 5, &dec, &sign ) ;
   printf( "str=%s, dec=%d, sign=%d\n", str,dec,sign ) ;
}
```

produces the following:

```
str=12345679, dec=3, sign=-1
```

gcvt

Converts the floating-point number value into a character string and stores the result in a buffer

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: String Conversion

Syntax

```
#include <stdlib.h>

char *gcvt (
   double   value,
   int      ndigits,
   char     *buffer);
```

Parameters

value

(IN) Specifies the value to be converted into a string.

ndigits

(IN) Specifies the desired total number of significant digits.

buffer

(OUT) Receives the character string.

Return Values

The **gcvt** function returns a pointer to the string of digits.

Remarks

The parameter *ndigits* specifies the number of significant digits desired. The converted number are rounded to this position.

If the exponent of the number is less than -4 or is greater than or equal to the number of significant digits wanted, then the number is converted into E-format. Otherwise, the number is formatted using F-format.

See Also

ecvt, **fcvt**, **printf**

Example

gcvt

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

main()
{
   char buffer [80] ;
   printf( "%s\n", gcvt(-123.456789, 5, buffer ) );
   printf( "%s\n", gcvt( 123.456789E+12, 5, buffer ) );
}
produces the following:
-123.46
1.2346E+014
```

itoa

Converts an integer to a string
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

char *itoa (
   int    value,
   char   *buffer,
   int    radix);
```

Parameters

value

(IN) Specifies the integer value to be converted.

buffer

(OUT) Receives a pointer to the character array.

radix

(IN) Specifies the base to be used in converting the integer.

Return Values

itoa returns the pointer to the result.

Remarks

The **itoa** function converts the integer value into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A NULL character is appended to the result.

The size of *buffer* must be at least (8 * **sizeof** (int) + 1) bytes when converting values in base 2. That makes the size 17 bytes on 16-bit machines and 33 bytes on 32-bit machines. If the value of *radix* is 10 and *value* is negative, a minus sign (-) is prepended to the result.

See Also

atoi, **strtol**, **utoa**

Example

itoa

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

int main()
{
   char    buffer[20];
   int     base;
   for (base=8; base<=16; base=base+2)
      printf ("%2d %s\n", base, itoa (12765, buffer, base) );
}
```

produces the following:

```
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

LenToASCIIZStr

Converts a length-preceded string to an ASCIIZ (NULL-terminated)
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: No
Service: String Conversion

Syntax

```
#include <nwstring.h>

int LenToASCIIZStr (
   char   *destStr,
   char   *srcStr);
```

Parameters

destStr

(OUT) Specifies the destination string.

srcStr

(IN) Specifies the source string.

Return Values

	(0x00)	ESUCCESS

The *srcStr* is only copied up to the first NULL character or the length specified, whichever is shorter. If the string is not converted (due to a NULL being encountered), the value returned is the number of characters not copied to the destination string.

Remarks

LenToASCIIZStr call copies and converts a length-preceded string to an ASCIIZ (NULL-terminated) string.

The *srcStr* parameter is an ASCII string with the length of the string of the first byte and the actual characters of the string following it.

The *srcStr* can be the same as the *destStr*.

ltoa

Converts a long integer to a string
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

char *ltoa (
   long int   value,
   char       *buffer,
   int        radix);
```

Parameters

value

(IN) Specifies the integer value to be converted.

buffer

(OUT) Receives a pointer to the character array.

radix

(IN) Specifies the base to be used when converting the integer.

Return Values

ltoa returns a pointer to the result.

Remarks

The **ltoa** function converts the integer value into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A NULL character is appended to the result. The size of *buffer* must be at least 33 bytes when converting values in base 2. If the value of *radix* is 10 and *value* is negative, then the result is prefixed with a minus sign (-).

See Also

atol, **strtol**, **strtoul**, **ultoa**

Example

ltoa

```
#include <stdlib.h>

void print_value (long value)
{
   int    base;
   char   buffer[33];
   for (base=8; base<=16; base=base+2)
      printf ("%2d %s\n", base, ltoa ( value, buffer, base ) );
}
```

produces the following:

```
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

strtod

Converts a string to double representation
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

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

double strtod (
   const char   *ptr,
   char         **endptr);
```

Parameters

ptr

(IN) Points to the string to be converted.

endptr

(OUT) Receives a pointer to the first unrecognized character.

Return Values

strtod returns the converted value. If the correct value causes overflow, plus or minus HUGE_VAL is returned according to the sign, and *errno* is set to ERANGE. If the correct value causes underflow, a value of 0 is returned, and *errno* is set to ERANGE. A value of 0 is returned when the input string cannot be converted.

Remarks

The **strtod** function converts the string pointed to by *ptr* to double representation. The function recognizes a string containing:

Optional white space

An optional plus (+) or minus (-) sign

A sequence of digits containing an optional decimal point

An optional e or E followed by an optionally signed sequence of digits

The conversion ends at the first unrecognized character. A pointer to that

character is stored in the object to which *endptr* points if *endptr* is not NULL.

See Also

atof

Example

strtod

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

double convert_pi ()
{
   double pi;
   pi = strtod ("3.141592653589793", NULL)
   return (pi);
}
```

strtoi

Converts an ASCII string to an integer
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

int strtoi (
   const char   *str,
   int          base);
```

Parameters

str

(IN) Specifies a pointer to the string to convert to an integer.

base

(IN) Specifies the number base of the string.

Return Values

strtoi returns the integer value for the input ASCII string and *base.* This value is inaccurate if a value in *str* is outside the range of the value specified in *base.* If a character in *p* is encountered outside the range of *base,* the function returns the value computed up to that point.

The *base* parameter can be 0, in which case *base* is given one of the following values:

16---if *str* begins with 0x, 0X, or contains any alphabetic characters
8---if *str* begins with 0 or 10.

See Also

atoi, **strtol**

Example

strtoi

```
#include <stdlib.h>

main()
{
   int decimal_value;
   int hex_value;
   decimal_value = strtoi( "1234567", 10 );

   /* decimal_value is printed as: 1234567 */
   printf( "decimal_value = %d\n", decimal_value );
   decimal_value = strtoi ("123a4567\n", 10);

   /* decimal_value is printed as: 123 */
   printf ("decimal_value = %d\n", decimal_value );
   hex_value = strtoi ("abc123", 16);

   /* hex_value is printed as: abc123 */
   printf ("hex_value = %x\n", hex_valu );
   hex_value = strtoi ("abch123", 16);

   /* hex_value is printed as: abc */
   printf ("hex_value = %x\n", hex_value);
}
```

strtol

Converts a string to an object of type long int
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

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

long int strtol (
   const char   *ptr,
   char         **endptr,
   int          base);
```

Parameters

ptr

(IN) Points to the string to be converted to an object.

endptr

(OUT) Receives a pointer to the first unrecognized character.

base

(IN) Specifies the base of the value being converted.

Return Values

strtol returns the converted value. If the correct value would cause overflow, LONG_MAX or LONG_MIN is returned according to the sign, and *errno* is set to ERANGE. If *base* is out of range, zero is returned and *errno* is set to EDOM. A value of 0 is returned when the input string cannot be converted. When an error has occurred, *errno* is set.

Remarks

The **strtol** function converts the string pointed to by *ptr* to an object of type long int. The function recognizes a string containing:

Optional white space

An optional plus (+) or minus (-) sign

A sequence of digits and letters

The conversion ends at the first unrecognized character. A pointer to that character is stored in the object to which *endptr* points if *endptr* is not NULL.

The *base* parameter must have a value between 2 and 36. The letters **a** through **z** and **A** through **Z** represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X can optionally precede the sequence of letters and digits.

See Also

ltoa, **strtoul**

Example

strtol

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

main ()
{
   long int v;
   v = strtol ("12345678", NULL, 10);
}
```

strtoul

Converts a string to an unsigned long integer.
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Conversion

Syntax

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

unsigned long int strtoul (
   const char   *ptr,
   char         **endptr,
   int          base);
```

Parameters

ptr

(IN) Points to the string to be converted.

endptr

(OUT) Receives a pointer to the first unrecognized character.

base

(IN) Specifies the base of the value being converted.

Return Values

strtoul returns the converted value. If the correct value would cause overflow, ULONG_MAX is returned and *errno* is set to ERANGE. If *base* is out of range, a value of 0 is returned and *errno* is set to EDOM. A value of 0 is returned when the input string cannot be converted. When an error has occurred, *errno* is set.

Remarks

The **strtoul** function converts the string pointed to by *ptr* to an unsigned long integer. The function recognizes a string containing optional white space, followed by a sequence of digits and letters. The conversion ends at the first unrecognized character. A pointer to that character is stored in the object to which *endptr* points if *endptr* is not NULL.

If base is zero, the first characters determine the base used for the

conversion. If the first characters are 0x or 0X, the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.

If *base* is not zero, it must have a value of between 2 and 36. The letters **a** through **z** and **A** through **Z** represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X can optionally precede the sequence of letters and digits.

See Also

ltoa, **strtol**, **ultoa**

Example

strtoul

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

main ()
{
   unsigned long int v;
   v = strtoul ("12345678", NULL, 10);
}
```

ultoa

Converts an unsigned long integer into the equivalent string in base notation

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: String Conversion

Syntax

```
#include <stdlib.h>

char *ultoa (
   unsigned long int   value,
   char                *buffer,
   int                 radix);
```

Parameters

value

(IN) Specifies an unsigned long value.

buffer

(OUT) Receives a pointer to the character array.

radix

(IN) Specifies the base to be used when converting the integer.

Return Values

ultoa returns the pointer to the result.

Remarks

The **ultoa** function converts the unsigned long integer *value* into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A NULL character is appended to the result. The size of buffer must be at least 33 bytes when converting values in base 2.

See Also

atol, **ltoa**, **strtol**, **strtoul**, **utoa**

Example

ultoa

```
#include <stdlib.h>

void print_value (unsigned long int value)
{
   int base;
   char buffer[33];
   for (base=2; base<36; ++base)
      printf ("%s\n", ultoa (value, buffer, base) );
}
```

utoa

Converts an unsigned integer into the equivalent string in base notation
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Conversion

Syntax

```
#include <stdlib.h>

char *utoa (
   unsigned int   value,
   char           *buffer,
   int            radix);
```

Parameters

value

(IN) Specifies an unsigned integer value.

buffer

(OUT) Receives a pointer to the character array.

radix

(IN) Specifies a base radix notation.

Return Values

utoa returns the pointer to the result.

Remarks

The **utoa** function converts the unsigned integer value into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A NULL character is appended to the result. The size of *buffer* must be at least 17 bytes when converting values in base 2.

See Also

atoi, **itoa**, **strtol**, **strtoul**

Example

utoa

```
#include <stdlib.h>

void print_value (unsigned int value)
{
   int  base;
   char buffer[18];
   for (base=2; base<36; ++base)
      printf ("%s\n", utoa (value, buffer, base) );
}
```

String Manipulation

String Manipulation: Functions

LenStrCat

Concatenates two length-preceded ASCII strings
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <nwstring.h>

char *LenStrCat (
   char   *destStr,
   char   *srcStr);
```

Parameters

destStr

(IN/OUT) Specifies the destination string.

srcStr

(IN) Specifies the source string.

Return Values

LenStrCat returns the address of the destination string.

Remarks

LenStrCat concatenates the *srcStr* onto the end of the *destStr*. The source and destination strings are two ASCII strings with the length of the string being the first byte of the string. NULL characters are copied the same as any other value. It is the programmer's job to make sure the destination string is large enough to hold the concatenated string.

LenStrCmp

Compares two length-preceded ASCII strings
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <nwstring.h>

int LenStrCmp (
   char   *string1,
   char   *string2);
```

Parameters

string1

(IN) Specifies the first string to be compared.

string2

(IN) Specifies the second string to be compared.

Return Values

A value < 0 if string1 < string2
A value = 0 if string1 = string2
A value > 0 if string1 > string2

Remarks

LenStrCmp compares two ASCII strings preceded by a byte length. The *string1* and *string2* parameters are two ASCII strings with the length of the string being the first byte of the string. This function emulates the standard **strcmp** function but works with length-preceded strings rather than NULL-terminated strings.

LenStrCpy

Copies a length-preceded ASCII string to another string
Local Servers: nonblocking
Remote Servers: N/A
Classification: 2.x, 3.x, 4.x
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <nwstring.h>

char *LenStrCpy (
   char   *destStr,
   char   *srcStr);
```

Parameters

destStr

(OUT) Specifies the destination string.

srcStr

(IN) Specifies the source string.

Return Values

LenStrCpy returns a pointer to *destStr*.

Remarks

The developer must ensure that the *destStr* parameter is large enough to contain the *srcStr* parameter. The source and destination strings are two ASCII strings with the length of the string being the first byte of the string. This function is similar to the standard **strcpy** function but works with length-preceded strings rather than NULL-terminated strings.

sprintf

Writes output to a specified character array under format control
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <stdio.h>

int sprintf (
   char         *buf,
   const char   *format,
   ... );
```

Parameters

buf

(OUT) Specifies the character array into which to place the output.

format

(IN) Points to the format control string.

Return Values

The **sprintf** function returns the number of characters written into the array, not counting the terminating NULL character. An error can occur while converting a value for output.

Remarks

The **sprintf** function is equivalent to **fprintf**, except that the argument *buf* specifies a character array into which the generated output is placed, rather than to a file. A NULL character is placed at the end of the generated character string. The format string is described under the description for the **printf** function.

See Also

fprintf, **printf**, **vsprintf**

Example

sprintf

To create a temporary file name using a counter:

```
#include <stdio.h>
char *make_temp_name ()
{
   static int  tempCount=0;
   static char namebuf[13];
   sprintf (namebuf, "ZZ%o6d.TMP", tempCount++);
   return (namebuf);
}

main ()
{
   int i;
   for (i=0; i<3; i++)
   printf ("%s\n", make_temp_name());
}
```

sscanf

Scans input from a character string under format control
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <stdio.h>

int sscanf (
   const char   *in_string,
   const char   *format,
   ... );
```

Parameters

in_string

(IN) Specifies a character string to scan.

format

(IN) Points to the format control string.

Return Values

The **sscanf** function returns EOF when scanning is terminated by reaching the end of the input string. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

Remarks

The **sscanf** function scans input from the character string *in_string* under control of the argument *format*. Following the format string is the list of addresses of items to receive values. The format string is described under the description of the **scanf** function.

See Also

fscanf, **scanf**, **vsscanf**

Example

sscanf

To scan the date in the form "Friday August 14 1991":

```
#include <stdio.h>
main ()
{
   int     day, year;
   char    weekday[20], month[20];
   sscanf ("Friday August 0014 1991", "%s %s %d  %d",
      &weekday, &month, &day, &year);
   printf ("%s %s %d %d\n", weekday, month, day, year);
}
```

produces the following:

```
Friday August 14 1991
```

strcat

Appends a copy of one string to the end of a second string (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strcat (
   char         *dst,
   const char   *src);
```

Parameters

dst

(OUT) Specifies the string to which to append a copy of another string.

src

(IN) Specifies the string to be copied.

Return Values

strcat returns the value of *dst*.

Remarks

The **strcat** function or macro appends a copy of the string pointed to by *src* (including the terminating NULL character) to the end of the string pointed to by *dst*. The first character of *src* overwrites the NULL character at the end of *dst*.

See Also

strncat

Example

strcat

```
#include <string.h>
```

```
#include <string.h>

#include <stdio.h>
main ()
{
   char    buffer[80];
   strcpy (buffer, "Hello ");
   strcat (buffer, "world");
   printf ("%s\n", buffer);
}
```

produces the following:

```
Hello world
```

strchr

Locates the first occurrence of a specified character in a string (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strchr (
   const char   *s,
   int          c);
```

Parameters

s

(IN) Specifies the string containing characters for which to search.

c

(IN) Specifies the character for which to search.

Return Values

strchr returns a pointer to the located character, or NULL if the character does not occur in the string.

Remarks

The **strchr** function or macro locates the first occurrence of*c* (converted to a char) in the string pointed to by*s*. The terminating NULL character is considered to be part of the string.

See Also

memchr, **strcspn**, **strrchr**, **strspn**, **strstr**, **strtok**

Example

strchr

```
#include <string.h>
```

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

main ()
{
   char    buffer[80];
   char    *where;
   strcpy (buffer, "01234ABCD");
   where = strchr (buffer, 'x');
   if (where == NULL)
   {
      printf  (" 'x' not found\n");
    }

}
```

strcmp

Compares two strings (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

int strcmp (
   const char   *s1,
   const char   *s2);
```

Parameters

s1

(IN) Specifies the string to be compared to the string pointed to by *s2*.

s2

(IN) Specifies the string to be compared to the string pointed to by *s1*.

Return Values

strcmp returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

Remarks

The **strcmp** function or macro compares the string pointed to by *s1* to the string pointed to by *s2*.

See Also

stricmp, **strncmp**, **strnicmp**

Example

strcmp

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

```
#include <stdio.h>

main ()
{
   printf ("%d\n", strcmp ("abcdef", "abcdef") );
   printf ("%d\n", strcmp ("abcdef", "abc") );
   printf ("%d\n", strcmp ("abc", "abcdef") );
   printf ("%d\n", strcmp ("abcdef", "mnopqr") );
   printf ("%d\n", strcmp ("mnopqr", "abcdef") );
}
```

produces the following:

```
0
1
-1
-1
1
```

strcmpi

Compares, with case insensitivity, two strings (implemented for NetWare® 3.11 and above)

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

int strcmpi (
   const char   *s1,
   const char   *s2);
```

Return Values

The **strcmpi** function returns an integer less than, equal to, or greater than 0, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*. All uppercase characters from *s1* and *s2* are mapped to lowercase characters for the purposes of doing the comparison.

Remarks

The **strcmpi** function compares, with case insensitivity, the string pointed to by *s1* to the string pointed to by *s2*.

See Also

strcmp, **stricmp**, **strncmp**, **strnicmp**

Example

strcmpi

```
#include <string.h>
main()
{
   printf( "%d\n", stricmp( "AbCDEF", "abcdef" ) );
   printf( "%d\n", stricmp( "abcdef", "ABC"    ) );
   printf( "%d\n", stricmp( "abc",    "ABCdef" ) );
   printf( "%d\n", stricmp( "Abcdef", "mnopqr" ) );
```

```
    printf( "%d\n", stricmp( "Mnopqr", "abcdef" ) );
}
```

produces the following:

```
0
100
-100
-12
12
```

strcoll

Compares two strings using the collating sequence of the current locale
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: String Manipulation

Syntax

```
#include <string.h>

int strcoll (
   const char   *s1,
   const char   *s2);
```

Parameters

s1

(IN) Specifies the string to be compared to the string pointed to by *s2*.

s2

(IN) Specifies the string to be compared to the string pointed to by *s1*.

Return Values

The **strcoll** function returns an integer less than, equal to, or greater than 0, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*, according to the collating sequence selected.

Remarks

The **strcoll** function compares the string pointed to by *s1* to the string pointed to by *s2*. The comparison uses the collating sequence selected by **setlocale**. The function is equivalent to **strcmp** when the collating sequence is selected from the "C" locale.

See Also

NWLstrcoll, **NWLsetlocale**, **setlocale**, **strcmp**

Example

strcoll

strcoll

```
#include <string.h>

main ()
{
   char   buffer[80];
   if (strcoll (buffer, "Hello") < 0)
      printf ("Less than\n");
}
```

strcpy

Copies a string into an array (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strcpy (
   char         *dst,
   const char   *src);
```

Parameters

dst

(OUT) Specifies the array into which to copy the string.

src

(IN) Specifies the string to be copied.

Return Values

strcpy returns the address of *dst*.

Remarks

The **strcpy** function or macro copies the string pointed to by *src* (including the terminating NULL character) into the array pointed to by *dst*. Copying of overlapping objects is not guaranteed to work properly. See the description for the **memmove** function to copy objects that overlap.

See Also

memmove, **strncpy**

Example

strcpy

```
#include <string.h>
main ()
```

```
main ()
{
   char    buffer[80];
   strcpy (buffer, "Hello ");
}
```

strcspn

Computes the length of the initial segment of a string consisting of characters **not** from a given set

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

size_t strcspn (
   const char   *str,
   const char   *charset);
```

Parameters

str

(IN) Specifies the string to be scanned.

charset

(IN) Specifies the set of characters for which to search.

Return Values

The **strcspn** function returns the offset position in a string where the first occurrence of *charset* begins.

Remarks

The **strcspn** function computes the length of the initial segment of the string pointed to by *str*, which consists entirely of characters not from the string pointed to by *charset*. The terminating NULL character is not considered part of *str*.

See Also

strspn

Example

strcspn

```
#include <string.h>

main ()
{
   printf ("%d\n", strcspn ("abcbcadef", "cba") );
   printf ("%d\n", strcspn ("xxxbcadef", "cba") );
   printf ("%d\n", strcspn ("123456789", "cba") );
}
```

produces the following:

```
0
3
9
```

strdup

Creates a duplicate of a string
Local Servers: blocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strdup (
   const char   *src);
```

Parameters

src

(IN) Specifies the string to be copied.

Return Values

The **strdup** function returns the pointer to the new copy of the string if successful; otherwise, it returns NULL.

Remarks

The **strdup** function creates a duplicate of the string pointed to by*src* and returns a pointer to the new copy. The memory for the new string is obtained by using the **malloc** function and can be freed using the **free** function.

See Also

strcpy

Example

strdup

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

main ()
```

```
{
   char    *new;
   new = strdup ("Make a copy");
   printf (new);
   free (new);
}
```

strerror

Maps an error number to an error message
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strerror (
   int   errnum);
```

Parameters

errnum

(IN) Specifies the error number to be mapped to an error message.

Return Values

The **strerror** function returns a pointer to the error message. The array containing the error string should not be modified by the program. This array can be overwritten by a subsequent call to the **strerror** function.

Remarks

The **strerror** function maps the error number contained in*errnum* to an error message.

See Also

perror

Example

strerror

```
#include <string.h>
#include <errno.h>

main ()
{
```

```
   FILE    *fp;
   fp = fopen ("file.nam", "r");
   if (fp == NULL)
      printf ("Unable to open file: %s\n", strerror (errno) );
}
```

stricmp

Compares, with case insensitivity, two strings
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

int stricmp (
   const char   *s1,
   const char   *s2);
```

Parameters

s1

(IN) Specifies the string to be compared to the string pointed to by *s2*.

s2

(IN) Specifies the string to be compared to the string pointed to by *s1*.

Return Values

The **stricmp** function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

Remarks

The **stricmp** function compares, with case insensitivity, the string pointed to by *s1* to the string pointed to by *s2*.

See Also

strcmp, **strncmp**, **strnicmp**

Example

stricmp

```
#include <string.h>
```

```
main ()
{
   printf ("%d\n", stricmp ("AbCDEF", "abcdef") );
   printf ("%d\n", stricmp ("abcdef", "ABC") );
   printf ("%d\n", stricmp ("abc",    "ABCdef") );
   printf ("%d\n", stricmp ("Abcdef", "nopqr") );
   printf ("%d\n", stricmp ("Mnopqr", "abcdef") );
}
```

produces the following:

```
0
100
-100
-12
12
```

strlen

Computes the length of a string (function or macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

size_t strlen (
   const char   *s);
```

Parameters

s

(IN) Specifies the string whose length is to be computed.

Return Values

strlen returns the number of characters that precede the terminating NULL character.

Remarks

The **strlen** function or macro computes the length of the string pointed to by *s*.

Example

strlen

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

main ()
{
   printf ("%d\n", strlen ("Howdy") );
   printf ("%d\n", strlen ("Hello world\n") );
   printf ("%d\n", strlen ("") );
}
```

produces the following:

```
5
12
0
```

strlist

Appends multiple strings together.
Local Servers: nonblocking
Remote Servers: N/A
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strlist (
   char          *dst,
   const char    *src1,
   ...);
```

Parameters

dst

(OUT) Specifies the string to which to append a copy of another string.

src1 ... srcn

(IN) Specifies the strings to be appended.

Return Values

strlist returns the value of *dst*.

Remarks

The **strlist** function appends multiple strings pointed to by *src1*, *src2*, ..., *srcn* to the string pointed to by *dst*. *srcn*, the last argument in the function, must be NULL. This function is only supported in CLIB V 4.11 or above.

strlwr

Replaces each character of a string with its lowercase equivalent
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>
char *strlwr (
   char   *str);
```

Parameters

str

(IN) Specifies the string to be converted to lowercase characters.

Return Values

The address of the converted string is returned.

Remarks

strlwr replaces the string *str* with lowercase characters by invoking the **tolower** function for each character in the string.

See Also

strupr

Example

strlwr

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

char source[ ] = { "A mixed-case STRING" };
main ()
{
   printf ("%s\n", source);
   printf ("%s\n", strlwr (source) );
```

```
    printf ("%s\n", source);
}
```

produces the following:

```
A mixed-case STRING
a mixed-case string
a mixed-case string
```

strncat

Appends a specified number of characters of one string to another string
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strncat (
   char         *dst,
   const char   *src,
   size_t       n);
```

Parameters

dst

(OUT) Specifies the string to which to append the characters.

src

(IN) Specifies the string containing the characters to be appended to the string pointed to by *dst*.

n

(IN) Specifies the maximum number of characters to append.

Return Values

The **strncat** function returns the value of *dst*.

Remarks

The **strncat** function appends not more than *n* characters of the string pointed to by *src* to the end of the string pointed to by *dst*. The first character of *src* overwrites the NULL character at the end of *dst*. A terminating NULL character is always appended to the result.

See Also

strcat

Example

strncat

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

char    buffer[80];
main ()
{
   strcpy (buffer, "Hello ");
   strncat (buffer,"world", 8);
   printf ("%s\n", buffer);
   strncat (buffer, "*************", 4);
   printf ("%s\n", buffer);
}
```

produces the following:

```
Hello world
Hello world****
```

strncmp

Compares a specified number of characters between two strings
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

int strncmp (
   const char   *s1,
   const char   *s2,
   size_t       n);
```

Parameters

s1

(IN) Specifies the string to be compared to the string pointed to by *s2*.

s2

(IN) Specifies the string to be compared to the string pointed to by *s1*.

n

(IN) Specifies the number of characters to be compared.

Return Values

The **strncmp** function returns an integer less than, equal to, or greater than 0, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

Remarks

The **strncmp** function compares not more than *n* characters from the string pointed to by *s1* to the string pointed to by *s2*.

See Also

strcmp, **stricmp**, **strnicmp**

Example

strncmp

```
#include <string.h>
#Include <stdio.h>

main ()
{
   printf ("%d\n", strncmp ("abcdef", "abcDEF", 10) );
   printf ("%d\n", strncmp ("abcdef", "abcDEF",  6) );
   printf ("%d\n", strncmp ("abcdef", "abcDEF",  3) );
   printf ("%d\n", strncmp ("abcdef", "abcDEF",  0) );
}
```

produces the following:

```
1
1
0
0
```

strncpy

Copies a specified number of characters from one string to another string
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strncpy (
   char         *dst,
   const char   *src,
   size_t       n);
```

Parameters

dst

(OUT) Specifies the array into which to copy the characters.

src

(IN) Specifies the string containing the characters to copy.

n

(IN) Specifies the number of characters to copy.

Return Values

The **strncpy** function returns the value of *dst*.

Remarks

The **strncpy** function copies no more than *n* characters from the string pointed to by *src* into the array pointed to by *dst*. Copying of overlapping objects is not guaranteed to work properly. See the "**memmove**" function if you want to copy objects that overlap.

If the string pointed to by *src* is shorter than *n* characters, NULL characters are appended to the copy in the array pointed to by *dst*, until *n* characters in all have been written. If the string pointed to by *src* is longer than *n* characters, only *n* characters are copied. No NULL characters are placed in *dst*.

See Also

Example

strncpy

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

main ()
{
   char buffer[15];
   printf ("%s\n", strncpy( buffer, "abcdefg", 10) );
   printf ("%s\n", strncpy( buffer, "1234567",  6) );
   printf ("%s\n", strncpy( buffer, "abcdefg",  3) );
   printf ("%s\n", strncpy( buffer, "*******",  0) );
}
```

produces the following:

```
abcdefg
123456g
abc456g
abc456g
```

strnicmp

Compares, with case insensitivity, a specified number of characters in one string to another string

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

int strnicmp (
   const char   *s1,
   const char   *s2,
   size_t       len);
```

Parameters

s1

(IN) Specifies the string to be compared to the string pointed to by *s2*.

s2

(IN) Specifies the string to be compared to the string pointed to by *s1*.

len

(IN) Specifies the number of characters to compare.

Return Values

The **strnicmp** function returns an integer less than, equal to, or greater than 0, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

Remarks

The **strnicmp** function compares, with case insensitivity, the string pointed to by *s1* to the string pointed to by *s2*, for at most *len* characters.

See Also

strcmp, **stricmp**, **strncmp**

Example

strnicmp

```
#include <string.h>
#Include <stdio.h>

main ()
{
   printf ("%d\n", strnicmp ("abcdef", "ABCXXX", 10) );
   printf ("%d\n", strnicmp ("abcdef", "ABCXXX",  6) );
   printf ("%d\n", strnicmp ("abcdef", "ABCXXX",  3) );
   printf ("%d\n", strnicmp ("abcdef", "ABCXXX",  0) );
}
```

produces the following:

```
-20
-20
0
0
```

strnset

Sets a specified number of characters in a string to a given character
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strnset (
   char     *s1,
   int      fill,
   size_t   len);
```

Parameters

s1

(OUT) Specifies the string to be filled with the specified character.

fill

(IN) Specifies the character be copied.

len

(IN) Specifies the number of bytes into which the fill character is to be copied.

Return Values

The address of the original string *s1* is returned.

Remarks

strnset fills the string *s1* with the value of the argument *fill*, converted to be a character value. When the value of *len* is greater than the length of the string, the entire string is filled. Otherwise, that number of characters at the start of the string is set to the fill character.

See Also

memset, **strset**

Example

strnset

```
#include <string.h>

char source[ ] = {"A sample STRING"};

main ()
{
   printf ("%s\n", source);
   printf ("%s\n", strnset (source, `=', 100) );
   printf ("%s\n", strnset (source, `*', 7) );
}
```

produces the following:

```
A sample STRING
===============
*******========
```

strpbrk

Locates the first occurrence in one string of any character from another string

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

char *strpbrk (
   const char   *str,
   const char   *charset);
```

Parameters

str

(IN) String for which to locate the first occurrence of any character from the string pointed to by *charset*.

charset

(IN) String containing the characters to be located in the string pointed to by *str*.

Return Values

The **strpbrk** function returns a pointer to the located character or NULL if no character from *charset* occurs in *str*.

Remarks

The **strpbrk** function locates the first occurrence in the string pointed to by *str* of any character from the string pointed to by *charset*.

See Also

strchr, **strcspn**, **strrchr**, **strtok**

Example

strpbrk

```
#include  <string.h>

main ()
{
   char *p;
   p = "Find all vowels";
   while (p != NULL)
   {
      printf ("%s\n", p);
      p = strpbrk (p+1, "aeiouAEIOU");
   }
}
```

produces the following:

```
Find all vowels
ind all vowels
all vowels
owels
els
```

strrchr

Locates the last occurrence of a specified character in a string
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strrchr (
   const char   *s,
   int          c);
```

Parameters

s

(IN) Specifies the string containing characters to be searched.

c

(IN) Specifies the character to locate.

Return Values

The **strrchr** function returns a pointer to the located character or a NULL pointer if the character does not occur in the string.

Remarks

The **strrchr** function locates the last occurrence of *c* (converted to a char) in the string pointed to by *s*. The terminating NULL character is considered to be part of the string.

See Also

strchr, strpbrk

Example

strrchr

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

```
#include <string.h>

main ()
{
   printf ("%s\n", strrchr ("abcdeabcde", 'a') );
   if (strrchr ("abcdeabcde",`x') == NULL)
      printf ("NULL\n");
}
```

produces the following:

```
abcde
NULL
```

strrev

Reverses the character order in a string
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strrev (
   char   *s1);
```

Parameters

s1

(IN) Specifies the string to be replaced.

Return Values

The address of the original string *s1* is returned.

Remarks

strrev replaces the string *s1* with a string whose characters are in the reverse order.

Example

strrev

```
#include <string.h>

char source[ ] = {"A sample STRING"};

main ()
{
   printf ("%s\n", source);
   printf ("%s\n", strrev (source) );
   printf ("%s\n", strrev (source) );
}
```

produces the following:

```
A sample STRING
GNIRTS elpmas A
A sample STRING
```

strset

Fills a string with a specified character
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strset (
   char   *s1,
   int    fill);
```

Parameters

s1

(IN) Specifies the string to be filled with the specified character.

fill

(IN) Specifies the character to be used in filling the string.

Return Values

The address of the original string *s1* is returned.

Remarks

strset fills the string *s1* with the character *fill*. The terminating NULL character in the original string remains unchanged.

See Also

strnset

Example

strset

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

char source[ ] = {"A sample STRING"};
```

```
char source[ ] = {"A sample STRING"};

main ()
{
   printf ("%s\n", source);
   printf ("%s\n", strset (source, '=') );
   printf ("%s\n", strset (source, '*') );
}
```

produces the following:

```
A sample STRING
===============
***************
```

strspn

Computes the length of the initial segment of a string consisting of characters from a given set

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

size_t strspn (
   const char   *str,
   const char   *charset);
```

Parameters

str

(IN) Specifies the string for which to compute the length of the initial segment.

charset

(IN) Specifies a set of characters.

Return Values

The **strspn** function returns the offset from the beginning of the string*str* where the characters in *charset* have ended.

Remarks

The **strspn** function computes the length of the initial segment of the string pointed to by *str*, which consists of characters from the string pointed to by *charset*. The terminating NULL character is not considered to be part of *charset*.

See Also

strcspn, **strpbrk**

Example

strspn

strspn

```
#include <string.h>
#Include <stdio.h>

main ()
{
   printf ("%d\n", strspn ("out to lunch", "aeiou") );
   printf ("%d\n", strspn ("out to lunch", "xyz") );
}
```

produces the following:

```
2
0
```

strstr

Scans a string for the first occurrence of a given substring
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strstr (
   const char   *str,
   const char   *substr);
```

Parameters

str

(IN) Specifies the string to be scanned.

substr

(IN) Specifies the substring for which to search.

Return Values

The **strstr** function returns a pointer to the located string, or NULL if the string is not found.

Remarks

The **strstr** function locates the first occurrence in the string pointed to by *str* of the sequence of characters (excluding the terminating NULL character) in the string pointed to by *substr*.

See Also

strcspn, **strpbrk**

Example

strstr

```
#include <string.h>
#include <strio.h>
```

```
#include <strio.h>

main ()
{
   printf ("%s\n", strstr ("This is an example", "is") );
}
```

produces the following:

```
is is an example
```

strtok

Breaks a string into a sequence of tokens, each of which is delimited by a character from another string

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

char *strtok (
   char         *s1,
   const char   *s);
```

Parameters

s1

(IN) Specifies the string to be broken into a sequence of tokens.

s2

(IN) Specifies the string containing the delimiter characters.

Return Values

The **strtok** function returns a pointer to the first character of a token or NULL if no token is found.

Remarks

The **strtok** function is used to break the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call to **strtok** returns a pointer to the first token in the string pointed to by *s1*. Subsequent calls to **strtok** must pass a NULL pointer as the first argument, in order to get the next token in the string. The set of delimiters used in each of these calls to **strtok** can be different from one call to the next.

The first call in the sequence searches *s1* for the first character that is not contained in the current delimiter string *s2*. If no such character is found, then there are no tokens in *s1* and the **strtok** function returns a NULL pointer. If such a character is found, it is the start of the first token.

The **strtok** function then searches from there for a character that is

contained in the current delimiter string. If no such character is found, the current token extends to the end of the string pointed to by*s1*. If such a character is found, it is overwritten by a NULL character, which terminates the current token. The**strtok** function saves a pointer to the following character, from which the next search for a token starts when the first argument is a NULL pointer.

Because **strtok** can modify the original string, that string should be duplicated if the string is to be reused.

See Also

strcspn, **strpbrk**

Example

strtok

```
#include <string.h>
main ()
{
   char *p;
   char *buffer;
   char *delims = { " .," };
   buffer = strdup ("Find words, all of them.");
   printf ("%s\n", buffer);
   p = strtok (buffer, delims);
   while (p != NULL)
   {
      printf ("word: %s\n", p);
      p = strtok (NULL, delims);
   }
   printf ("%s\n", buffer );
}
```

produces the following:

```
Find words, all of them.
word: Find
word: words
word: all
word: of
word: them
Find
```

strupr

Replaces each character of a string with its uppercase equivalent
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

```
#include <string.h>

char *strupr (
   char   *s1);
```

Parameters

s1

(IN) Specifies the string to be replaced with uppercase characters.

Return Values

The address of the original string *s1* is returned.

Remarks

strupr replaces the string *s1* with uppercase characters by invoking the **toupper** function for each character in the string.

See Also

strlwr

Example

strupr

```
#include <string.h>

char source[ ] = { "A mixed-case STRING" };

main ()
{
   printf ("%s\n", source);
```

```
    printf ("%s\n", strupr ( source) );
    printf ("%s\n", source);
}
```

produces the following:

```
A mixed-case STRING
A MIXED-CASE STRING
A MIXED-CASE STRING
```

strxfrm

Transforms a specified number of characters from one string to another string

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: No

Service: String Manipulation

Syntax

```
#include <string.h>

size_t strxfrm (
   char         *dst,
   const char   *src,
   size_t       n);
```

Parameters

dst

(OUT) Specifies the array into which to place the transformed characters.

src

(IN) Specifies the string to be transformed.

n

(IN) Specifies the number of characters to transform.

Return Values

The **strxfrm** function returns the length of the transformed string. If this length is more than *n*, the contents of the array pointed to by *dst* are indeterminate.

Remarks

The **strxfrm** function transforms, for no more than *n* characters, the string pointed to by *src* to the buffer pointed to by *dst*. The transformation uses the collating sequence selected by **setlocale** so that two transformed strings compare identically (using **strncmp**) to a comparison of the original two strings using **strcoll**. The function is equivalent to **strncpy** (except there is no padding of the *dst* argument with NULL characters when the argument *src* is shorter than *n* characters) when the collating sequence is selected from the "C" locale.

To determine how much room ia needed to store the results of the function, set *dst* to NULL and *n* to 0 and call the function with *src* specifying the strint to convert.

See Also

setlocale, **strcoll**

Example

strxfrm

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

main ( )
{
   char   dst[ ] = {"This is the string"};
   char   src [ ] = {"EXAMPLE"};
   int    n=4;
   printf ("%s\n", dst);
   printf ("%d\n", strxfrm (dst, src, n) );
   printf ("%s\n", dst);
   printf ("%s\n", src);
}
```

produces the following:

```
This is the string
EXAM is the string
EXAMPLE
```

swab

Copies bytes (which must be an even number of bytes) from a source buffer to a destination buffer, swapping every pair of characters

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: String Manipulation

Syntax

```
For CLIB V 4.11 or above:

#include <string.h>

void swab (
   cinst void   *src,
   void         *dst,
   size_t       n);

For all other CLIB versions:

#include <string.h>

void swab (
   char   *src,
   char   *dst,
   int    *n);
```

Parameters

src

(IN) Specifies the string to be transformed.

dst

(OUT) Specifies the array into which to place the transformed characters.

n

(IN) Specifies the number of characters to transform.

Return Values

None

Remarks

The **swab** function copies *n* bytes (which should be even) from *src* to *dst* swapping every pair of characters. This is useful for preparing binary data to be transferred to another machine that has a different byte ordering.

WARNING: The bytes copied must be of an even number, or the server will abend.

Example

swab

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

char *msg = "hTsim seasegi  swspaep.d";
#define NBYTES 24

main()
{
   auto char buffer[80];
   printf( "%s\n", msg );
   memset( buffer, `\0', 80 );
   swab( msg, buffer, NBYTES );
   printf( "%s\n", buffer );
}
```

produces the following:

```
hTsim seasegi  swspaep.d
This message is swapped.
```

swaw

Copies words (which should be even) from a source buffer to a destination buffer, swapping every two pairs of characters

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

SMP Aware: Yes

Service: String Manipulation

Syntax

```
#include <string.h>

void swaw (
   const void   *src,
   void         *dst,
   size_t       n);
```

Parameters

src

(IN) Specifies the string to be transformed.

dst

(OUT) Specifies the array into which to place the transformed characters.

n

(IN) Specifies the number of words to transform.

Return Values

None

Remarks

The **swaw** function copies *n* words (which should be even) from *src* to *dst* swapping every two pairs of characters. This is useful for preparing binary data to be transferred to another machine that has a different byte ordering. This function is only supported in CLIB V 4.11 or above.

Example

swaw

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

char *msg = "hTsim seasegi  swspaep.d";
#define NBYTES 12

main()
{
   auto char buffer[80];
   printf( "%s\n", msg );
   memset( buffer, `\0', 80 );
   swaw( msg, buffer, NBYTES );
   printf( "%s\n", buffer );
}
```

produces the following:

```
isThes mgesas  iapswd.pe
This message is swapped.
```

vsprintf

Formats data under control of the format control string
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: String Manipulation

Syntax

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

int vsprintf (
   char         *buf,
   const char   *format,
   va_list      arg);
```

Parameters

buf

(OUT) Specifies the buffer to which to write the result.

format

(IN) Points to the format control string.

arg

(IN) Specifies a variable argument.

Return Values

The **vsprintf** function returns the number of characters written, or a negative value if an output error occurred.

Remarks

The **vsprintf** function formats data under control of the format control string and writes the result to *buf*. The format string is described under the description for **printf**. The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced with *arg*, which has been initialized by the **va_start** macro.

See Also

fprintf, **printf**, **sprintf**, **va_arg**, **va_end**, **va_start**

Example

vsprintf

This example shows the use of **vsprintf** in a general error message routine.

```
#include <stdarg.h>
#include <stdio.h>
#include <striup.h>

char msgbuf[80];
char *fmtmsg (char *format, ... )

{
   va_list arglist;
   va_start (arglist, format);
   strcpy (msgbuf, "Error: ");
   vsprintf (&msgbuf[7], format, arglist);
   va_end (arglist);
   return (msgbuf);
}
```

vsscanf

Scans input from a string under format control
Local Servers: blocking
Remote Servers: N/A
Classification: Other
SMP Aware: Yes
Service: String Manipulation

Syntax

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

int vsscanf (
   const char   *in_string,
   const char   *format,
   va_list      arg);
```

Parameters

in_string

(IN) Specifies the string to be scanned.

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 string. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

Remarks

The **vsscanf** function scans input from the string designated by *in_string* under control of the argument *format*. The format list is described with the **scanf** function.

The **vsscanf** function is equivalent to the **sscanf** function, with a variable argument list replaced with *arg*, which has been initialized using the **va_start** macro.

See Also

fscanf, **scanf**, **sscanf**, **va_arg**, **va_end**, **va_start**

Example

vsscanf

```
#include <stdio.h>
#include <stdarg.h>
.
.
.
int Input (in_data, format, ...)
char *in_data, format;

{
   va_list    arglist;
   va_start (arglist, format);
   return (vsscanf (in_data, format, arglist));
}
```

Thread

Thread: Guides

Thread: General Guide

This is not a complete discussion of threads. For more detailed information about using threads, see NLM Code Development.

In the NetWare OS, a thread is a process that represents a single path of execution. An NLM can have more than one thread. The Thread Services functions allow you to start up and stop threads and perform various other thread-related operations as needed. They also allow you to start other NLM applications.

About Threads and Thread Management

- Threads
- Thread Management in NetWare 3.x
- Thread Management in NetWare 4.x
- When to Schedule a Routine as Work
- When to Schedule a Routine as a Thread
- Creating and Terminating Threads
- Thread Groups
- Creating and Terminating Thread Groups
- Thread Function List

Thread Priority

- Temporarily Handicapping Threads
- Permanently Handicapping Threads
- Low Priority Threads

Global Data

- NetWare Global Data
- Thread Global Data
- Thread Group Global Data
- NLM Global Data

Hierarchy of Global Data

Synchronization

Interprocess Synchronization

Additional Links

Thread: Functions

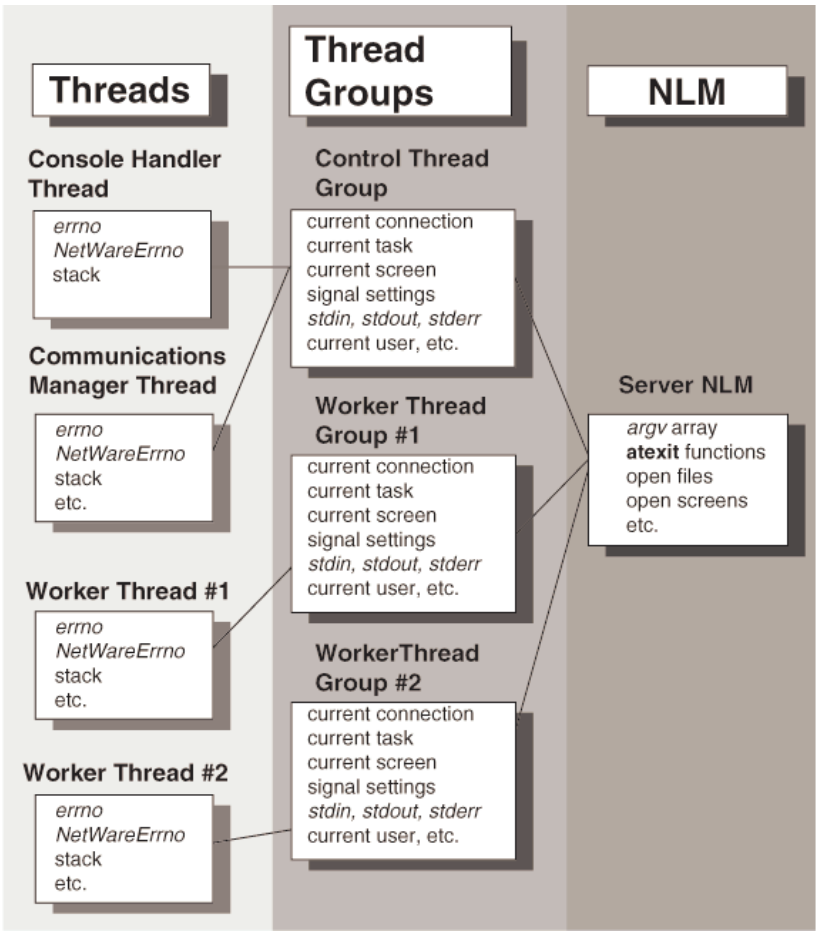
Thread: Concepts

Hierarchy of Global Data

The different scope levels of the global data items in the NetWare API form a three-tier hierarchy. The thread global data items comprise the bottom level of the hierarchy. At the middle level are the thread group global data items. The top level of the hierarchy includes NLM global data items.

For example, consider the organization of a hypothetical NLM that services requests from multiple clients, as shown in the following figure.

Figure 20. Hierarchy of Global Data Items



Bottom Level---Thread Global Data Items: Each of the four threads has its own set of global data items. The global data items in one thread cannot be referenced by any of the other threads.

Middle Level---Thread Group Global Data Items: The global data items for the Control Thread Group are common to two threads: the Console Handler Thread and the Communications Manager Thread. The global items for the worker Thread Group #1 can be referenced only by worker Thread #1. The global items for the worker Thread Group #2 can be referenced only by Worker Thread #2.

Top Level---NLM Global Data Items: The global data items for the Server NLM are common to all the thread groups and threads.

Parent Topic: NetWare Global Data

Interprocess Synchronization

Use local semaphores to control finite resources, to synchronize execution among threads, or to queue threads that need to use critical code sections. A semaphore has an associated signed 32-bit value.

Local semaphores can be used only by NLM applications running on a particular server (as opposed to network semaphores, which can be used by all NLM applications executing on, and all workstations attached to, the server).

The following Execution Thread functions deal with local semaphores:

CloseLocalSemaphore

ExamineLocalSemaphore

OpenLocalSemaphore

SignalLocalSemaphore

WaitOnLocalSemaphore

The **OpenLocalSemaphore** function allocates a semaphore and gives the NLM access (a handle) to it.

A thread can use the **WaitOnLocalSemaphore** call to gain access to the associated resource or to wait for the resource to become available. **WaitOnLocalSemaphore** can also be used to cause one thread to wait for another thread to signal it to continue. **WaitOnLocalSemaphore** decrements the semaphore's associated value.

When a thread is finished using a semaphore's resource, it typically calls **SignalLocalSemaphore** to increment the semaphore's value. A thread can also use this function to cause a thread that is waiting on a semaphore to resume execution. **SignalLocalSemaphore** increments the semaphore's associated value.

The **ExamineLocalSemaphore** call allows a thread to retrieve a semaphore's value. The semaphore value can be positive or negative (from -2^{31} through $2^{31}-1$). A negative value means that one or more threads are waiting on the semaphore.

Low Priority Threads

(NetWare 4.x only) Low priority threads run when there is nothing to run except hardware polling routines and temporarily handicapped threads. Currently, no NetWare applications are designed to run as low priority threads. Possible future programs are file compression utilities, once-a-week backup and cleanup utilities.

A thread can reschedule itself as a low priority thread by calling **ThreadSwitchLowPriority**.

To maintain a thread as low priority when you relinquish control, you must use **ThreadSwitchLowPriority**. If you use **ThreadSwitch**, the thread becomes a regular thread.

Parent Topic: When to Schedule a Routine as a Thread

NetWare 4.x Global Data

The NetWare API for NetWare 4.x has added custom data areas to the thread and tread group contexts. These custom data areas can be referenced with the following variables:

Table auto. Custom Data

Variable	Description
threadCustomDataPtr	A void pointer that points to a data area always associated with the current thread group. You can use this area to associate data with a thread.
threadCustomDataSize	The size (in bytes) of the data area pointed to by *threadCustomDataPtr*. This variable is a LONG.
threadGroupCustomDataPtr	A void pointer that points to a data area always associated with the current thread group. You can use this area to associate

	data with a thread group.
threadGroupCustomDataSize	The size (in bytes) of the data area pointed to by *threadGroupCustomDataPtr*. This variable is a LONG.

When a thread is running it can use the custom data associated with its thread and thread group. You can use these data areas to store in memory information associated with a thread or a thread group.

threadCustomDataPtr and *threadGroupCustomDataPtr*point to areas in memory that the NetWare API has set aside. Before using the data areas these pointers point to, you should check *threadCustomDataSize* and *threadGroupCustomDataSize* to see if the available space is sufficient. (These data areas may shrink with future versions of the OS.)

> **NOTE:** You should not change these pointers to point to data that you have allocated. However, you can use the data areas to hold the addresses to data that you have allocated.

Parent Topic: NetWare Global Data

NetWare Global Data

Because NLM applications can vary widely in their design and purpose, the NetWare API maintains a variety of global data items for NLM applications. These data items are divided into the following categories:

Thread Global Data

Thread Group Global Data

NLM Global Data

These global data items can be set and queried by various functions in the NetWare API. Each category of global data items represents a different level of scope or context.

Related Topics

NetWare 4.x Global Data

Hierarchy of Global Data

NLM Global Data

These data items have only one value for the entire NLM. The data items are global to all the thread groups and threads in the NLM. Any changes made to the values of NLM global data items affect all the thread groups and

to the values of NLM global data items affect all the thread groups and threads in the NLM.

Table auto. Global Data Item Examples

Data Item	Description
active "Advertisers"	Each NLM may have a set of active "advertisers" (started by **AdvertiseService**).
argv Array	This is the *argv* array passed to main.
atexit, AtUnload	Registers functions that are to be called when the NLM exits normally or is unloaded.
Libraries' Work Areas Pointers	Pointers to the data areas of any NLM libraries that the NLM has called (see Library for more information on library work areas).
locale Settings	Used by the locale functions.
Open Directories	The set of directories (opened by **opendir**) that the NLM has opened.
Open IPX/SPX Sockets	The set of IPX/SPX™ sockets that the NLM has opened.
Open Files	The set of files the NLM has opened. First-level open files include those opened with **open**, **sopen**, **create**; second-level files include those opened with **fopen**, **fdopen**, **freopen**.
Open Network Semaphores	The set of network semaphores (opened by **OpenSemaphore**) the NLM has opened.
Open Screens	The set of screens the NLM has opened.
Original Command Line	A copy of the original command line that was entered when the NLM was started is saved (used by **getcmd**).
Resource Tag	A tag used whenever the NLM allocates memory.
Thread Name	Pattern used for naming new threads (used by **BeginThread** and **BeginThreadGroup**).

Parent Topic: NetWare Global Data

Permanently Handicapping Threads

(NetWare 4.x only) If a particular NLM is being "bad" and does not yield often enough, the OS places a handicap in the thread's process control block (PCB), which prevents the thread from being rescheduled immediately. For example, it the OS places a handicap of 100 on the "bad" thread, 100 other pieces of work or threads run and yield before the "bad" thread is

rescheduled in the RunList Queue.

A thread can also handicap itself by calling **SetThreadHandicap**.

Parent Topic: When to Schedule a Routine as a Thread

Temporarily Handicapping Threads

(NetWare 3.x and 4.x) If a thread needs a resource that will not be ready for a moment, but you do not want it to assume the overhead of sleeping on a semaphore or doing busy waiting, you can have the thread reschedule itself with a temporary handicap using **ThreadSwitchWithDelay**.

An example of busy waiting is the following:

```
while (!finished)
   ThreadSwitchWithDelay();
```

Temporarily handicapped threads are not placed in the Run Queue until their handicap has expired. Upon expiration, they are rescheduled at the end of the Run Queue. Letting threads temporarily handicap themselves prevents needless rescheduling overhead caused by a busy-waiting condition.

Temporarily handicapping threads is an issue for NetWare 4.x OS since the Low-Priority Queue does not gain control of the CPU unless there is nothing else for the CPU to do. If a thread does "busy waiting", continually rescheduling itself on the Run Queue (by using **ThreadSwitch**), the Low-Priority Queue cannot gain control of the CPU.

Parent Topic: When to Schedule a Routine as a Thread

Thread Function List

Table auto. Thread Services Functions

Function	Purpose
abort	Terminates an NLM abnormally.
atexit	Creates a list of functions that are executed on a last-in, first-out basis when the NLM exits normally or is unloaded.
AtUnload	Registers a function that is called if the NLM is unloaded with the **UNLOAD** console command.
BeginThread	Initiates a new thread within the

	current thread group.
BeginThreadGroup	Establishes a new thread within a new thread group.
Breakpoint	Suspends NLM execution and causes a break into the NetWare Internal Debugger.
ClearNLMDontUnloadFlag	Sets a flag in the NLM header to allow the NLM to be unloaded with the **UNLOAD** console command.
delay	Suspends execution for an interval (milliseconds).
EnterCritSec	Prevents all other threads in the NLM from being scheduled.
exit	Causes the NLM to terminate normally.
exit	Terminates the NLM without executing atexit functions or flushing buffers.
ExitCritSec	Allows other threads in the NLM to run.
ExitThread	Terminates either the current thread or the NLM.
FindNLMHandle	Returns the handle of a loaded NLM.
getcmd	Returns the command line it its original format (unparsed).
getenv	Searches the environment area for the environment variable and returns its value (presently environment variables are not supported).
GetNLMHandle	Returns the handle of the current NLM.
GetNLMID	Returns the ID of the current NLM.
GetNLMNameFromNLMID	Returns the name of an NLM.
GetPrty	Returns the execution priority for a thread (NetWare 3.x).
GetThreadContextSpecifier	Returns the CLIB context used by callback routines scheduled by the specified thread.
GetThreadGroupID	Returns the ID of the current thread group.
GetThreadHandicap	Gets the number of context switches a thread is delayed before being rescheduled.

GetThreadID	Returns the thread ID of the current thread.
GetThreadName	Returns the name of a thread.
longjmp	Restores a saved environment.
main	A developer-supplied function where NLM execution begins.
MapNLMIDToHandle	Returns the handle associated with the NLM ID.
raise	Sends a signal to the executing program.
RenameThread	Renames a thread.
ResumeThread	Allows a previously suspended thread to run.
RetunNLMVersionInfoFromFile	Returns version information for a loaded NLM that corresponds to the specified file.
ReturnNLMVersionInformation	Returns version information for a loaded NLM that corresponds to a specified NLM handle.
ScheduleWorkToDo	Schedules a routine as work, which puts it on the highest priority queue (NetWare 4.x only).
setjmp	Saves its calling environment in its jmp_buf for subsequent use by **longjmp**.
SetNLMDontUnloadFlag	Sets a flag in the NLM header to prevent the NLM from being unloaded with the **UNLOAD** console command.
SetNLMID	Changes the current NLM.
SetPrty	Sets a new execution priority for a thread (NetWare 3.x).
SetThreadContextSpecifier	Determines the CLIB context that is used by all callback routines scheduled by the specified thread (NetWare 4.x only).
SetThreadGroupID	Changes the current thread group.
SetThreadHandicap	Sets the number of context switches a thread is permanently handicapped (delayed) before being rescheduled (NetWare 4.x only).
signal	Specifies an action to take place when certain conditions are detected (signalled).

spawnlp, spawnvp	Creates and executes a new child process.
SuspendThread	Prevents a thread from being scheduled.
system	Used to execute OS commands.
ThreadSwitch	Allows other threads a chance to run, where no natural break in the running thread would normally occur.
ThreadSwitchLowPriority	Reschedules a thread onto the low-priority queue (NetWare 4.x only).
ThreadSwitchWithDelay	Reschedules a thread to be place on the RunList after a specified number of switches have taken place (NetWare 4.x only).
Local Semaphore Functions:	
CloseLocalSemaphore	Closes a local semaphore.
ExamineLocalSemaphore	Returns the current value of a semaphore.
OpenLocalSemaphore	Allocates a local semaphore and gives the NLM access to it.
SignalLocalSemaphore	Increments a semaphore's value.
TimedWaitOnLocalSemaphore	Waits on a local semaphore until it is signalled or the specified timeout elapses.
WaitOnLocalSemaphore	Decrements a semaphore's value.
WaitOnLocalSemaphore	Decrements a semaphore's value

Thread Global Data

Each thread has its own set of data items. The data items are global only within that thread. That is, they have separate values for each thread. The data items of one thread cannot be referenced by another thread.

A thread is the lowest level within an NLM, and its context can consist of the following data items:

Table auto. Data Item Examples

Data Item	Description
asctime, asctime_r	Only allocated if **asctime** is called. The **asctime**

char String Pointer	function returns a char *.
Critical Section Count	Contains the number of outstanding **EnterCritSec** calls against a thread.
ctime, ctime_r, **gmtime, gmtime_r**, and **localtime, localtime_r** tm Structure Pointer	The **ctime, ctime_r**, **gmtime, gmtime_r**, and **localtime, localetime_r** functions return a pointer to a tm structure. Each thread has its own tm structure. The tm structure is allocated only if one of these three functions is called.
errno	Some functions set the *errno* return code to the last error code that was detected.
Last Value from the rand Function	Each thread has its own **seed** value (to start or continue a sequence of random numbers).
NetWareErrno	A NetWare specific error code. Some functions set both *NetWareErrno* and *errno*
stack	This points to the block of memory that **BeginThread** allocated for the thread's stack.
strtok Pointer	The **strtok** function maintains a pointer into the string being parsed.
t_errno	Used with Transport Level Interface (TLI) functions. chapter).
Thread Custom Data Area Pointer and Size	The *threadCustomDataPtr* points to space that the NetWare API allocates to be associated with an individual thread. The *threadCustomDataPtrSize* variable specifies the size (in bytes) of this data.
Suspend Count	This count contains the number of outstanding **SuspendThread** calls against a thread.

Parent Topic: NetWare Global Data

Thread Group Global Data

One instance of the following data items exists for each thread group. Any change that one thread makes to the value of a thread group global data item affects all the threads in the group. All threads within a thread group share the same thread group context.

Table auto. Data Item Examples

Data Item	Description
Current Connection	The current connection number is described in Connection Number and Task Management.
Current Screen	The current screen is the target of screen I/O functions (see Screen Handling).

Current Task	The current task number is described in Connection Number and Task Management.
CWD	Current working directory (see File System).
Current User	The "current user" is the user context used in Directory Services functions.
signal Settings	Signal handler functions are set by the **signal** function. (See **signal** and **raise**.)
stdin, stdout, stderr	These data items are the second-level standard I/O handles (see Stream I/O).
Thread Group Custom Data Area Pointer and Size	The *threadGroupCustomDataPtr* points to space that the NetWare API allocates to be associated with a thread group. The *threadGroupCustomDataPtrSize* variable specifies the size (in bytes) of this data.
umask Flags	These flags are set by the **umask** function (see File System).

Parent Topic: NetWare Global Data

Thread Groups

Each NLM can have more than one thread group, and each thread group may consist of one or more threads, as defined by the programmer. When an NLM is started, it has one thread group that includes the thread that executes the user-supplied **main** function.

Threads are created by the NetWare® API in four ways, which determine the thread group:

By default, a thread is started at the function **main**. This thread belongs to a default thread group.

BeginThread is called, creating a new thread that belongs to the current thread group.

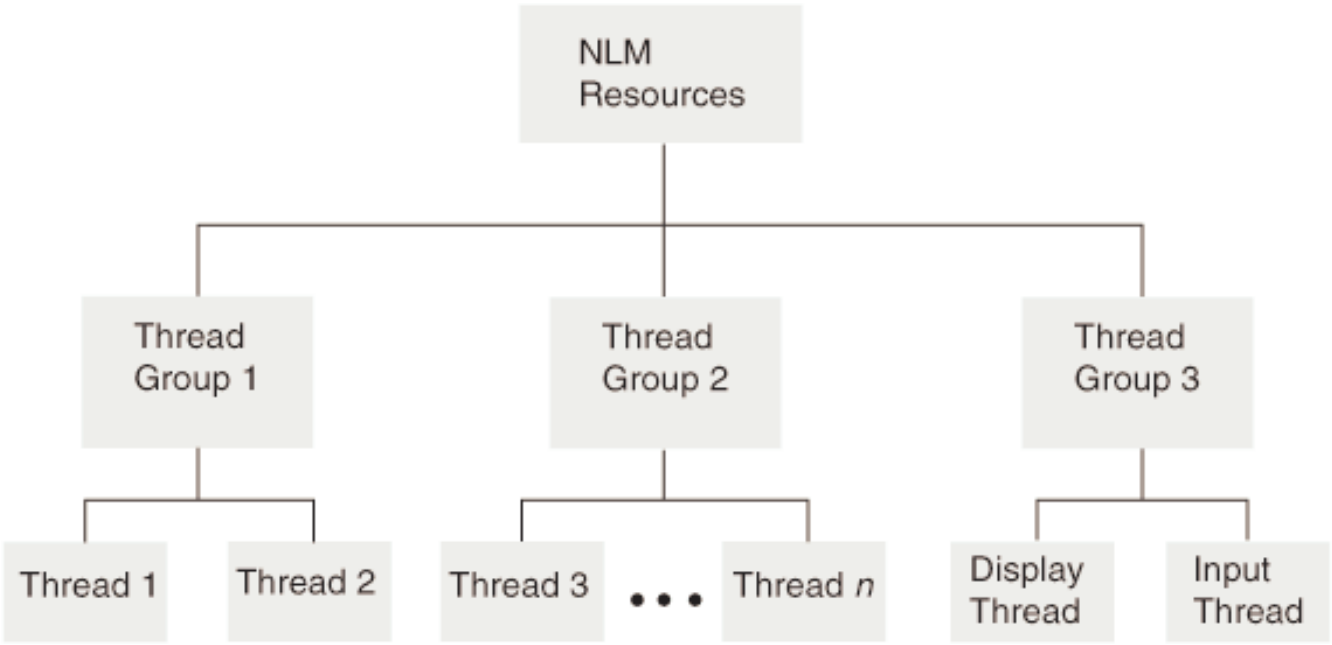
BeginThreadGroup is called, creating a new thread group with one new thread belonging to it.

ScheduleWorkToDo is called, creating a new thread that belongs to the current thread group. **ScheduleWorkToDo** is new to the NetWare 4.x OS.

The following figure shows a sample multithreaded NLM™ configuration. Threads 1 and 2 belong to the same group, Thread Group 1. All other numbered threads belong to Thread Group 2. This means threads 1 and 2 share the same thread group level context information (where their CWD could be \MYDIR1) and threads 3 through *n* share a different thread group level context (where their CWD could be \MYDIR2). Developers must

level context (where their CWD could be \MYDIR2). Developers must understand that when there is more than one thread in a thread group, changing the context (such as CWD) for one thread changes the context for all of the threads in the group. For example, if thread 3 changes its CWD, it also changes the CWD of threads 4 through *n*.

Figure 21. Multithreaded NLM Configuration



Because the display and input threads work together to handle server commands, the two threads have been assigned to the same thread group. This allows them to share the current working directory and current screen, among other resources.

Creating and Terminating Threads

The **BeginThread** function creates a thread. A thread can terminate itself using the **ExitThread** function as follows:

```
ExitThread(EXIT_THREAD, ...)
ExitThread(TSR_THREAD, ...)
```

A return statement from the original function (the function that was started by **BeginThread**) also terminates the thread.

Creating and Terminating Thread Groups

A single thread or multiple threads can be grouped to have a unique context.

The **BeginThreadGroup** function creates a thread group. A thread group can be terminated using the **ExitThread** function as follows:

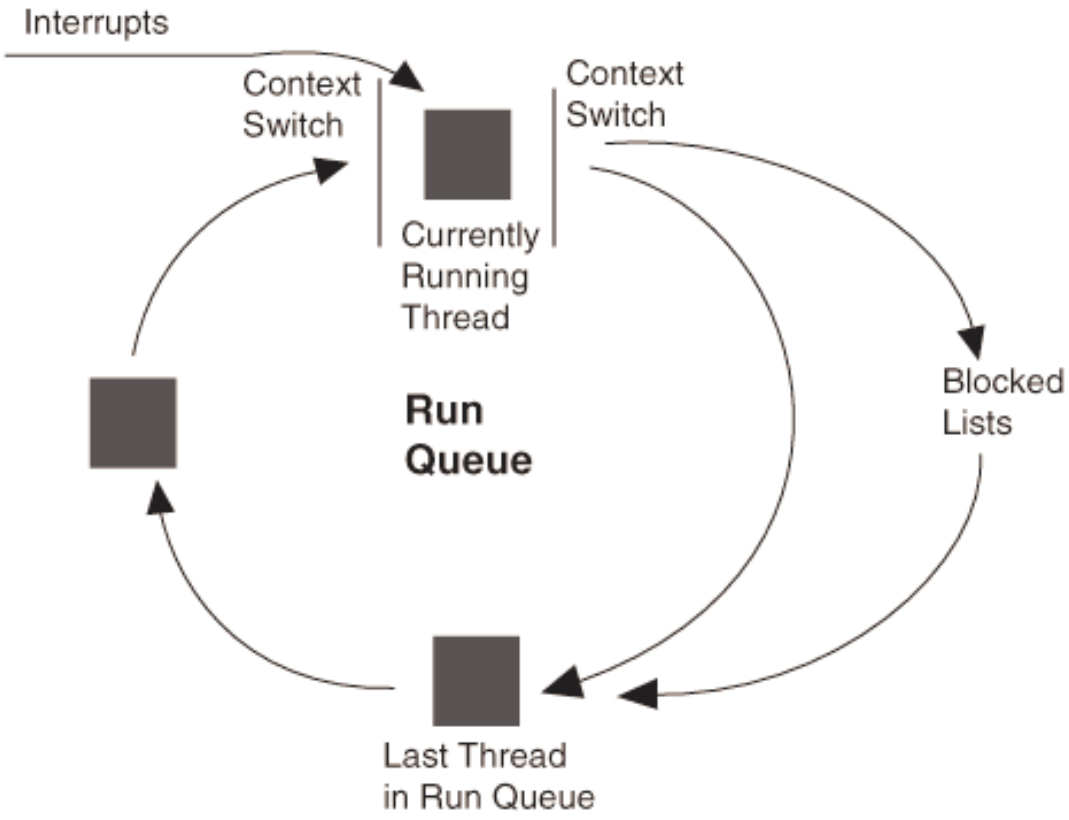
```
ExitThread(EXIT_THREAD, ...)
in the last thread in the group
ExitThread(TSR_THREAD, ...)
in the last thread in the group,
```

A return statement from the original function (the function that was started by **BeginThread**) in the last thread in the group also terminates the thread group.

Thread Management in NetWare 3.x

The thread management for the NetWare 3.x OS is different from that of the NetWare 4.x OS. The following figure illustrates thread management in the NetWare 3.x OS.

Figure 22. NetWare 3.x Thread Management



In the NetWare 3.x OS, threads waiting to be executed are placed in a Run Queue, which is serviced on a FIFO (first in, first out) order.

Threads waiting for resources to become available are placed in a blocked list, yielding the CPU to other threads in the Run Queue. When the needed resources become available, the thread moves from a blocked list to the end of the Run Queue.

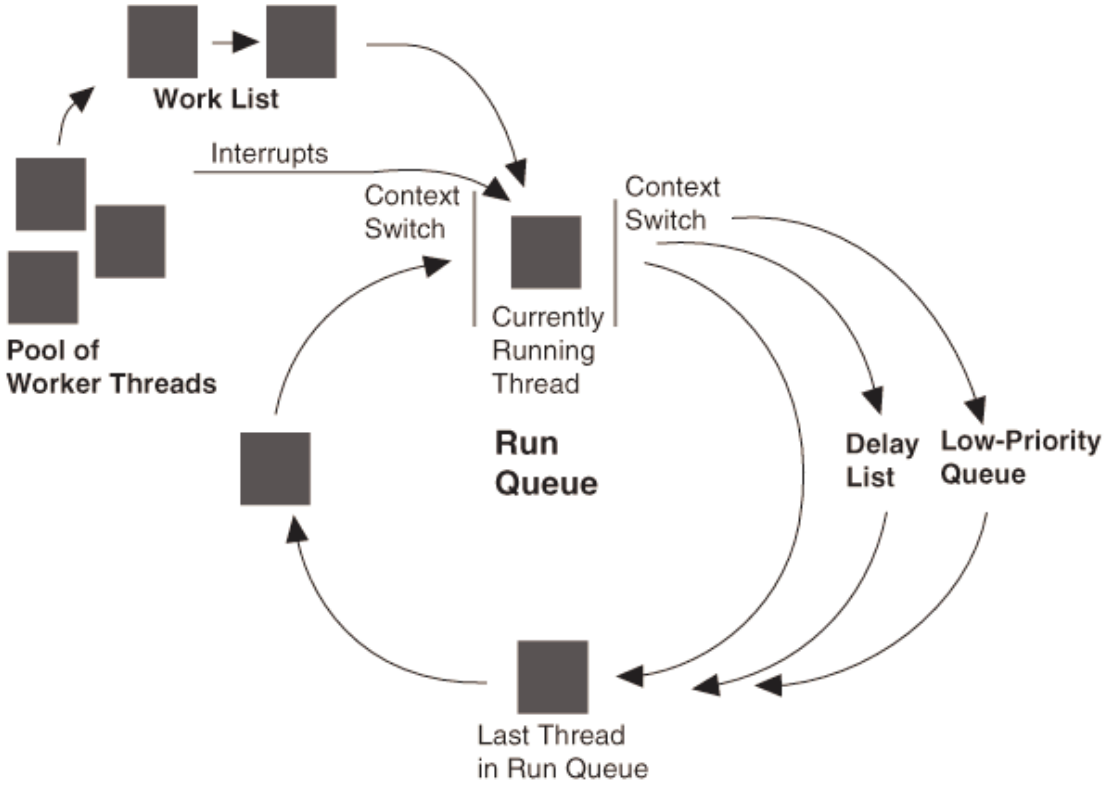
Each time a thread becomes the current (executing) thread, or changes from the current thread to another state, a context switch occurs.

Parent Topic: Threads

Thread Management in NetWare 4.x

The NetWare 4.0 OS introduces new features that add flexibility to thread management. The following figure illustrates thread management in the NetWare 4.x OS.

Figure 23. NetWare 4.x Thread Management



The NetWare 4.0 OS introduces the concept of **work**. Each work unit has a routine and data associated with it, but the work unit is not a thread. To handle these work units, the OS reserves a pool of worker threads that are dedicated to running work.

When a work unit is scheduled, it is placed on the Work To Do List, and is serviced immediately after the current thread relinquishes the CPU. If the current thread is already a worker thread, the worker thread does not relinquish control; instead, it executes the next work unit, thereby avoiding a context switch.

Because worker threads avoid unnecessary context switching, single threads running many separate work units provide higher performance.

> **NOTE:** Novell® recommends that work units be short, discrete routines that can complete quickly. If the work code calls a function that relinquishes control of the CPU, the worker thread is transformed into a regular thread.

Worker threads wait in their own pool. Regular (not worker) threads wait in the Run Queue, the Low-Priority Queue, or the Delay List, queues that are serviced on a FIFO order by the CPU.

The Work To Do List has the highest priority, followed by the Run Queue and the Low-Priority Queue. The Delay List has a variable priority status, and is usually serviced after the Low-Priority Queue has gained control of the CPU.

The Low-Priority Queue does not gain control of the CPU unless there is nothing else for the CPU to do. If a thread does "busy waiting" (looping while waiting for a resource to become available, for example), continually rescheduling itself on the Run Queue, the Low-Priority Queue cannot gain control of the CPU.

Following is an example of busy waiting:

```
while (!finished)
   ThreadSwitchWithDelay();
```

To allow the Low-Priority Queue to be serviced by the CPU, threads that do busy waiting should be rescheduled on the Delay List. After a thread has been scheduled on the Delay List, it waits for a number of context switches (50 is the default), then is placed at the end of the Run Queue. While threads wait in the Delay List, the Low-Priority Queue has a chance to be serviced by the CPU.

As in NetWare 3.x, each time a thread becomes the current thread, or changes from the current thread to another state, a context switch occurs.

For more information about programming with threads, see NLM Code Development.

Parent Topic: Threads

Threads

A **thread** is simply a NetWare kernel process. NetWare kernel processes are referred to as **threads** instead of **processes** on the basis of the following two assumptions:

A process typically saves most of the processor's state when it is swapped out and a thread typically saves less of the processor's state.

Processes are usually preemptive (they take control of all resources) and threads are nonpreemptive. NetWare kernel processes typically match the characteristics of threads; they save only part of the processor's state and are nonpreemptive.

The NetWare OS allows NLM applications to establish multiple threads, each representing a single path of execution. An NLM usually contains at least one thread to accommodate the **main** function. (This is not true if the NLM is a library, such as CLIB.NLM.)

Only one thread can run at a time. While a thread is running, it has control of the CPU. The NetWare OS is a nonpreemptive ("good guy") scheduling environment. When a thread gains control of the CPU, the thread remains in control until it has run to the end of its execution or until it calls a function that 'blocks'---that is, relinquishes control of the CPU. (Blocking functions are identified in the function reference manuals.)

No other thread can interfere with an active thread, regardless of priority. Only a hardware interrupt can temporarily interrupt a currently running thread.

NOTE: The NetWare 4.x OS has the ability to handicap a thread that does not relinquish the CPU in a timely manner. After this "bad" thread yields, the OS does not reschedule it right away; instead, the OS handicaps the "bad" thread, allowing a certain number of other threads to run before the "bad" thread is rescheduled. For this reason, a thread is more responsive in the long run if it takes the initiative to yield control of the CPU often.

Related Topics

Thread Management in NetWare 3.x

Thread Management in NetWare 4.x

When to Schedule a Routine as a Thread

The following conditions serve as guidelines for when to schedule a routine

as a thread:

The routine is a long-term process

It needs a very large stack

It needs to deliberately handicap itself temporarily to avoid **spin-waiting** (being rescheduled while waiting for something needed to complete execution).

If a routine is a long-term process, little benefit results from scheduling it as work because work that yields cannot be rescheduled as work. Instead, it is rescheduled on the Run Queue at the same priority as a normal thread.

All work is given a single stack size; but you can specify a stack size for threads. If you need to specify the stack size, you must schedule your routine as a thread.

If your routine is a polling process or one that does spin-waiting, you should schedule it as a thread.

In general, if your NLM already uses the NetWare 3.x process-scheduling scheme---which can still be carried out in the NetWare 4.x kernel---and its routines are mostly long-term, continue to schedule the routines as threads. But if any of the routines are short-term, you can reschedule them as work.

> **NOTE:** If your NLM is going to run in the NetWare 3.x environment as well as in the NetWare 4.x environment, you cannot schedule any threads as work, since work does not exist in the NetWare 3.x environment. Schedule all threads then as normal threads.

Changing Thread Priority: It is possible to change the priority of a thread in one of the following ways:

Temporarily Handicapping Threads

Permanently Handicapping Threads

Low Priority Threads

When to Schedule a Routine as Work

The following criteria can help determine when you should schedule a routine as work:

The routine has a high priority.

It needs to gain control of the processor quickly, that is, get in and run with as little scheduling overhead as possible.

Examples

A database request routine needs to gain access to the processor quickly and does not need to yield before it is completed. The performance of the database would be enhanced by scheduling requests as **work**, so that the work is serviced quickly and the CPU is relinquished to the next thread.

Similarly, scheduling end-of-routine cleanup as work enhances the operation of all threads in the kernel. A task such as freeing up the stack needs to be executed immediately after a routine ends, and is quick.

Other candidates for work are service routines, which check and update an item regularly. A service routine that updates object information for network management, for example, could be scheduled as work.

In general, NLM applications benefit when lower-level services are scheduled as work. For example, repetitive services such as disk reads could be scheduled as work. NetWare typically does a lot of disk reading. The entire operation usually completes without yielding because the data is found in cache memory.

> **NOTE:** Work is unique to the 4.x OS. If your NLM is going to run on 3.x servers, you cannot schedule threads as work.

Thread: Functions

abort

Terminates an NLM™ application abnormally
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Thread

Syntax

```
#include <assert.h>
#include <nwthread.h>
#include <stdlib.h>

void abort (void);
```

Return Values

None

Remarks

This function causes the NLM to be terminated abnormally. It writes the following termination message to the System Console Screen:

```
ABNORMAL NLM TERMINATION in: NLMname
```

The **abort** function then raises SIGABRT and calls **_exit**(3).

The following sequence of events occurs when the NLM is terminated abnormally:

All threads in the NLM are destroyed.

Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library.)

All screens are closed.

All first-level files (opened with **open**, **sopen**, **create**), including UNIX STREAMS, and also including files opened as a result of second-level I/O (opened with **fopen**, **fdopen**, **freopen**), are closed; however, the buffers of these are not flushed.

All open directories are closed.

All service advertising (started by **AdvertiseService**) is terminated.

All memory allocated by the NLM is freed.

The NLM is unloaded.

See Also

exit, _exit, ExitThread, raise

Example

abort

```
#include <assert.h>
#include <nwthread.h>
#include <stdlib.h>
#include <nwconio.h>

main()
{
   printf("this should print\r\n");
   getch();
   abort();
   printf("this should not print\r\n");
   getch();
}
```

atexit

Creates a list of functions that are executed on a "last-in, first-out" basis when the NLM exits normally or is unloaded ---THIS IS **NOT** AN NLM CLEANUP ROUTINE (See "Remarks" below)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: No

Service: Thread

Syntax

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

int atexit (
   void   (*func) (void));
```

Parameters

func

(IN) Specifies the function to be registered as an **exit** function.

Return Values

0	(0x00)	ESUCCESS	Registration was successful.
-1		EFAILURE	Registration failed (32 functions are already registered).

Remarks

WARNING: The atexit function is designed to be used by drivers. Do not use it as an NLM cleanup routine. A server abend may result for this reason:

CLIB context does not exist for a thread that is running an **atexit** or **AtUnload** routine, since all NLM thread groups have been destroyed by the time these functions are called. Thus any saved thread group ID is invalid, and neither **atexit** nor **AtUnload** routines can use **SetThreadGroupID** to establish CLIB context for the thread. They therefore also cannot use NetWare API functions that require thread group or thread level context.

It is wise programing practice to have one exit point for a program, which could be the cleanup routine for the NLM. The "Remarks" section of **AtUnload** gives suggestions for developing a cleanup routine to be executed when an NLM is manually unloaded. The same routine can be used when **exit** is called from your program.

The **atexit** function is called when an NLM is terminated normally.

Successive calls to **atexit** create a list of functions that are executed on a "last-in, first-out" basis when:

The NLM calls **exit**.

The NLM calls **ExitThread** and it causes the NLM to be terminated.

The last thread in the NLM returns from its original function.

The NLM is unloaded with the **UNLOAD** command.

No more than 32 functions can be registered with **atexit**. The functions have no parameters and do not return values. Such functions can use only NLM (OS) level context.

See Using atexit() functions: Example.

See Also

AtUnload, **exit**, **_exit**, **ExitThread**

AtUnload

Registers a function that is called if the NLM is unloaded with the **UNLOAD** command---THIS IS **NOT** AN NLM CLEANUP ROUTINE (See "Remarks" below)

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

int AtUnload (
   void   (*func) (void));
```

Parameters

func

(IN) Specifies the function to be registered.

Return Values

0	(0x00)	ESUCCESS	Registration was successful.
-1		EFAILURE	Function was already registered.

Remarks

WARNING: AtUnload was designed for use with drivers when an NLM is unloaded with the UNLOAD command. Do not use it as a cleanup routine for NLM applications. CLIB context does not exist for a thread that is running an atexit or AtUnload routine, and a server abend may result.

The cleanup routine to be run at NLM unload time should be registered as a signal handler using **signal** with the condition SIGTERM. This signal handler can use the thread group ID from the main NLM to switch to a CLIB context, allowing it to call any CLIB function. Ensure that the signal handler routine always restores the original thread group ID before exiting. Also take care that the main NLM does not exit before the signal

handler exits. You can do this with a global variable modified by the signal handler, which is monitored by the main NLM before exiting (see **GetThreadGroupID** and **SetThreadGroupID**).

The **AtUnload** function is passed the address of a function to be called when the NLM is unloaded. Such functions can use only NLM (OS) level context.

Only one function can be registered with **AtUnload**.

See Using AtUnload() functions: Example.

See Also

atexit, **exit**, **_exit**, **ExitThread**

BeginThread

Initiates a new thread of execution within the current thread group
Local Servers: blocking
Local Servers: nonblocking
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int BeginThread (
   void       (*func) (void *),
   void       *stack,
   unsigned   stackSize,
   void       *arg);
```

Parameters

func

(IN) Points to the function to execute as the new thread.

stack

(IN) Points to a block of memory to use for the new thread's stack.

stackSize

(IN) Specifies the size (in bytes) of the stack.

arg

(IN) Points to an argument to be passed to the new thread.

Return Values

This function returns the new thread's ID if successful. It returns EFAILURE if an error occurs.

If an error occurs,*errno* is set to:

5	ENOMEM	Not enough memory.
9	EINVAL	Invalid argument was passed in.

Remarks

The new thread begins execution at the specified function (*func*). The function *func* receives *arg* as a parameter. The *stack* parameter is a pointer to a block of memory that the new thread uses as its stack.

If *stack* is NULL, a block of memory (as specified by the *stackSize* parameter) is allocated.

If *stack* is NULL and the specified stack size is too small, the size for the new thread stack is increased automatically.

If *stack* is not NULL and the specified stack size is too small, the function fails and *errno* is set to EINVAL. The minimum stack size for the 3.x OS is 2,064 bytes and 16,384 bytes for the NetWare 4.x OS.

If *stackSize* is zero and stack is NULL, then the default stack size (8,192 bytes for 3.x or 16,384 bytes for 4.x) is used.

The *arg* parameter is any 32-bit quantity, although typically some sort of pointer is passed, or NULL is passed if the specified function does not take any arguments.

If the newly created thread returns from the function *func*, it is be equivalent to its having executed the **ExitThread** function with an action code of EXIT_THREAD.

To begin a thread in a new thread group, call **BeginThreadGroup**.

See Also

BeginThreadGroup, **ExitThread**

Example

BeginThread

```
#include <nwthread.h>

void newThreadFunc (char *funcArg);
int      completionCode;
.
.
.
completionCode = BeginThread (newThreadFunc, NULL, 8192, /A/Q "input.fi
.
.
.
void newThreadFunc (char *arg)
{
   printf ("in new thread\n");
}
```

BeginThreadGroup

Establishes a new thread within a new thread group
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int BeginThreadGroup (
   void       (*func) (void*),
   void       *stack,
   unsigned   stackSize,
   void       *arg);
```

Parameters

func

(IN) Points to the function to execute as the new thread.

stack

(IN) Points to a block of memory to use for the new thread's stack.

stackSize

(IN) Specifies the size (in bytes) of the stack.

arg

(IN) Points to an argument to be passed to the new thread.

Return Values

This function returns the new thread group's ID if successful. It returns EFAILURE if an error occurs.

If an error occurs,*errno* is set to:

5	ENOMEM	Not enough memory.
9	EINVAL	Invalid argument was passed in.

Remarks

The **BeginThreadGroup** function creates a new thread group which contains one thread defined by *func*. Other than putting the new thread in its own thread group, this function is identical to **BeginThread**. A thread group can consist of one or more threads as defined by the programmer, and an NLM can have more than one thread group.

The new thread group is not the current thread group. To create more threads within the new thread group, you must make the new thread group current by calling **SetThreadGroupID** with the thread group ID returned by **BeginThreadGroup**. You can then create more threads within the new thread group by calling **BeginThread** for each additional thread for the new thread group.

The new thread begins execution at the specified function (*func*). The function *func* receives *arg* as a parameter. The *stack* parameter is a pointer to a block of memory that the new thread uses as its stack.

If *stack* is NULL, a block of memory (as specified by the *stackSize* parameter) is allocated.

If *stack* is NULL and the specified stack size is too small, the size for the new thread stack is increased automatically.

If *stack* is not NULL and the specified stack size is too small, the function fails and *errno* is set to EINVAL. The minimum stack size for the 3.x OS is 2,064 bytes and 8,192 bytes for the 4.x OS.

If *stackSize* is zero and stack is NULL, then the default stack size (8,192 bytes for 3.x and for 4.x) is used.

The *arg* parameter is any 32-bit quantity, although typically some sort of pointer is passed, or NULL is passed if the specified function does not take any arguments.

If the newly created thread returns from the function *func*, it is be equivalent to its having executed the **ExitThread** function with an action code of EXIT_THREAD.

See Also

BeginThread

Example

BeginThreadGroup

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

void newThreadFunc (char *funcArg);
```

```
int   completionCode;
.
.
.
completionCode = BeginThreadGroup (newThreadFunc, NULL, 8192, "/A/Q inp
.
.
.
void newThreadFunc(char *arg)
{
   printf ("in new thread group\n");
}
```

Breakpoint

Suspends the execution of an NLM and causes a break into the NetWare Internal Debugger

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

void Breakpoint (
   int   breakFlag);
```

Parameters

breakFlag

(IN) Indicates whether or not to take a breakpoint.

Return Values

None

Remarks

This function causes a breakpoint in the program if *breakFlag* is nonzero. The breakpoint occurs at the instruction following the call to **Breakpoint**. The *breakFlag* parameter can be any nonzero value to cause a breakpoint. The *breakFlag* value is loaded into the EDI register.

NOTE: If your application relies on the EDI register, you should not call **Breakpoint** with a nonzero value. Rather, call **EnterDebugger**, which merely enters the system debugger.

ClearNLMDontUnloadFlag

Sets a flag in the header of an NLM to allow it to be unloaded with the **UNLOAD** command at the system console

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

int ClearNLMDontUnloadFlag (
   int   NLMID);
```

Parameters

NLMID

(IN) The ID of the NLM that is to be made so it can be unloaded from the system console. This ID can be obtained from the **GetNLMID** function.

Return Values

-1		EFAILURE	NLMID was an invalid NLM ID.
0	(0x00)	ESUCCESS	The don't unload flag has been set.

Remarks

This function reverses the effects of the **SetNLMDontUnloadFlag** function.

If **SetNLMDontUnloadFlag** is called, the NLM cannot be unloaded until **ClearNLMDontUnloadFlag** is called.

For more information unloading NLM applications, see CHECK Function .

See Also

SetNLMDontUnloadFlag, **GetNLMID**

Example

See example for **SetNLMDontUnloadFlag**.

CloseLocalSemaphore

Closes a local semaphore
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsemaph.h>

int CloseLocalSemaphore (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle
(IN) Specifies the semaphore handle of an open semaphore.

Return Values

0	(0x00)	ESUCCESS

WARNING: A bad semaphore handle causes the server to abend.

Remarks

This function closes an open semaphore and makes any threads waiting on the semaphore runnable. After this function is called, the semaphore handle is no longer valid and should not be used again.

See Also

ExamineLocalSemaphore, **OpenLocalSemaphore**, **SignalLocalSemaphore**, **TimedWaitOnLocalSemaphore**, **WaitOnLocalSemaphore**

delay

Suspends execution for an interval (milliseconds)
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

void delay (
   unsigned   milliseconds);
```

Parameters

milliseconds

(IN) Specifies the number of milliseconds the calling thread is to be delayed.

Return Values

This function returns no value. If an error occurs, *errno* is set to:

-1	ENOMEM	Not enough memory.

Remarks

The **delay** function puts the calling thread to sleep for the amount of time specified by the *milliseconds* parameter.

The actual time the thread is delayed is rounded to an integral number of system clock ticks.

A thread blocked on delay can be restarted (the **delay** function is cancelled) by calling **ResumeThread**.

NOTE: The actual minimum delay granularity is 1/8 sec., therefore, if the millisecond parameter passed is 2, the actual delay time would be 55 milliseconds.

See Also

EnterCritSec, **SuspendThread**, **ThreadSwitch**

Example

delay

```
#include <nwthread.h>
#include <stdio.h>
#include <nwconio.h>

main()
{
   printf("start");
   delay(10000);
   printf("end");
   getch();
}
```

EnterCritSec

Prevents all other threads in the NLM from being scheduled
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int EnterCritSec (void);
```

Return Values

0	(0x00)	ESUCCESS	Threads were suspended in a critical section.

Remarks

The **EnterCritSec** function suspends all the other threads in the NLM.

After executing **EnterCritSec**, the current thread runs exclusively until it calls **ExitCritSec**. The **ExitCritSec** function makes the other threads in the NLM runnable again.

The **EnterCritSec** function maintains a count of the number of outstanding **EnterCritSec** requests. The count is increased with each **EnterCritSec** function call and decreased with each call to **ExitCritSec**. An equal number of calls to **EnterCritSec** and **ExitCritSec** must be performed to restore normal thread dispatching. This allows calls to **EnterCritSec** and **ExitCritSec** to be nested.

The maximum number of concurrent critical sections is 4 billion.

Since NetWare 3.x and 4.x are nonpreemptive operating systems, NLM applications very rarely need to use this function. The only time it is needed is when a critical section of code calls a function which might relinquish control.

Additionally, use of the **EnterCritSec** function should be avoided in favor of using locks or semaphores.

NOTE: If a new thread is started while the NLM is in a critical section,

the thread is in the critical section.

See Also

ExitCritSec

ExamineLocalSemaphore

Returns the current value of a local semaphore
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsemaph.h>

int ExamineLocalSemaphore (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle

(IN) Specifies the semaphore handle of an open semaphore.

Return Values

If successful, this function returns the current value of a semaphore.

WARNING: If a bad semaphore handle is specified, the server abends.

Remarks

This function returns the current value of a semaphore. The semaphore value is decremented for each **WaitOnLocalSemaphore** and incremented for each **SignalLocalSemaphore**. A positive semaphore value indicates that the thread can access the associated resource. If the semaphore value is zero or negative, the thread must either enter a waiting queue by calling the function **WaitOnLocalSemaphore**, or temporarily abandon its attempt to access the resource.

A semaphore handle is obtained by calling **OpenLocalSemaphore**.

See Also

CloseLocalSemaphore, **OpenLocalSemaphore**, **SignalLocalSemaphore**, **TimedWaitOnLocalSemaphore**, **WaitOnLocalSemaphore**

exit

Causes the NLM to terminate normally
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>
#include <stdlib.h>
#include <unistd.h>

void exit (
   int   status);
```

Parameters

status

(IN) Specifies the NLM's return code. (Currently, the status is ignored.)

Return Values

None

Remarks

The **exit** function causes a normal termination consisting of the following sequence of events:

All threads in the NLM are destroyed.

This function calls the **atexit** functions, which are executed in "last-in, first-out" order.

Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library.)

I/O buffers are flushed, and all second-level files (opened with **fopen**, **fdopen**, **freopen**) are closed. Any files created by **tmpfile** are deleted and purged.

All screens are closed.

All remote sessions are terminated.

All other resources allocated by the NLM are freed.

The NLM is unloaded.

See Also

abort, **atexit**, **_exit**, **ExitThread**, **RegisterLibrary**

_exit

Terminates the NLM without executing **atexit** functions or flushing buffers
Local Servers: blocking
Remote Servers: N/A
Classification: Other
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

void _exit (
   int   status);
```

Parameters

status

(IN) Specifies the NLM's return code (currently, the status is ignored.)

Return Values

This function does not return to its caller.

Remarks

The **_exit** function causes a normal termination consisting of the following sequence of events:

All threads in the NLM are destroyed.

Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library.)

All second-level files (opened with **fopen**, **fdopen**, **freopen**) are closed; however, the buffers of these are not flushed. Any files created by **tmpfile** are deleted and purged.

All screens are closed.

All remote sessions are terminated.

All other resources allocated by the NLM are freed.

The NLM is unloaded.

See Also

abort, exit, ExitThread, RegisterLibrary

Example

_exit

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

int main (int argc, char **argv)
{
   FILE *fp;
   atexit (myFunction);          /* myFunction declared elsewhere */
   fp = fopen (argv[1], "r");
   if (fp == NULL)
   {
      fprintf (stderr, Unable to open '%s'\n, argv[1]);
      _exit (1);
   }
   fclose (fp);
   exit (0);
}
```

ExitCritSec

Allows other threads in the NLM to run
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int ExitCritSec (void);
```

Return Values

0	(0x0 0)	ESUCCESS	Threads were suspended in a critical section.
1 9		EWRNGK ND	One or more threads in the NLM were not in a critical section.

Remarks

The **ExitCritSec** function reverses the effect of the **EnterCritSec** function.

NOTE: If a thread is created (with **BeginThread** or **BeginThreadGroup**) while the NLM is in a critical section, **ExitCritSec** returns EWRNGKND. However, the function still works normally in this case with respect to all of the old threads.

See Also

EnterCritSec, **ResumeThread**

ExitThread

Terminates either the current thread or the NLM
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

void ExitThread (
   int   action_code,
   int   status);
```

Parameters

action_code

(IN) Action code:

TSR_THREAD (-1)---Terminate only the current thread.

EXIT_THREAD (0)---Terminate the current thread; if the current thread is also the last thread, terminate the NLM.

EXIT_NLM (1)---Terminate the entire NLM.

status

(IN) The return code of the NLM (currently, the status is ignored).

Return Values

None

Remarks

The action code determines whether to destroy the current thread or the NLM:

Action code TSR_THREAD terminates only the current thread and should only be used in NLM applications that are libraries (that is, they export symbols).

Action code EXIT_THREAD is used to terminate the current thread. If the current thread is also the only thread of the NLM, the NLM itself is terminated.

Action code EXIT_NLM is equivalent to the **exit** function.

The **ExitThread** function causes a normal NLM termination consisting of the following sequence of events:

All threads in the NLM are destroyed.

This function calls the **atexit** functions, which are executed in "last-in, first-out" order.

Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library.)

I/O buffers are flushed, and all second-level files (opened with **fopen**, **fdopen**, **freopen**) are closed. Any files created by **tmpfile** are deleted and purged.

All screens are closed.

All remote sessions are terminated.

All other resources allocated by the NLM are freed.

The NLM is unloaded.

NOTE: Executing the following statement

```
return (completionCode);
```

from the function (including **main**) where a thread began is equivalent to the following:

```
ExitThread (EXIT_THREAD, completionCode);
```

See Also

abort, **exit**, **_exit**, **RegisterLibrary**

FindNLMHandle

Returns the handle of a loaded NLM
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

unsigned int FindNLMHandle (
   char   *moduleName);
```

Parameters

moduleName

(IN) Specifies the full filename (including the extension) of the NLM whose handle is desired.

Return Values

If the NLM is loaded, its handle is returned. Otherwise, NULL is returned.

Remarks

This function searches the list of loaded NLM applications, comparing their names to the specified *moduleName*. If a match is found, the handle for that NLM is returned.

See Also

GetNLMHandle

Example

FindNLMHandle

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

unsigned int   moduleHandle;
```

```
moduleHandle = FindNLMHandle ("TEST.NLM");
if (NLMHandle == NULL)
   printf ("This NLM is not loaded!\n");
```

getcmd

Returns the command line parameters in their original format (unparsed)
Local Servers: nonblocking
Remote Servers: N/A
Classification: Other
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

char *getcmd (
   char   *originalCmdLine);
```

Parameters

originalCmdLine

(OUT) Points to a buffer into which to copy the command line parameters.

Return Values

The address of *originalCmdLine* is returned if *originalCmdLine* is not NULL. Otherwise, the address of the system's copy of the original command line is returned.

Remarks

The **getcmd** function copies the command line, with the "load" command and the program name removed, into a buffer specified by *originalCmdLine* (if *originalCmdLine* is not NULL).

If *originalCmdLine* is NULL, **getcmd** returns a pointer to the system's copy of the original command line (without the "load" command or the program name). The system's copy should not be written over.

The information is terminated with a \0 character. This provides a method of obtaining the original parameters to a program unchanged (with the white space intact).

This information can also be obtained by examining the *argv* vector of program parameters passed to the **main** function in the program.

See Also

spawnlp, spawnvp

Example

getcmd

```
#include <stdio.h>
#include <string.h>
#include <nwthread.h>

main()
{
   char   originalCmdLine[80];
   char   cmdPtr;
   getcmd(originalCmdLine)
   printf("%s\n",originalCmdLine);
   cmdPtr=getcmd(NULL);
   printf("%s\n", cmdPtr);

/*    If the load command is:
   "load test param1 param2 param3"
   The output is:
   param1 param2 param3
   param1 param2 param3
*/
}
```

getenv

Searches the environment area for the environment variable and returns the environment variable's value (presently, environment variables are not supported)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: No

Service: Thread

Syntax

```
#include <stdlib.h>

char *getenv (
   const char   * varname);
```

Parameters

varname

(IN) Points to an environment variable.

Return Values

This function returns a pointer to the string assigned to the environment variable if found, and NULL if no match was found. Currently, NULL is always returned.

Remarks

The matching is case-insensitive; all lowercase letters are treated as if they were uppercase.

GetNLMHandle

Returns the handle of the current NLM
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

unsigned int GetNLMHandle (void);
```

Return Values

Returns the handle of the current NLM.

Remarks

Ordinarily, the current NLM is the NLM that owns the currently running thread.

See Importing a Function: Example.

See Also

FindNLMHandle

GetNLMID

Returns the ID of the current NLM
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMID (void);
```

Return Values

This function returns the ID of the current NLM.

Remarks

Ordinarily, the current NLM is the NLM that owns the currently running thread. The current NLM identifies which NLM owns any subsequently allocated resources.

NOTE: The current NLM is usually the client even though the client might be executing a library's code.

See Also

SetNLMID

GetNLMIDFromThreadID

Returns the ID of the NLM that the specified thread currently belongs to
Local Servers: nonblocking
Remote Servers: N/A
Classification: 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMIDFromThreadID (
   int    threadID,
   char   *fileName );
```

Parameters

threadID

(IN) Specifies the ID of the thread.

fileName

(IN) Receives the name of the file that the associated NLM was loaded from.

Return Values

If successful, **GetNLMIDFromThreadID** returns the ID of the NLM that the thread is associated with. On failure, it returns EFAILURE and*errno* is set to EBADHNDL.

Remarks

An NLM, such as a library NLM, can take over ownership of another NLM's threads by calling **SetNLMID** or **SetThreadGroupID**. However, an NLM must ensure that it owns all of its threads before unloading.

An NLM can keep track of the IDs of the threads it originates and can use **GetNLMIDFromThreadID** to determine if it still owns the threads. If at unload time, an NLM determines that it does not own its threads, the NLM must wait until ownership of the threads is returned to it. Then it can safely unload.

GetNLMIDFromThreadID returns the NLM ID only for threads that have CLIB context. This function returns EFAILURE if passed the ID of a thread that is running as an OS thread.

An example of an OS thread is a procedure scheduled with **ScheduleSleepAESProcessEvent** and with the registering thread's context specifier set to NO_CONTEXT. The registered thread does not have CLIB context when it runs.

NOTE: The interface to this function might change in the next release of the NetWare API to also return the name of the NLM. Currently, the name it returns is the name of the file that the NLM was loaded from. (An NLM can have a different name from its file name.)

See Also

GetThreadID

GetNLMNameFromNLMID

Returns the name of a C Library NLM
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMNameFromNLMID (
   int    NLMID,
   char   *NLMFileName,
   char   *NLMName);
```

Parameters

NLMID

(IN) Specifies an NLM ID.

NLMFileName

(OUT) Points to the NLM filename used in the linker file.

NLMName

(OUT) Points to the descriptive name of the NLM.

Return Values

This function returns the NLM name. If an invalid NLM ID is passed, it returns a value of -1 and *errno* is set to EBADHNDL.

Remarks

This function returns the long name (as it appears on the module listing) and the short name (as specified in the NAME directive in the linker directive file) of the NLM. For example, if you specify the ID for CLIB.NLM for the *NLMID* parameter, on return *NLMFileName* points to `CLIB.NLM` and *NLMName* points to `NLM`.

See Also

MapNLMIDToHandle

GetPrty

Discontinued; see **GetThreadHandicap**

GetThreadContextSpecifier

Returns the CLIB context that is used by callback routines scheduled by the specified thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadContextSpecifier (
   int   threadID);
```

Parameters

threadID

(IN) Gives the ID of the thread whose context specifier you want to get.

Return Values

-1		EFAILURE
0	(0x00)	NO_CONTEXT
1	(0x01)	USE_CURRENT_CONTEXT
Other values are valid thread group IDs.		

Remarks

Many of the functions that are registered as callbacks run as OS threads. These threads need CLIB context to use the NetWare API functions. The function **SetThreadContextSpecifier** can be set to give these threads context when the callbacks are registered. This function lets you find out what those settings were.

If additional callbacks are registered, their context run as part of the thread group that corresponds to the thread group ID that is returned by this function.

For more information about CLIB context, seeNLM Code Development.

See Also

SetThreadContextSpecifier

GetThreadGroupID

Returns the ID of the current thread group
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadGroupID (void);
```

Return Values

This function returns the ID of the current thread group.

Remarks

Ordinarily, the current thread group is the thread group that the currently running thread belongs to. The current thread group identifies which thread group's current connection, current task, current screen, and so on is active. (See Thread Groups.)

See Also

SetThreadGroupID

GetThreadHandicap

Gets the number of context switches a thread is delayed before being rescheduled

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwthread.h>

LONG GetThreadHandicap (
   int   threadID);
```

Parameters

threadID

(IN) Specifies a thread ID.

Return Values

This function returns the current handicap for the specified thread.

Remarks

A context switch is the task switching enacted by the OS when swapping the current thread.

See Also

ThreadSwitchWithDelay

GetThreadID

Returns the thread ID of the currently executing thread
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadID (void);
```

Return Values

The thread ID is returned.

Remarks

This function is used to get a thread ID for those functions which require a thread ID.

See Also

GetNLMID, **GetThreadGroupID**, **SuspendThread**

GetThreadName

Returns the name of a C Library thread
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadName (
   int    threadID,
   char   *tName);
```

Parameters

threadID

(IN) Specifies a thread ID.

tName

(OUT) Specified the name of the thread.

Return Values

This function returns the name of a thread. If an invalid thread ID is passed, it returns an EBADHNDL error.

Remarks

This function returns the name of the specified C Library thread in *tName*. The *tName* parameter can hold up to 17+1 characters.

See Also

GetThreadID, **RenameThread**

longjmp

Restores a saved environment
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Thread

Syntax

```
#include <setjmp.h>

void longjmp (
   jmp_buf   env,
   int       value);
```

Parameters

env

(IN) Specifies the environment to be restored.

value

(IN) Specifies the value to return.

Return Values

This function returns no value.

Remarks

The **longjmp** function restores the environment saved by the most recent call to **setjmp** with the corresponding *jmp_buf* argument. After **longjmp** restores the environment, program execution continues as if the corresponding call to **setjmp** has just returned the value specified by *value*.

The **setjmp** function must be called before **longjmp**. The routine that called **setjmp** and set up *env* must still be active and cannot have returned before **longjmp** is called. If this happens, the results are unpredictable. If *value* is 0, the value returned is 1.

See Also

setjmp

main

A user-supplied function where NLM execution begins
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int main (
   int          argc,
   const char   *argv[]);

int main (void);

void main (
   int          argc,
   const char   *argv[]);

void main (void);
```

Parameters

argc

(IN) Specifies the number of arguments on the command line.

argv

(IN) Points to the array of command line arguments pointers.

Syntax

The syntax for the **main** function can be any of the following:

Return Values

Currently, the return code from **main** is ignored.

Remarks

The **main** function is a user-written function that is executed as the initial thread of the NLM.

Prior to the **main** function receiving control, the **_Prelude** function (in

PRELUDE.OBJ) does the following:

The current connection is set to 0 and a unique task number is allocated.

A new screen is created. The screen name is the name specified by the linker directive SCREENNAME. If the screen name is not specified, the description text specified in the FORMAT directive is used as the screen name. If the screen name is "none," "default," or "System Console," no new screen is created.

A new thread is started with the specified stack size. If no stack size is specified, then the default stack size (8192 bytes) is used.

The thread's name is the name specified by the linker directive THREADNAME. The thread name can be up to 16 characters long. The first thread name is generated by appending "0" to the specified thread name, the second by appending "1", and so on. If the thread name is not specified, the name specified with the linker directive NAME (with .NLM appended) is used as the pattern for generating thread names.

If the **main** function returns with a return code of rc, it is equivalent to its executing **ExitThread** (EXIT_THREAD, rc). See the discussion of the **ExitThread** function.

The command line to the program is assumed to be a sequence of tokens separated by blanks. The tokens are passed to the **main** function as an array of pointers to character strings in the *argv* parameter. The first element of *argv* is a pointer to a character string containing the program name, including the full path. The last element of the array pointed to by *argv* is a NULL pointer (*argv[argc]* is NULL). Arguments that contain blanks can be passed to the main function by enclosing them within double quote characters (which are removed from that element in the *argv* vector).

The command line can also be obtained in its original format by using the **getcmd** function.

See Also

abort, **_exit**, **exit**, **ExitThread**

Example

main

```
#include <nwthread.h>

int main (int argc, char *argv[])
{
   /* Do the work */
```

```
    .
    .
    .
    /* Terminate thread and NLM */
    return 0;
}
```

MapNLMIDToHandle

Returns the handle associated with the NLM ID
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int MapNLMIDToHandle (
   int   NLMID);
```

Parameters

NLMID
(IN) Specifies an NLM ID (C Library structure).

Return Values

This function returns the handle associated with the specified NLM ID. It returns a value of -1 if an invalid NLM ID was passed.

Remarks

This function can be used in registered LOAD and UNLOAD event handlers to compare known C Library NLM IDs with the NLM handle that the event handler function is passed with.

See Also

FindNLMHandle, **GetNLMID**, **RegisterForEvent**

NWSMPIsLoaded

Returns whether the SMP kernel is loaded
Local Servers: N/A
Remote Servers: N/A
Classification: 4.x SMP
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsmp.h>

int NWSMPIsLoaded (
   void);
```

Return Values

TRUE	Returned if the SMP kernel is loaded
FALSE	Returned if the SMP kernel is not loaded

Remarks

The SMP kernel is loaded by loading smp.nlm in the startup.ncf file of a 4.x file server.

The value returned by **NWSMPIsLoaded** is not an indicator that more than one processor has been brought online. It is only an indicator the kernel is loaded.

MPDRIVER.NLM must be loaded in order to have multiple processors available to the SMP kernel.

NWThreadToMP

Migrates a NetWare thread from the Netware kernel to the SMP kernel
Local Servers: N/A
Remote Servers: N/A
Classification: 4.x SMP
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsmp.h>

void NWThreadToMP (
   void);
```

Return Values

None

Remarks

NWThreadToMP takes the currently running thread that was created by **BeginThread**, **BeginThreadGroup**, or **ScheduleWorkToDo** and migrates it to the SMP kernel. The SMP kernel then owns the thread scheduling and state context switches for that thread.

A thread migrated to the SMP kernel is still capable of having a CLib context. You can use the same CLib context functions to manipulate the CLib context.

Should an SMP thread call **NWThreadToMP**, the thread remains an SMP thread.

NWThreadToMP is exported by THREADS.NLM of the CLib suite.

See Also

NWThreadToNetWare

NWThreadToNetWare

Migrates an SMP thread managed by the SMP kernel to the NetWare kernel as a NetWare thread

Local Servers: N/A

Remote Servers: N/A

Classification: 4.x SMP

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwsmp.h>

void NWThreadToNetWare (
   void);
```

Return Values

None

Remarks

NWThreadToNetWare takes a thread previously migrated to the SMP kernel, and puts it back on the NetWare kernel as a NetWare thread on the end of the run queue. A NetWare thread that calls **NWThreadToNetWare** will remain on the NetWare run queue.

NWThreadToNetWare is exported by THREADS.NLM of the CLib suite.

SMP threads are created by first creating a NetWare thread by calling **BeginThread**, **BeginThreadGroup**, or **ScheduleWorkToDo** and then migrating the thread to the SMP kernel by calling **NWThreadToMP**.

See Also

NWThreadToMP

OpenLocalSemaphore

Allocates a local semaphore and gives the NLM access to it
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsemaph.h>

LONG OpenLocalSemaphore (
   LONG   initialValue);
```

Parameters

initialValue

(IN) Specifies the value to assign the semaphore.

Return Values

This function returns the semaphore handle.

Remarks

This function creates and initializes the semaphore to *initialValue*. The *initialValue* parameter indicates the number of threads that can access the resource at a time. A call to **SignalLocalSemaphore** increments this value. A call to **WaitOnLocalSemaphore** decrements this value.

WARNING: Developers must make sure to close local semaphores that are opened in an NLM because they are not automatically closed when an NLM unloads. If a local semaphore is opened but not closed before the NLM unloads, the server abends.

See Also

CloseLocalSemaphore, **ExamineLocalSemaphore**, **SignalLocalSemaphore**, **TimedWaitOnLocalSemaphore**, **WaitOnLocalSemaphore**

raise

Sends a signal to the executing program
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: Yes
Service: Thread

Syntax

```
#include <signal.h>

int raise (
   int   condition);
```

Parameters

condition

(IN) Specifies the condition for which to raise a signal.

Return Values

This function returns a value of 0 when the condition is successfully raised and a nonzero value otherwise.

Remarks

The **raise** function signals the exception condition indicated by the *condition* parameter. The **signal** function can be used to specify the action to take place when a signal is raised. See the discussion of the **signal** function for a list of the possible conditions.

There can be no return of control following a call to the **raise** function if the action for that condition is to terminate the program or to transfer control using **longjmp** .

See Also

longjmp, **signal**

RenameThread

Renames a C Library thread
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.11, 3.12, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int RenameThread (
   int    threadID,
   char   *newName);
```

Parameters

threadID

(IN) Specifies a thread ID.

newName

(IN) Specifies the new thread name.

Return Values

This function returns ESUCCESS if it completes successfully. It returns EBADHNDL if an invalid thread ID is passed.

Remarks

This function renames a C Library thread. The new name can be up to 17 characters long.

ResumeThread

Allows a previously suspended thread to run
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int ResumeThread (
   int   threadID);
```

Parameters

threadID

(IN) Specifies the ID of the thread to be resumed.

Return Values

0	(0x00)	ESUCCESS	Thread was resumed.
9		EINVAL	Thread tried to resume itself.
19		EWRNGKND	Thread was not suspended.
22	(0x16)	EBADHNDL	Bad thread ID was passed in.

Remarks

The **ResumeThread** function reverses the effect of **SuspendThread**.

CAUTION: Your application must be aware of how its threads are suspended and must call ResumeThread only when it is appropriate. For example, if your application suspends a thread, it is appropriate for your application to resume the thread. However, it is not appropriate for your application to call ResumeThread for one of its threads that has been suspended by the OS, while the thread is running in OS code. Calling ResumeThread at an inappropriate time will cause unpredictable behavior.

See Also

ExitCritSec, **SuspendThread**

ReturnNLMVersionInfoFromFile

Returns version information for a loaded NLM that corresponds to a specified file

Local Servers: blocking

Remote Servers: blocking

Classification: 3.12, 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

int ReturnNLMVersionInfoFromFile (
   BYTE   *__pathName
   LONG   *majorVersion,
   LONG   *minorVersion,
   LONG   *revision,
   LONG   *year,
   LONG   *month,
   LONG   *day,
   BYTE   *copyrightString
   BYTE   *description);
```

Parameters

__pathName

(IN) Specifies the path to the NLM file whose version information is to be returned.

majorVersion

(OUT) Receives the major version number of the NLM.

minorVersion

(OUT) Receives the minor version number of the NLM.

revision

(OUT) Receives the revision number of the NLM.

year

(OUT) Receives the number of the year that the NLM was created.

month

(OUT) Receives the number of the month that the NLM was created.

day

(OUT) Receives the number of the day that the NLM was created.

copyrightString

(OUT) Points to a buffer that receives an ASCIIZ string containing the copyright string of the NLM. Buffer size should be 256 bytes.

description

(OUT) Points to a buffer that receives an ASCIIZ string containing the name that is displayed when the NLM is loaded. Buffer size should be 128 bytes.

Return Values

-1	EFAILURE	An invalid NLM handle was specified.
0	ESUCCESS	

Remarks

While *__pathName* must be supplied, the other parameters can be set to NULL if you do not want the information they return.

The NLM specified by *__pathName* does not need to be running for this function to retrieve its information.

The information for *majorVersion*, *minorVersion*, *revision*, *copyrightString*, and *description* are set with linker options when the NLM applications are linked. For more information about the linker options, see NLM Linkers.

The buffer *description* points to should be at least 128 bytes.

For the NetWare 3.11 OS, this function was made available in CLIB.NLM, version 3.11b.

See Also

MapNLMIDToHandle, **FindNLMHandle**, **ReturnNLMVersionInformation**

ReturnNLMVersionInformation

Returns version information for a loaded NLM that corresponds a specified NLM handle

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

int ReturnNLMVersionInformation (
   int    NLMHandle,
   LONG   *majorVersion,
   LONG   *minorVersion,
   LONG   *revision,
   LONG   *year,
   LONG   *month,
   LONG   *day,
   BYTE   *copyrightString,
   BYTE   *description);
```

Parameters

NLMHandle

(IN) Gives the handle of the NLM for which to return version information. This handle can be obtained by calling **FindNLMHandle** or **MapNLMIDToHandle**.

majorVersion

(OUT) Receives the major version number of the NLM.

minorVersion

(OUT) Receives the minor version number of the NLM.

revision

(OUT) Receives the revision number of the NLM.

year

(OUT) Receives the number of the year that the NLM was created.

month

(OUT) Receives the number of the month that the NLM was created.

day

(OUT) Receives the number of the day that the NLM was created.

copyrightString

(OUT) Points to a buffer that receives an ASCIIZ string containing the copyright string of the NLM. Buffer size should be 256 bytes.

description

(OUT) Points to a buffer that receives an ASCIIZ string containing the name that is displayed when the NLM is loaded. Buffer size should be 128 bytes.

Return Values

-1	EFAILURE	An invalid NLM handle was specified.
0	ESUCCESS	

Remarks

While *NLMHandle* must be supplied, the other parameters can be set to NULL if you do not want the information they return.

The information for *majorVersion*, *minorVersion*, *revision*, *copyrightString*, and *description* are set with linker options when the NLM applications are linked. For more information about the linker options, see NLM Linkers.

For the NetWare 3.11 OS, this function was made available in CLIB.NLM, version 3.11b.

See Also

MapNLMIDToHandle, **FindNLMHandle**, **ReturnNLMVersionInfoFromFile**

ScheduleWorkToDo

Schedules a routine as work, which puts it on the highest priority queue, the WorkToDoList

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwthread.h>

int ScheduleWorkToDo (
   void        (*ProcedureToCall) ( ) ,
   void        *workData,
   WorkToDo    *workToDo);
```

Parameters

ProcedureToCall

(IN) Points to the routine being scheduled as work.

workData

(IN) Points to the data to be passed to the worker thread.

workToDo

(IN) Points to a WorkToDo structure.

Return Values

0	Success
5	ENOMEM

Remarks

This function schedules work to be executed by an OS worker thread.

The *ProcedureToCall* parameter points to the procedure to be scheduled as work. Work is a high-priority, low overhead procedure. See When to Schedule a Routine as Work.

The *workData* parameter contains the data to be passed to *ProcedureToCall*.

The *workToDo* parameter is a structure used by CLIB.NLM to set up WorkToDo process scheduling. The structure must be allocated before calling this function and released afterward (for example, by calling **malloc** and **free**). Other than allocating *workToDo*, the developer does not need to be concerned with the details of this stucture since the only user-defined fields in the structure are set by the *ProcedureToCall* and *workData* parameters. WorkToDo is defined in nwthread.h.

Since the work that is scheduled is done by an OS worker thread, it is not be able to use the NetWare API functions that use context, unless context is given to the OS worker thread.

The context that is given to the OS worker thread 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.

See Also

BeginThread, **BeginThreadGroup**

Example

ScheduleWorkToDo

```
#include <stdio.h>
#include <nwthread.h>
#include <nwconio.h>

int   count = 0;

/*.....................*/
void ScreenUpdater(void *data)
{
   data = data;
   count++;
   clrscr();
   printf("You could use a work to do thread\n\n");
   printf("to do screen updates.  %i.\n\n\n", count);
   printf("Work to do threads get into the\n\n");
   printf("system fast and are useful to\n\n");
   printf("accomplish finite definable tasks.");
}

/*.....................*/
main()
{
   WorkToDo   screenWork;
   char       ch = 0;
   SetAutoScreenDestructionMode(TRUE);
   while (ch != 'q')
   {
      ScheduleWorkToDo(ScreenUpdater, NULL, &screenWork);

      /* ThreadSwitch makes sure work to do gets a chance to run. */
      ThreadSwitch();
      if (!kbhit())
      ThreadSwitchWithDelay();
      else
      ch = getch();
   }
}
/*.....................*/
```

setjmp

Saves its calling environment in its *env* parameter for subsequent use by the **longjmp** function

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Thread

Syntax

```
#include <setjmp.h>

int setjmp (
   jmp_buf   env);
```

Parameters

env

(OUT) Specifies the buffer in which to save environment.

Return Values

This function returns a value of 0 when it is initially called. The return value is nonzero if the return is the result of a call to the **longjmp** function. An if statement is often used to handle these two returns. When the return value is 0, the initial call to **setjmp** has been made; when the return value is nonzero, a return from a **longjmp** has just occurred.

Remarks

In some cases, error handling can be implemented by using **setjmp** to record the point to which a return occurs following an error. When an error is detected in a called function, that function uses **longjmp** to jump back to the recorded position. The original function which called **setjmp** must still be active (it cannot have returned to the function which called it).

Special care must be exercised to ensure that any side effects that have occurred (such as allocated memory and opened files) are satisfactorily handled.

See Also

longjmp

SetNLMDontUnloadFlag

Sets a flag in the header of an NLM to prevent the NLM from being unloaded with the **UNLOAD** command at the system console

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

int SetNLMDontUnloadFlag (
   int   NLMID);
```

Parameters

NLMID

(IN) The ID of the NLM that is to be made so it cannot be unload from the command line.

Return Values

-1		EFAILURE	NLMID was an invalid NLM ID.
0	(0x00)	ESUCCESS	The don't unload flag has been set.

Remarks

A console operator can unload an NLM from the system console command line by issuing the following command:

```
UNLOAD NLM_NAME
```

where NLM_NAME is the name of the NLM being unloaded.

If there is a check function for the NLM (declared with the OPTION CHECK directive), it is called when the "UNLOAD NLM_NAME" command is entered. This function then must decide if it is safe to unload the NLM. If it is safe to unload the NLM, the function returns 0 and the NLM is unloaded. If the function determines that the NLM should not be

unloaded, it returns a nonzero value and the following prompt is displayed on the system console:

```
UNLOAD module anyway? n
```

In this case, the console operation can choose to unload the NLM anyway, by pressing the "y" key, instead of the "n" key.

If **SetNLMDontUnloadFlag** is called, the NLM can only be unloaded after **ClearNLMDontUnloadFlag** is called.

For more information about unloading NLM applications, see CHECK Function.

See Also

ClearNLMDontUnloadFlag, **GetNLMID**

Example

SetNLMDontUnloadFlag

```
#include <nwconio.h>
#include <errno.h>
#include <stdio.h>
#include <nwthread.h>

main()
{
   int NLMID, result;
   NLMID=GetNLMID();
   result=SetNLMDontUnloadFlag(NLMID);
   if(result==ESUCCESS)
   {
      printf("DONTUNLD.NLM cannot be unloaded now.\n");
      printf("Press any key to be able to unload this NLM\n");
      getch();
      ClearNLMDontUnloadFlag(NLMID);
      printf("\nYou can unload DONTUNLD.NLM now.\n");
      getch();
   }
   else
      printf("Could not set the don't unload flag.\n");
}
```

SetNLMID

Chanages the current NLM ID
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int SetNLMID (
   int   newNLMID);
```

Parameters

newNLMID

(IN) Specifies the ID of NLM to make current (returned by a previous call to **SetNLMID** or **GetNLMID**.

Return Values

This function returns the ID of the previously current NLM if successful. Otherwise, it returns EFAILURE and sets*errno* to:

22	(0x16)	EBADHNDL	Invalid NLM ID was passed in.

Remarks

The current NLM determines which NLM "owns" resources that are subsequently allocated. (See Connection Number and Task Management for a discussion of resources.)

The main implication of "ownership" of resources is the automatic cleanup performed by the NetWare API when an NLM terminates. In the case of a library/client relationship:

If a library allocates resources while being called by a client, by default (without calling **SetNLMID**), the resources are owned by the client.

If the library calls **SetNLMID** to make itself the current NLM and then allocates resources, the library owns the resources.

NOTE: A library should save and restore the client's NLM ID when it changes the current NLM ID.

See Also

GetNLMID

SetPrty

Discontinued; see **SetThreadHandicap**

SetThreadContextSpecifier

Determines the CLIB context that is to be used by all callback routines scheduled by the specified thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: No

Service: Thread

Syntax

```
#include <nwthread.h>

int SetThreadContextSpecifier(
   int   threadID,
   int   contextSpecifier);
```

Parameters

threadID

(IN) Gives the ID of the thread whose context specifier you want to get.

contextSpecifier

(IN) Gives the context to give callback threads.

Return Values

-1		EFAILURE
0	(0x00)	ESUCCESS

Remarks

Many of the functions that are registered as callbacks run as OS threads. These threads need CLIB context to use the NetWare API functions, such as **printf**. This function is used to determine what context is given to callbacks when they are registered by the calling thread.

The default setting is for callbacks to have the context of the thread that calls them.

The thread context specifier is set on a per-thread basis. Changing the

context specifier for one thread does not change it for any of the other threads. The *threadID* parameter specifies which thread should have its context specifier set.

The *contextSpecifier* parameter tells what the context should be. It can be one of the following:

NO_CONTEXT---Do not give CLIB context to callback functions when they are registered. You would use this option when your callback is not going to use any NetWare API functions other than local semaphore calls and **SetThreadGroupID**, which then creates context. You could also use this if you want to manually set the callbacks function with a call to **SetThreadGroupID**.

USE_CURRENT_CONTEXT---Set the context of the callback being scheduled to be the same as the thread that is scheduling the callback. This is the default setting that exists when a new thread is started.

A valid thread group ID---Set the context of the callback to this thread group ID. The ID of the current thread group can be returned with a call to **GetThreadGroupID**.

See Also

GetThreadContextSpecifier, **SetThreadGroupID**, **GetThreadGroupID**

SetThreadGroupID

Changes the thread group ID of the running thread
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int SetThreadGroupID (
   int   newThreadGroupID);
```

Parameters

newThreadGroupID

(IN) Specifies the ID of the thread group to make current (returned by a previous call to **SetThreadGroupID** or **GetThreadGroupID**.

Return Values

This function returns the ID of the previously current thread group if successful. Otherwise, it returns EFAILURE and sets*errno* to:

22	(0x16)	EBADHNDL	Invalid thread group ID was passed in.

Remarks

The **SetThreadGroupID** function determines which instance of the current connection, current task, current screen, and so on, is used. (See Thread Groups.) Since a thread group is owned by a particular NLM, this function also sets the current NLM ID to the NLM that owns the thread group that is being made current.

NOTE: A library should save and restore the thread group ID whenever it changes the current thread group.

See Also

GetThreadGroupID, **SetNLMID**

Example

SetThreadGroupID

```
#include <nwthread.h>

int   currentThreadGroupID;
int   newThreadGroupID;
currentThreadGroupID = SetThreadGroupID (newThreadGroupID);
```

SetThreadHandicap

Sets the number of context switches a thread is permanently handicapped (delayed) before being rescheduled

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwthread.h>

void SetThreadHandicap (
   int   threadID,
   int   handicap);
```

Parameters

threadID

(IN) Specifies a thread ID.

handicap

(IN) Specifies the number of context switches the thread waits before being put on the Run Queue.

Return Values

None

Remarks

This function sets the value used to determine the number of context switches a thread waits before being put on the Run Queue. This sets the permanent handicap.

See Also

GetThreadHandicap, **ThreadSwitchWithDelay**

signal

Specifies an action to take place when certain conditions are detected (signalled) while a program executes

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: Yes

Service: Thread

Syntax

```
#include <signal.h>

void (*signal (
   int    sig,
   void   (*func) (
      int)))
   (int);
```

Parameters

sig

(IN) Specifies the condition being signalled.

func

(IN) Points to the function to be called when the signalled condition occurs.

Return Values

Returns the previous setting if successful or SIG_ERR if a failure occurred.

Remarks

signal is used to specify an action to take place when certain conditions are detected while a program executes. These conditions are defined to be:

SIGABRT	Abnormal termination, caused by the **abort** function.
SIGFPE	An erroneous floating-point operation occurs, such as division by zero, overflow and underflow (supported only for compiler option / fpc; not supported for options /fpi, /7, /fpi87).

SIGILL	An illegal instruction is encountered. (Currently not supported.)
SIGINT	Raised if the Ctrl+C keys are pressed during screen output (other than to the System Console Screen).
SIGSEGV	An illegal memory reference is detected. (Currently not supported.)
SIGTERM	An UNLOAD command has been entered for the NL.

The *func* parameter is used to specify an action to take for the specified condition:

When the *func* parameter is a function name, that function is called equivalently to the following code sequence.

```
   /*"sig" is the condition being signalled*/
signal (sig, SIG_DFL);
(*func) (sig);
```

The function specified by the *func* parameter can terminate the program by calling the **exit**, **_exit**, **ExitThread**, or **abort** functions. It can also call the **longjmp** function or it can return. Because the next signal is handled with default handling, the program must again call **signal** if it is desired to handle the next condition of the type that has been signalled.

NOTE: The **exit**, **_exit**, **ExitThread**, and **abort** functions cannot be called from the context of a SIGTERM handler or the server console will be inoperational.

A registered SIGTERM signal handler in NetWare 3.11 and 4.x is on a per-NLM basis.You only have to register your SIGTERM handler once for the NLM. The other signals are on a per-threadgroup basis. For these signals, you have to register your signal handler every time you start a new thread group (by calling the **BeginThreadGroup** function). If not, your signal handler is not called.

SIG_IGN	This value causes the indicated condition to be ignored.
SIG_DFL	This value causes the default action for the condition to occur.

The initial settings for the NetWare API are as follows:

```
SIGABRT   SIG_DFL
SIGFPE    SIG_IGN
SIGILL    SIG_IGN
SIGINT    SIG_DFL
SIGSEGV   SIG_IGN
SIGTERM   SIG_DFL
```

A condition can be generated by a program by calling the **raise** function.

The default action for the SIGABRT action is to call **_exit**(3). The default action for SIGINT is to call **abort**(). The default action for the other conditions is to ignore the condition.

The functions registered with **signal** run as callbacks, so CLIB context is an issue. If a callback does not have CLIB context, it cannot make calls to the NetWare API functions that require context.

The functions registered for SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM have the thread group context of the thread that was running when the signal condition was detected. They can use the NetWare API functions without additional setup.

However, you do need to set up context for the functions registered for SIGABRT.

For 3.11 NLM applications, you must manually create the thread group context in your callback functions, 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 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.

See Also

abort, **_exit**, **longjmp**, **raise**, **setjmp**

SignalLocalSemaphore

Increments the semaphore value of the specified semaphore
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsemaph.h>

int SignalLocalSemaphore (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle
(IN) Specifies the semaphore handle of an open semaphore.

Return Values

If successful, this function returns zero.

WARNING: A bad semaphore handle causes the server to abend.

Remarks

A thread normally call this function when finished accessing the resource associated with the semaphore.

A thread can also use this function to restart another thread waiting on the semaphore, as a means of interprocess synchronization.

If there are threads waiting on the semaphore (the semaphore value is negative), the first thread in the queue is released (made runnable).

A semaphore handle can be obtained by calling **OpenLocalSemaphore**.

See Also

CloseLocalSemaphore, **ExamineLocalSemaphore**, **OpenLocalSemaphore**, **TimedWaitOnLocalSemaphore**, **WaitOnLocalSemaphore**

spawnlp, spawnvp

Execute a new NLM
Local Servers: blocking
Remote Servers: N/A
Classification: Other
SMP Aware: No
Service: Thread

Syntax

```
#include <nwthread.h>

int spawnlp (
   int          mode,
   const char   *path,
   char         *arg0,
   ...);

int spawnvp (
   int          mode,
   const char   *path,
   char         **argv);
```

Parameters

mode

(IN) Specifies how the invoking program behaves after it is initiated.

path

(IN) Points to the name of the compiled program to be started.

arg0

(IN) Points to the first of a list of arguments to be passed to the invoked program.

argv

(IN) Points to an array of pointers to arguments to be passed to the invoked program.

Return Values

0	(0x00)	ESUCCESS	Program was successfully loaded.
-1	(0xFF)	EFAILURE	The function failed.

On *errno* is set to one of the following values:

1	(0x01)	ENOENT
3	(0x03)	ENOEXEC
5	(0x05)	ENOMEM
6	(0x06)	EACCES
9	(0X09)	EINVAL (See note below)
16	(0x10)	EINUSE
19	(0x13)	EWRNGKND
21	(0x15)	ERESOURCE
28	(0x1C)	EIO
37	(0x25)	EALREADY

Through a Novell internal conversion process, *NwErrno* may also be set to one of the following values:

1	(0x01)	LOAD_COULD_NOT_FIND_FILE
2	(0x02)	LOAD_ERROR_READING_FILE
3	(0x03)	LOAD_NOT_NLM_FILE_FORMAT
4	(0x04)	LOAD_WRONG_NLM_FILE_VERSION
5	(0x05)	LOAD_REENTRANT_INITIALIZE_FAILURE
6	(x006)	LOAD_CAN_NOT_LOAD_MULTIPLE_COPIES
7	(0x07)	LOAD_ALREADY_IN_PROGRESS
8	(0x08)	LOAD_NOT_ENOUGH_MEMORY
9	(0x09)	LOAD_INITIALIZE_FAILURE
10	(0x0A)	LOAD_INCONSISTENT_FILE_FORMAT
11	(0x0B)	LOAD_CAN_NOT_LOAD_AT_STARTUP
12	(0x0C)	LOAD_AUTO_LOAD_MODULES_NOT_LOADED
13	(0x0D)	LOAD_UNRESOLVED_EXTERNAL

14	(0x0E)	LOAD_PUBLIC_ALREADY_DEFINED

NOTE: The *errno* code EINVAL indicates only that *code* was set to other than P_NOWAIT.

Remarks

The value of *mode* determines how the program is loaded and how the invoking program behaves after the it is initiated:

P_NOWAIT	The invoked program is loaded into available memory and is executed. The original program executes simultaneously with the invoked program.

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments in the spawn call. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. The length of this concatenated string must not exceed 128 bytes.

The arguments can be passed as a list of arguments (**spawnlp**) or as a vector of pointers (**spawnvp**). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this first argument is the name of the program.

If the arguments are passed as a list, there must be a NULL pointer to mark the end of the argument list.

See Also

abort, **atexit**, **exit**, **_exit**, **getcmd**, **getenv**, **main**, **system**

Example

spawnlp, spawnvp

(spawnlp)

```
#include <nwthread.h>

int   completionCode:
completionCode = spawnlp (P_NOWAIT, "helper.NLM", NULL);
```

(spawnvp)

```
#include <nwthread.h>
```

```
int    completionCode;
char   *argv[5];
completionCode = spawnvp (P_NOWAIT, "helper.nlm", argv);
```

SuspendThread

Prevents a specified thread in the NLM from being scheduled
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

int SuspendThread (
   int   threadID);
```

Parameters

threadID

(IN) Specifies the ID of the thread to be suspended.

Return Values

0	(0x00)	ESUCCESS	Thread was suspended.
22	(0x16)	EBADHNDL	A bad thread ID was passed in.

Remarks

This function causes a specified thread to be suspended. **ResumeThread** makes the thread runnable once again.

SuspendThread maintains a count of the number of times a thread is suspended. An equal number of calls to **ResumeThread** must be performed for the thread to run again. This allows calls to **SuspendThread** and **ResumeThread** to be nested.

Blocking Information SuspendThread does not block when suspending other threads, but blocks when suspending itself.

See Also

EnterCritSec, ResumeThread

system

Executes operating system commands
Local Servers: blocking
Remote Servers: N/A
Classification: ANSI
NetWare Server: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

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

int system (
   const char   *command);
```

Parameters

command

(IN) Specifies a command to execute.

Return Values

A success code (0) is always returned. However, errors in executing the NetWare 3.x OS command are shown on the system console screen.

Remarks

This function always echoes input directly to the system console screen. Errors in executing the NetWare 3.x OS command are shown on the system console screen.

NOTE: If the console operator is typing, your string will be intermixed with his.

See Also

abort, **atexit**, **exit**, **_exit**, **spawnlp**, **spawnvp**

Example

system

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

/*-----------------------------------*/
main ()
{
   .
   .
   .
   system ("LOAD MONITOR");
   .
   .
   .
}
/*-----------------------------------*/
```

ThreadSwitch

Allows other runnable threads a chance to get some work done, where no natural break in the currently running thread would normally occur

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwthread.h>

void ThreadSwitch (void);
```

Return Values

None

Remarks

The NetWare 3.x and 4.x environment is a nonpreemptive environment in which threads can only relinquish control via system calls. Unless an executing thread relinquishes control, other threads do not have the opportunity to work.

If no natural break occurs via a system call in a particular thread, **ThreadSwitch** can be used to cause that thread to relinquish control and allow other runnable threads to execute.

NOTE: If you are using "busy waiting" or "spin locks" you should use **ThreadSwitchWithDelay** instead of **ThreadSwitch** because threads preempted with **ThreadSwitch** still have higher priority than threads on the low priority queue. These low priority threads (such as those doing file compression in the OS) still need an opportunity to run in the nonpreemptive 4.x environment.

ThreadSwitchLowPriority

Reschedules a thread onto the low-priority queue
Local Servers: blocking
Remote Servers: N/A
Classification: 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwthread.h>

void ThreadSwitchLowPriority (void);
```

Return Values

None

Remarks

The **ThreadSwitchLowPriority** function can be used to schedule a thread to run only when there is nothing but hardware polling routines and temporarily handicapped threads to run. Routines suitable for this priority level would be once-a-week backup, file compression utilities, low-priority clean-up utilities, and so forth.

ThreadSwitchWithDelay

Reschedules the thread to be placed on the RunList after *n* number of context switches have taken place

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwthread.h>

void ThreadSwitchWithDelay (void);
```

Return Values

None

Remarks

If a thread needs a resource that will not be ready for a moment but does not want the overhead of sleeping on a semaphore, rather than rescheduling itself repetitively the thread can reschedule itself with a temporary handicap.

Temporarily handicapped threads are scheduled on a waiting queue, the DelayedList, until their handicap has expired. Upon expiration, they are rescheduled at the end of the RunList. Letting threads temporarily handicap themselves prevents needless rescheduling caused by a spin-waiting condition.

The number of switches in each temporary handicap is a tunable parameter inside the NetWare OS.

TimedWaitOnLocalSemaphore

Waits on a local semaphore until it is signalled or the specified timeout elapses

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x

SMP Aware: Yes

Service: Thread

Syntax

```
#include <nwsemaph.h>

int TimedWaitOnLocalSemaphore (
   LONG   semaphoreHandle,
   LONG   timeout);
```

Parameters

semaphoreHandle

(IN) Specifies the handle of the semaphore to wait on.

timeout

(IN) Specifies the maximum time, in milliseconds, to wait on the semaphore.

Return Values

0	(0x00)	ESUCCESS
254	(0xFE)	ERR_TIMEOOUT_FAILURE

WARNING: A bad semaphore handle causes the server to abend.

Remarks

This function is similar to **WaitOnLocalSemaphore** except that a *timeout* is specified. If the semaphore is not signalled prior to the expiration of the *timeout* period, the function returns with an error at that point in time.

See Also

CloseLocalSemaphore

OpenLocalSemaphore, SignalLocalSemaphore, WaitOnLocalSemaphore

WaitOnLocalSemaphore

Decrements the semaphore value of the specified semaphore
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
SMP Aware: Yes
Service: Thread

Syntax

```
#include <nwsemaph.h>

int WaitOnLocalSemaphore (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle
(IN) Specifies the semaphore handle of an open semaphore.

Return Values

If successful, this function returns zero.

WARNING: A bad semaphore handle causes the server to abend.

Remarks

A thread would typically call this function before accessing the resource associated with the semaphore. An NLM can also use this function to cause a thread to wait until another thread signals it to resume.

If the semaphore value is still greater than or equal to zero after the function decrements it, the thread is not suspended. If the semaphore value is negative, the thread is suspended until the semaphore is signalled one more time than there are threads ahead of the current thread on the specified semaphore's queue.

A semaphore handle can be obtained by calling **OpenLocalSemaphore**.

See Also

CloseLocalSemaphore, **ExamineLocalSemaphore**, **OpenLocalSemaphore**, **SignalLocalSemaphore**, **TimedWaitOnLocalSemaphore**

Variable Length Argument Lists

Variable Length Argument Lists: Functions

va_arg

Obtains the next argument in a list of variable arguments (macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Variable Length Argument Lists

Syntax

```
#include <stdarg.h>

type va_arg (
   va_list param,
   type);
```

Parameters

param

(IN) Specifies a variable argument.

type

(IN) Specifies the argument type.

Return Values

va_arg returns the value of the next variable argument, according to type passed as the second parameter.

Remarks

The macro **va_arg** can be used to obtain the next argument in a list of variable arguments. It must be used with the associated macros **va_start** and **va_end**. A sequence such as:

```
va_list curr_arg;
type next_arg;
next_arg = va_arg (curr_arg, type);
```

causes next_arg to be assigned the value of the next variable argument. The *type* is the type of the argument originally passed.

The macro **va_start** must be executed first in order to properly initialize the variable next_arg and the macro **va_end** should be executed after all arguments have been obtained.

The data item curr_arg is of type va_list which contains the information

to permit successive acquisitions of the arguments.

See Also

va_end, va_start, vfprintf, vprintf, vsprintf

Example

va_arg

```
#include <stdarg.h>

void test_fn (const char *msg, const char *types,  ...);

int main ()
{
      printf ("VA...TEST\n");
      test_fn ("PARAMETERS: 1, \"abc\", 546", "isi", 1, "abc", 546 );
      test_fn ("PARAMETERS: \"def\", 789", "si", "def", 789 );
}

static void test_fn (

const char *msg,      /* Message to be printed */
const char *types,    /* Parameter types (i,s) */
   ... )              /* Variable arguments    */

{
   va_list  argument;
   int   arg_int;
   char  *arg_string;
   char *types_ptr;
   types_ptr = types;
   printf ("\n%s -- %s\n", msg, types);
   va_start (argument, types);
   while (*types_ptr != `\0')
   {
      if (*types_ptr == `i')
      {
         arg_int = va_arg (argument, int );
         printf  ("integer: %d\n", arg_int );
      }
      else if (*types_ptr == `s')
      {
         arg_string = va_arg ( argument, char *);
         printf ("string:  %s\n", arg_string);
      }
      ++types_ptr;
   }
   va_end (argument);
```

```
}
```

produces the following:

```
PARAMETERS: 1, "abc", 546
integer: 1
string:  abc
integer: 546
PARAMETERS: "def", 789
string:  def
integer: 789
```

va_end

Completes the acquisition of arguments from a list of variable arguments (macro)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

SMP Aware: No

Service: Variable Length Argument Lists

Syntax

```
#include <stdarg.h>

void va_end (
   va_list   param);
```

Parameters

param

(IN) Specifies a variable argument.

Return Values

None

Remarks

The macro **va_end** is used to complete the acquisition of arguments from a list of variable arguments. It must be used with the associated macros **va_start** and **va_arg**. See the description for **va_arg** for complete documentation on these macros.

See Also

va_arg, **va_start**, **vfprintf**, **vprintf**, **vsprintf**

va_start

Starts the acquisition of arguments from a list of variable arguments (macro)
Local Servers: nonblocking
Remote Servers: N/A
Classification: ANSI
SMP Aware: No
Service: Variable Length Argument Lists

Syntax

```
#include <stdarg.h>

void va_start (
   va_list   param,
   previous);
```

Parameters

param

(IN) Specifies a variable argument.

previous

(IN) Specifies the previous argument being passed to the called function.

Return Values

None

Remarks

The macro **va_start** is used to start the acquisition of arguments from a list of variable arguments. The *param* argument is used by the **va_arg** macro to locate the current acquired argument. The *previous* argument is the argument that immediately precedes the "..." notation in the original function definition. It must be used with the associated macros **va_arg** and **va_end**. See the description of **va_arg** for complete documentation on these macros.

See Also

va_arg, **va_end**, **vfprintf**, **vprintf**, **vsprintf**