Novell Developer Kit

www.novell.com

NLM™ USER INTERFACE DEVELOPER COMPONENTS

104-000158-002

Novell®

Legal Notices

Novell, Inc.
122 East 1700 South
Provo, UT 84606
U.S.A.

www.novell.com

NLM User Interface Developer Components
September 2002

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

Novell Trademarks

AppNotes is a registered trademark of Novell, Inc.
AppTester is a registered trademark of Novell, Inc., in the United States.
ASM is a trademark of Novell, Inc.
BorderManager is a registered trademark of Novell, Inc.
BrainShare is a registered service mark of Novell, Inc., in the United States and other countries.
C3PO is a trademark of Novell, Inc.
Client 32 is a trademark of Novell, Inc.
ConsoleOne is a registered trademark of Novell, Inc.
Controlled Access Printer is a trademark of Novell, Inc.
Custom 3rd-Party Object is a trademark of Novell, Inc.
DeveloperNet is a registered trademark of Novell, Inc. in the United States and other countries.
DeveloperNet 2000 is a trademark of Novell, Inc.
Direct Connect is a service mark of Novell, Inc.
DirXML is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
Full Service Directory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc. in the United States and other countries.
Hardware Specific Module is a trademark of Novell, Inc.
Hot Fix is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
Internetwork Packet Exchange is a trademark of Novell, Inc.
IPX is a trademark of Novell, Inc.
IPX/SPX is a trademark of Novell, Inc.
Link Support Layer is a trademark of Novell, Inc.
LSL is a trademark of Novell, Inc.
ManageWise is a registered trademark of Novell, Inc., in the United States and other countries.
Mirrored Server Link is a trademark of Novell, Inc.
MSL is a trademark of Novell, Inc.
MSM is a trademark of Novell, Inc.
NCP is a trademark of Novell, Inc.
NDPS is a registered trademark of Novell, Inc.
NDS is a registered trademark of Novell, Inc. in the United States and other countries.
NDS Administrator is a trademark of Novell, Inc.
NDS Manager is a trademark of Novell, Inc.
NE2000 is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc. in the United States and other countries.
NetWare/IP is a trademark of Novell, Inc.
NetWare Connect is a registered trademark of Novell, Inc. in the United States.
NetWare Core Protocol is a trademark of Novell, Inc.
NetWare Loadable Module is a trademark of Novell, Inc.
NetWare Management Portal is a trademark of Novell, Inc.
NetWare Management System is a trademark of Novell, Inc.
NetWare MHS is a registered trademark of Novell, Inc.
NetWare Name Service is a trademark of Novell, Inc.
NetWare Peripheral Architecture is a trademark of Novell, Inc.
NetWare Print Server is a trademark of Novell, Inc.
NetWare Requester is a trademark of Novell, Inc.

NetWare SFT and NetWare SFT III are trademarks of Novell, Inc.
NetWare SQL is a trademark of Novell, Inc.
NetWire is a registered service mark of Novell, Inc. in the United States and other countries.
NIMS is a trademark of Novell, Inc.
NLM is a trademark of Novell, Inc.
NMAS is a trademark of Novell, Inc.
NMS is a trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc. in the United States and other countries.
Novell Authorized Reseller is a service mark of Novell, Inc.
Novell Authorized Service Center is a service mark of Novell, Inc.
Novell Certificate Server is a trademark of Novell, Inc.
Novell Client is a trademark of Novell, Inc.
Novell Cluster Services is a trademark of Novell, Inc.
Novell Directory Services is a registered trademark of Novell, Inc.
Novell Distributed Print Services is a trademark of Novell, Inc.
Novell Internet Messaging System is a trademark of Novell, Inc.
Novell Labs is a trademark of Novell, Inc.
Novell Press is a trademark of Novell, Inc.
Novell SecretStore is a trademark of Novell, Inc.
Novell Security Attributes is a trademark of Novell, Inc.
Novell Storage Services is a trademark of Novell, Inc.
Novell Web Server is a trademark of Novell, Inc.
Novell, Yes, Tested & Approved logo is a trademark of Novell, Inc.
NSI is a trademark of Novell, Inc.
ODI is a trademark of Novell, Inc.
Open Data-Link Interface is a trademark of Novell, Inc.
Packet Burst is a trademark of Novell, Inc.
Printer Agent is a trademark of Novell, Inc.
QuickFinder is a trademark of Novell, Inc.
Red Box is a trademark of Novell, Inc.
Remote Console is a trademark of Novell, Inc.
RX-Net is a trademark of Novell, Inc.
Sequenced Packet Exchange is a trademark of Novell, Inc.
SFT and SFT III are trademarks of Novell, Inc.
SPX is a trademark of Novell, Inc.
Storage Management Services is a trademark of Novell, Inc.
System V is a trademark of Novell, Inc.
Topology Specific Module is a trademark of Novell, Inc.
Transaction Tracking System is a trademark of Novell, Inc.
TSM is a trademark of Novell, Inc.
TTS is a trademark of Novell, Inc.
Universal Component System is a registered trademark of Novell, Inc.
Virtual Loadable Module is a trademark of Novell, Inc.
VLM is a trademark of Novell, Inc.
ZENworks is a registered trademark of Novell, Inc.

Third-Party Trademarks

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

Contents

NLM User Interface Developer Components Preface 13

1 NWSNUT Concepts 15

The NWSNUT Environment . . . 15
Portals . . . 17
- Portal Types . . . 17
- Basic Steps for Using Portals . . . 18
- Considerations During Portal Creation . . . 18
- Considerations for Writing Data to a Portal . . . 19
- Special Purpose Portals . . . 21
- Basic Portal Functions . . . 22

Zones . . . 22
Lists . . . 23
- Basic Steps for Using Lists . . . 24
- Manipulating List Elements . . . 24
- Handling Lists . . . 26
- Routines for List Elements . . . 26
- Specialized Lists . . . 26

Menus . . . 27
- Basic Steps for Using Lists . . . 27

Forms . . . 27
- Field Structure . . . 27
- Prompt Fields . . . 30
- Menu Fields . . . 30
- Custom Fields . . . 31
- Form Functions . . . 31

Text in NWSNUT . . . 33
- NWSNUT Messages . . . 33
- Help Screens . . . 35

User Input . . . 36
- Keyboard Input . . . 37
- Function Keys . . . 37
- Interrupt Keys . . . 38
- Editing a String . . . 38
- Confirmation of a Decision . . . 38

NWSNUT Function List . . . 39

2 NWSNUT Tasks 51

Using Portals . . . 51
Using Lists . . . 53

Using Menus . . . 54
Using Forms . . . 55

3 NWSNUT Functions 57

NWSAlert to NWSAppend* Functions . . . 57
NWSAlert . . . 58
NWSAlertWithHelp . . . 60
NWSAlignChangedList . . . 62
NWSAlloc . . . 64
NWSAppendBoolField . . . 65
NWSAppendCommentField . . . 68
NWSAppendHexField . . . 70
NWSAppendHotSpotField . . . 73
NWSAppendIntegerField . . . 76
NWSAppendMenuField . . . 79
NWSAppendPasswordField . . . 82
NWSAppendPromptField . . . 86
NWSAppendScrollableStringField . . . 88
NWSAppendStringField . . . 92
NWSAppendToForm . . . 95
NWSAppendToList . . . 99
NWSAppendToMenu . . . 101
NWSAppendToMenuField . . . 103
NWSAppendUnsignedIntegerField . . . 105
NWSAscii* to NWSDestroy* Functions . . . 108
NWSAsciiHexToInt . . . 109
NWSAsciiToInt . . . 110
NWSAsciiToLONG . . . 111
NWSClearPortal . . . 112
NWSComputePortalPosition . . . 113
NWSConfirm . . . 115
NWSCreatePortal . . . 117
NWSDeleteFromList . . . 122
NWSDeleteFromPortalList . . . 124
NWSDeselectPortal . . . 126
NWSDestroyForm . . . 127
NWSDestroyList . . . 128
NWSDestroyMenu . . . 129
NWSDestroyPortal . . . 130
NWSDisable* to NWSDraw* Functions . . . 131
NWSDisableAllFunctionKeys . . . 132
NWSDisableAllInterruptKeys . . . 133
NWSDisableFunctionKey . . . 134
NWSDisableInterruptKey . . . 135

NWSDisablePortalCursor . . . 136
NWSDisplayErrorCondition . . . 137
NWSDisplayErrorText . . . 140
NWSDisplayHelpScreen . . . 142
NWSDisplayInformation . . . 143
NWSDisplayInformationInPortal . . . 146
NWSDisplayPreHelp . . . 152
NWSDisplayTextInPortal . . . 154
NWSDisplayTextJustifiedInPortal . . . 156
NWSDrawPortalBorder . . . 159
NWSEdit* to NWSFree Functions . . . 160
NWSEditForm . . . 161
NWSEditPortalForm . . . 164
NWSEditPortalFormField . . . 167
NWSEditString . . . 170
NWSEditText . . . 174
NWSEditTextWithScrollBars . . . 177
NWSEnableAllFunctionKeys . . . 180
NWSEnableFunctionKey . . . 181
NWSEnableFunctionKeyList . . . 182
NWSEnableInterruptKey . . . 183
NWSEnableInterruptList . . . 185
NWSEnablePortalCursor . . . 187
NWSEndWait . . . 188
NWSFillPortalZone . . . 189
NWSFillPortalZoneAttribute . . . 192
NWSFree . . . 195
NWSGet* Functions . . . 196
NWSGetADisk . . . 197
NWSGetDefaultCompare . . . 199
NWSGetFieldFunctionPtr . . . 200
NWSGetHandleCustomData . . . 202
NWSGetKey . . . 204
NWSGetLineDrawCharacter . . . 206
NWSGetList . . . 207
NWSGetListHead . . . 208
NWSGetListIndex . . . 209
NWSGetListNotifyProcedure . . . 211
NWSGetListSortFunction . . . 213
NWSGetListTail . . . 214
NWSGetMessage . . . 215
NWSGetNUTVersion . . . 217
NWSGetPCB . . . 218
NWSGetScreenPalette . . . 220

NWSGetSortCharacter . . . 221
NWSInit* to NWSModify* Functions . . . 222
NWSInitForm . . . 223
NWSInitializeNut . . . 227
NWSInitList . . . 231
NWSInitListPtr . . . 234
NWSInitMenu . . . 235
NWSInitMenuField . . . 238
NWSInsertInList . . . 240
NWSInsertInPortalList . . . 242
NWSIsAnyMarked . . . 245
NWSIsdigit . . . 247
NWSIsxdigit . . . 249
NWSKeyStatus . . . 251
NWSList . . . 252
NWSMemmove . . . 256
NWSMenu . . . 257
NWSModifyInPortalList . . . 260
NWSPop* to NWSRestore* Functions . . . 262
NWSPopHelpContext . . . 263
NWSPopList . . . 265
NWSPopMarks . . . 267
NWSPositionCursor . . . 268
NWSPositionPortalCursor . . . 270
NWSPromptForPassword . . . 271
NWSPushHelpContext . . . 274
NWSPushList . . . 276
NWSPushMarks . . . 278
NWSRemovePreHelp . . . 279
NWSRestoreDisplay . . . 280
NWSRestoreList . . . 281
NWSRestoreNut . . . 283
NWSRestoreZone . . . 284
NWSSave* to NWSSet* Functions . . . 286
NWSSaveFunctionKeyList . . . 287
NWSSaveInterruptList . . . 288
NWSSaveList . . . 289
NWSSaveZone . . . 291
NWSScreenSize . . . 293
NWSScrollPortalZone . . . 294
NWSScrollZone . . . 296
NWSSelectPortal . . . 299
NWSSetDefaultCompare . . . 300
NWSSetDynamicMessage . . . 302

NWSSetErrorLabelDisplayFlag . . . 304
NWSSetFieldFunctionPtr . . . 305
NWSSetFormRepaintFlag . . . 308
NWSSetHandleCustomData . . . 310
NWSSetList . . . 312
NWSSetListNotifyProcedure . . . 313
NWSSetListSortFunction . . . 315
NWSSetScreenPalette . . . 317
NWSShow* to NWSWait* Functions . . . 319
NWSShowLine . . . 320
NWSShowLineAttribute . . . 322
NWSShowPortalLine . . . 324
NWSShowPortalLineAttribute . . . 326
NWSSortList . . . 328
NWSStartWait . . . 329
NWSStrcat . . . 331
NWSToupper . . . 332
NWSTrace . . . 333
NWSUngetKey . . . 335
NWSUnmarkList . . . 337
NWSUpdatePortal . . . 338
NWSViewText . . . 339
NWSViewTextWithScrollBars . . . 341
NWSWaitForEscape . . . 344
NWSWaitForEscapeOrCancel . . . 345
NWSWaitForKeyAndValue . . . 347

4 NWSNUT Structures 351

FIELD . . . 352
INTERRUPT . . . 357
LIST . . . 358
LISTPTR . . . 361
MessageInfo . . . 362
MFCONTROL . . . 363
NUTInfo . . . 365
PCB . . . 370
PROCERROR . . . 375

5 NLM Internationalization Concepts 377

Internationalization Checklist . . . 378
File Name Changes for MSGLIB . . . 379
Format Specifiers Supported in Localization . . . 380
Language IDs . . . 380
Locale-Sensitive Routines for Text Formatting . . . 381

Message Database Flags and Fields . . . 382
Message Librarian (MSGLIB.EXE) . . . 383
String Expansion Factors . . . 383
Transfer File Keywords . . . 384
Use of Single Byte Characters for NetWare Designations . . . 386
Internationalization Tools . . . 386
Most Commonly Used Internationalization Tools . . . 386
Complete Summary of Internationalization Tools . . . 386
Summaries of the Tools . . . 387

6 NLM Internationalization Tasks 389

Isolating the User Interface . . . 389
Making the User Interface Localizable . . . 392
Annotating NLM Programs . . . 394
Identifying Length Limits . . . 397
Describing Insertion Parameters . . . 397
Annotating Context . . . 398
Clarifying Ambiguous Words . . . 399
Handling Variations in Localized Text . . . 399
Designing Displays to Accommodate Text Expansion . . . 400
Designing Screen Layouts to Accommodate Text Expansion . . . 400
Using Functions that Support Parameter Reordering . . . 400
Formatting Text in a Locale Sensitive Way . . . 401
Designing Case-Conversion Routines to Skip over Double-Byte Characters . . . 401
Eliminating Locale-Specific Programming Techniques . . . 402
Handling Double-Byte Characters . . . 402
Using "Double-Byte Aware" String-Handling Functions . . . 403
Eliminating Single-Byte Specific Programming Techniques . . . 404
Converting to and from Unicode as Needed . . . 405
Isolating the Hardware Interface . . . 405
Dealing with Hardware Resource Constraints . . . 405
Testing Your Program . . . 406
Scanning the Source Code . . . 407
Scanning the Message Database . . . 408
Using Fake Message Modules . . . 409
Testing for String Isolation . . . 410
Testing for String Concatenation . . . 411
Testing for String Expansion . . . 411
Testing for Parameter Reordering . . . 411
Using MSGLIB . . . 411
Starting MSGLIB . . . 411
Creating a New Message Library . . . 412
Editing an Existing Message Library . . . 414
Printing a Message Library . . . 414

7 NLM Internationalization Tools Reference 417

MSGCHECK . . . 418
MSGDUMP . . . 420
MSGEDIT . . . 422
MSGEXP . . . 423
MSGEXTR . . . 426
MSGFILES . . . 429
MSGFLAGS . . . 430
MSGGLOSS . . . 431
MSGGREP . . . 433
MSGIMP . . . 435
MSGLABEL . . . 437
MSGMAKE . . . 439
MSGPFIX . . . 441
MSGPURGE . . . 442
MSGRENAM . . . 443
MSGSTATS . . . 444
MSGTRAN . . . 447
MSGXFER . . . 449

Revision History 453

NLM User Interface Developer Components Preface

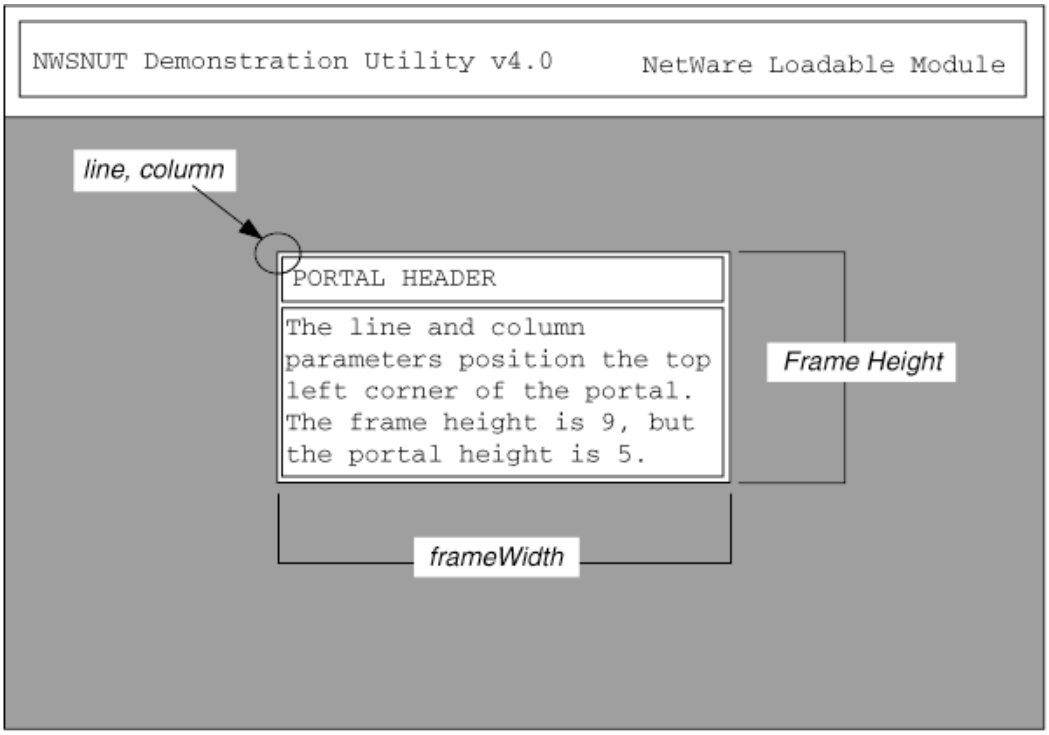
The building blocks that customize the NetWare® 3.x, 4.x, and 5.x OS are known as NetWare Loadable Modules (NLMs). These programs are built to run in server memory with the NetWare OS. You can load or unload NLMs from server memory while the server is running. Once loaded, NLMs become part of the OS. This means that NLMs can access NetWare services directly without using a service protocol such as the Novell NCP service. The server procedures that NLMs can access are collectively called the NetWare API. A fundamental part of the NetWare API is a core set of APIs that provide a direct programming link into the NetWare 3.x, 4.x, and 5.x OS services.

1 NWSNUT Concepts

This documentation describes NWSNUT, its functions, and features.

The NWSNUT Environment

The NWSNUT environment is created by calling **NWSInitializeNut (page 227)**. 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.



The NUTInfo (page 365) 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.

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 (page 310)**. To retrieve the data and release this function, call **NWSGetHandleCustomData (page 202)**.

When you have finished using the NWSNUT library, call **NWSRestoreNut (page 283)**. This function cleans up resources used by NWSNUT.

Portals

The portal is the central element of NWSNUT. A portal is a "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 (page 218)**. This structure holds information about the portal, including its position on the screen, its frame size, and cursor state and position.

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 (page 117)**. Other portal functions require a pointer to a PCB. The PCB pointer can be obtained by calling **NWSGetPCB**.

The portal type is determined by the value of the *directFlag* parameter (VIRTUAL or DIRECT) that is passed to **NWSCreatePortal (page 117)** when the portal is created.

Portal Types

Portals are classified as one of two types:

Direct portal

Portal screen access is direct.

Virtual portal

Portal screen access is buffered and indirect.

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.

Virtual Portals

A *virtual portal* is an area of memory into which data intended for the portal is written. When **NWSUpdatePortal (page 338)** 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 (page 117)**. 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 (page 338)**.

Basic Steps for Using Portals

Using portals involves four basic steps:

1. Create a portal.
2. Convert the portal number to a PCB.
3. Write data to the portal.
4. Destroy the portal.

These steps are described in detail in the task "Using Portals" on page 51.

Considerations During Portal Creation

In portal creation, video attributes, positioning of a portal, and the screen palette for specifying colors are each important considerations.

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)

Positioning the Portal

NWSNUT provides two functions that can help you calculate the positioning parameters to use when creating portals. **NWSScreenSize (page 293)** returns the number of display lines and columns on the server screen. **NWSComputePortalPosition (page 113)** returns the zero-based top row and left-most column (the line and column parameters for **NWSCreatePortal (page 117)**), given the desired line and column to center the portal on.

Screen Palette

The screen palette determines the colors of your portal. The screen palette can be set by calling **NWSSetScreenPalette (page 317)**. 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 (page 220)**.

Considerations for Writing Data to a Portal

After a portal is created, portal selection, control of the portal cursor, characters for line drawing, and ways of scrolling a portal are each important considerations.

Selecting Portals

To select a portal, call **NWSSelectPortal (page 299)**. 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 (page 126)** dims the border and disables the portal cursor.

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 functions are provided by NWSNUT for changing the cursor state and manipulating the cursor:

NWSEnablePortalCursor (page 187)

Enables the portal cursor.

NWSDisablePortalCursor (page 136)

Disables the portal cursor.

NWSPositionCursor (page 268)

Positions the cursor relative to the entire screen.

NWSPositionPortalCursor (page 270)

Positions the portal cursor within the portal.

Line Drawing Characters

To obtain line drawing characters for drawing in a portal, call **NWSGetLineDrawCharacter (page 206)** (see "character and key constants" in nwsnut.h).

Scrolling the Portal

You can scroll the display area of a portal up or down by calling **NWSScrollPortalZone (page 294)**. 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 (page 338)** to transfer the scroll to the physical screen.

Special Purpose Portals

In addition to providing functions that allow you to create and manipulate portals, NWSNUT provides several special purpose portals. Each of the following functions automatically creates a portal:

NWSDisplayInformation (page 143)	Displays text in a portal. Video attribute can be specified for the text and palette can be specified for the portal.
NWSDisplayInformationInPortal (page 146)	Displays text in a portal. Justification style, indentation, 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 (page 339)	Displays text within a portal.
NWSViewTextWithScrollBars (page 341)	Displays scrollable text within a portal.
NWSShowLine (page 320)	Displays text at a specified screen location.
NWSShowLineAttribute (page 322)	Identical to **NWSShowLine (page 320)**, but also allows screen attribute specification.
NWSEditString (page 170)	Allows the user to edit text within a portal. Insertion and action routines can be specified, and input type can be restricted.
NWSEditText (page 174)	Allows the user to edit text within a portal.
NWSEditTextWithScrollBars (page 177)	Allows the user to edit scrollable text within a portal.
NWSStartWait (page 329)	Creates a portal containing a "please wait" message.
NWSEndWait (page 188)	Destroys the portal created by **NWSStartWait (page 329)**.
NWSDisplayErrorCondition (page 137)	Creates an error portal that displays an error message and the name of the procedure that resulted in the error.
NWSDisplayErrorText (page 140)	Creates an error portal that displays an error message.
NWSAlert (page 58)	Creates an alert portal.
NWSAlertWithHelp (page 60)	Creates an alert portal with help context.
NWSConfirm (page 115)	Creates a confirm portal.
NWSGetADisk (page 197)	Creates a portal prompting the user to insert a floppy disk.

NWSTrace (page 333)	Displays an information portal on the screen.

Basic Portal Functions

The following functions assist with portals as indicated:

NWSShowPortalLine (page 324)	Displays a line of text. Portal line and column can be specified.
NWSShowPortalLineAttribute (page 326)	Changes the video attribute of a specified portion of a portal line.
NWSDisplayTextInPortal (page 154)	Displays text in an existing portal. Starting portal line for text and indent level can be specified.
NWSDisplayTextJustifiedInPortal (page 156)	Same as above, but text width and video attribute can be specified.
NWSFillPortalZone (page 189)	Fills a specified region of the portal with a character.
NWSFillPortalZoneAttribute (page 192)	Fills a specified region of the portal with a video attribute.
NWSClearPortal (page 112)	Blanks out a portal.
NWSUpdatePortal (page 338)	Redraws a virtual portal to show changes since creation or last update.
NWSDrawPortalBorder (page 159)	Redraws the border of a portal.

Zones

Zones are used to display only text anywhere on the console screen. NWSNUT provides the following three functions for manipulating zones:

NWSRestoreZone (page 284)	Restores text saved in a buffer to the screen.
NWSSaveZone (page 291)	Saves text in a defined area on the screen to a buffer.
NWSScrollZone (page 296)	Enables text displayed in a defined area on the screen to be scrolled.

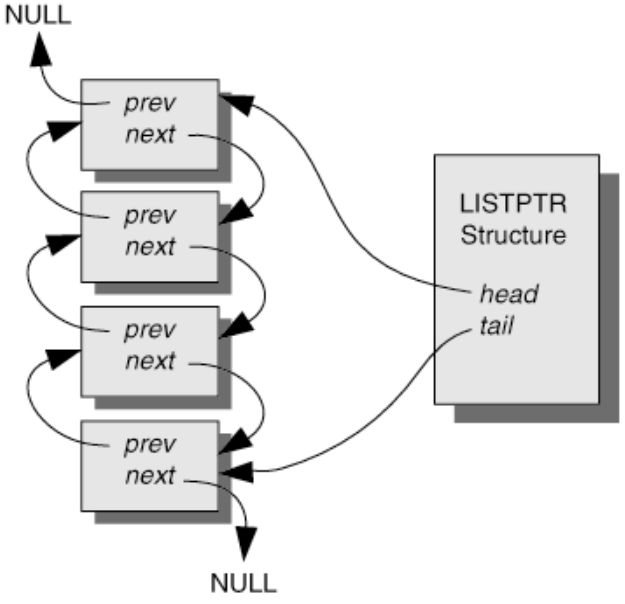
Lists

This explains NWSNUT lists and how to create, add to, manipulate, and destroy 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. 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 that 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 1 List Structure



Fields in the LISTPTR and LIST structures should not be changed directly. NWSNUT functions keep the information in these structures current.

Basic Steps for Using Lists

Using lists involves four basic steps:

1. Initialize the list.
2. Add items to the list.
3. Display the list and allow the user to access it.
4. Destroy the list.

These steps are described in detail in the task "Using Lists" on page 53.

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 functions are used to add, delete, and modify list elements:

NWSAppendToList (page 99)	Adds an element to the end of the current list.
NWSInsertInList (page 240) **NWSInsertInPortalList (page 242)**	Inserts an element into a list at a specific location.
NWSDeleteFromPortalList (page 124)	Deletes current and marked list elements.
NWSDeleteFromList (page 122)	Deletes a specified list element.
NWSModifyInPortalList (page 260)	Modifies the text field of a list element.

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 at once on which to perform an action. The following table summarizes functions that deal with marking list elements:

NWSIsAnyMarked (page 245)

Determines whether any items in a list are marked.

NWSUnmarkList (page 337)

Changes the status of all list elements to unmarked.

NWSPushMarks (page 278)

Saves the marked status of all list elements in the current list.

NWSPopMarks (page 267)

Retrieves the saved marked status of all list elements in the current list.

Sorting Lists

The list can be sorted by calling **NWSSortList (page 328)**. This function uses the *defaultCompareFunction* from the NUTInfo structure to sort. If you want to create your own sorting routine, call **NWSSetDefaultCompare (page 300)**. To obtain the current default compare function, call **NWSGetDefaultCompare (page 199)**. **NWSGetSortCharacter (page 221)** returns the weighted value used for sorting a given character. If you want to create you own list sort function, call **NWSSetListSortFunction (page 315)**. Call **NWSGetListSortFunction (page 213)** to retrieve the current list sort function being used by the NLM.

NWSGetDefaultCompare (page 199)

Obtains the current routine for comparing list elements.

NWSGetListSortFunction (page 213)

Returns a pointer to the currently set list sort function.

NWSGetSortCharacter (page 221)

Returns the weighted value used for sorting a given character.

NWSSetDefaultCompare (page 300)

Specifies a routine for sorting list elements.

NWSSetListSortFunction (page 315)

Enables the use of a customized list sort function.

NWSSortList (page 328)

Sorts the current list alphabetically.

Handling Lists

NWSSetList (page 312) makes a list current. To obtain the pointers for a list, call **NWSGetList (page 207)**. To obtain a pointer to the head or to the tail of a list, call **NWSGetListHead (page 208)** or **NWSGetListTail (page 214)**, respectively.

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 (page 276)** and **NWSPopList (page 265)** are used to save lists to the list stack. **NWSSaveList (page 289)** and **NWSRestoreList (page 281)** are used to save lists to the save stack. Both stacks are held in the NUTInfo structure.

Routines for List Elements

Although the calling NLM turns control over to NWSNUT while the user makes a selection (when **NWSList (page 252)** 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 the LIST structure and is set by calling **NWSSetListNotifyProcedure (page 313)**. To obtain the entry procedure, call **NWSGetListNotifyProcedure (page 211)**. The action procedure is specified when **NWSList (page 252)** is called to allow the user to manipulate the list.

Specialized Lists

Menus (page 27) and Forms (page 27) are special types of lists. They are created with different functions and are not directly manipulated as lists, but their NWSNUT-internal structures are based on a list.

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 "Using Menus" on page 54.

Basic Steps for Using Lists

Using menus involves four basic steps:

1. Initialize the menu.
2. Add options to the menu.
3. Display the menu and allow the user to choose from its options.
4. Destroy the menu.

These steps are described in detail in the task "Using Menus" on page 54.

Forms

This explains an NWSNUT form, several types of fields, and functions available for creating, editing, manipulating, and destroying forms.

A form is another specific type of list. Forms are designed to allow the user to input various types of data.

Field Structure

Each "append" function creates a structure of type FIELD, defined in nwsnut.h. The FIELD structure is defined 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
```

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*, etc.).

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.

Value	Meaning
NORMAL_FIELD	Normal, editable field.
LOCKED_FIELD	Inaccessible field.
SECURE_FIELD	Noneditable field.
REQURED_FIELD	Verify field on form exit.

Value	Meaning
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 (page 195)** so that **NWSAlloc (page 64)** 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.

Prompt Fields

Notice that **NWSAppendCommentField (page 68)** and **NWSAppendPromptField (page 86)** do essentially the same thing. The difference between the two functions is that **NWSAppendCommentField (page 68)** takes a string (type BYTE) as input for its message, whereas **NWSAppendPromptField (page 86)** takes a message identifier (see "NWSNUT Messages" on page 33).

Menu Fields

Creating a menu field is more complicated than creating other fields. To create a menu field, you must complete the following steps:

1. Initialize the menu field by calling **NWSInitMenuField (page 238)**.
2. Build the menu by calling **NWSAppendToMenuField (page 103)** once for each menu option in the menu field.
3. Append the menu field to the form by calling **NWSAppendMenuField (page 79)**.

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 standalone menu.

Custom Fields

NWSAppendToForm (page 95) 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 (page 305)**, and retrieved by calling **NWSGetFieldFunctionPtr (page 200)**.

Form Functions

Among functions that can be used in working with forms are functions for creating or appending field types and functions for editing forms.

Form Field Type Functions

Field	Append Function	Use
Boolean	**NWSAppendBoolField (page 65)**	Creates a special menu list where the choices are yes and no.
Comment	**NWSAppendCommentField (page 68)**	Creates a field with a prompt. The user cannot edit this field.

Field	Append Function	Use
Custom	NWSAppendToForm (page 95)	Creates a field that is developer-defined.
Hexadecimal	NWSAppendHexField (page 70)	Creates a field that accepts only hexadecimal input from the user.
Hot Spot	NWSAppendHotSpotField (page 73)	Creates a field that calls a "spot action" routine when selected.
Integer	NWSAppendIntegerField (page 76)	Creates a field that accepts only an integer as input from the user.
Menu	NWSAppendMenuField (page 79)	Creates a field that displays a menu when selected.
Password	NWSAppendPasswordField (page 82)	Creates an edit field for password input.
Prompt	NWSAppendPromptField (page 86)	Creates a field with a prompt. The user cannot edit this field.
Scrollable String	NWSAppendScrollableStringField (page 88)	Creates a scrollable field containing an editable string.
String	NWSAppendStringField (page 92)	Creates a field containing an editable string.
Unsigned Integer	NWSAppendUnsignedIntegerField (page 105)	Creates a field that accepts only an unsigned integer as input from the user.

Form Editing Functions

NWSEditForm (page 161)

Allows the user to edit the form.

NWSEditPortalForm (page 164)

Same as **NWSEditForm (page 161)**, but allows you to specify help context for the form.

NWSEditPortalFormField (page 167)

Same as **NWSEditForm (page 161)**, but allows you to specify help context and first field to highlight.

NWSSetFormRepaintFlag (page 308)

Repaints the form to refect changes made to the form.

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

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 (page 227)** 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 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 (page 227)**, pass the array index (0, 1, etc.) of the desired message.

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 (page 215)**. 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.

Dynamic Messages

In addition to the message table specified when **NWSInitializeNut (page 227)** is called, you can specify messages by using the dynamic messages held in the MessageInfo structure. Call **NWSSetDynamicMessage (page 302)** 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);
```

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 (page 58)** or **NWSAlertWithHelp (page 60)**. Error portals are created with **NWSDisplayErrorText (page 140)** and **NWSDisplayErrorCondition (page 137)**.

NWSAlert (page 58) 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 (page 140) and **NWSDisplayErrorCondition (page 137)** 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 (page 137)** 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 (page 304)**.

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 (page 227)**, 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 (page 274)**), 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 (page 274)**. 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 (page 263)**. 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 (page 142)**.

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 (page 152)**. To remove a pre-help portal, call **NWSRemovePreHelp (page 279)**.

User Input

NWSNUT provides several ways to receive user input. You can obtain key strokes, strings, fill text buffers, use forms, or you can 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” on page 37
- “Function Keys” on page 37
- “Interrupt Keys” on page 38
- “Editing a String” on page 38
- “Confirmation of a Decision” on page 38

Keyboard Input

The functions that deal with keyboard input are as follows:

NWSGetKey (page 204)

Reads one key from the keyboard buffer.

NWSKeyStatus (page 251)

Indicates whether a key is waiting in the keyboard buffer.

NWSUngetKey (page 335)

Inserts a key into the keyboard buffer.

NWSWaitForEscape (page 344)

Waits for the Esc key to be pressed.

NWSWaitForEscapeOrCancel (page 345)

Waits for either the Esc or cancel (F7) key to be pressed.

NWSWaitForKeyAndValue (page 347)

Waits for one of the keys in a specified set to be pressed.

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. NWSNUT's Function Key functions are as follows:

NWSDisableAllFunctionKeys (page 132)

Disables all function keys.

NWSDisableFunctionKey (page 134)

Disables a single function key.

NWSEnableAllFunctionKeys (page 180)

Enables all function keys.

NWSEnableFunctionKey (page 181)

Enables a single function key.

NWSEnableFunctionKeyList (page 182)

Enables a list of function keys.

NWSSaveFunctionKeyList (page 287)

Saves a list of function keys.

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:

NWSDisableAllInterruptKeys (page 133)

Disables all interrupt keys.

NWSEnableInterruptKey (page 183)

Enables in interrupt key and associates that key with a routine.

NWSEnableInterruptList (page 185)

Enables a list of interrupt keys.

NWSSaveInterruptList (page 288)

Saves a list of interrupt keys.

Editing a String

NWSNUT provides two functions that allow the user to edit a string—**NWSEditString (page 170)** and **NWSEditText (page 174)**. Both functions create a portal in which the user can edit a string, but **NWSEditString (page 170)** allows you to specify an insertion routine and an action routine.

Confirmation of a Decision

NWSNUT provides a function that creates a special-purpose portal to allow a user to confirm a decision. **NWSConfirm (page 115)** displays a portal that contains a menu with only "yes" and "no" options.

NWSNUT Function List

NOTE: NWSNUT functions require that the 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 (page 58)

Displays an alert portal.

NWSAlertWithHelp (page 60)

Displays an alert portal with access to help.

NWSAlloc (page 64)

Allocates memory for NWSNUT purposes.

NWSAppendBoolField (page 65)

Adds a yes/no field to a form.

NWSAppendCommentField (page 68)

Adds a comment field to a form.

NWSAppendHexField (page 70)

Adds a form field that accepts only hexadecimal input.

NWSAppendHotSpotField (page 73)

Adds a form field that calls a routine when selected.

NWSAppendIntegerField (page 76)

Adds a form field that accpts only a digital number as input.

NWSAppendMenuField (page 79)

Adds a form field that displays a menu when selected.

NWSAppendPasswordField (page 82)

Adds a form field that enables password input.

NWSAppendPromptField (page 86)

Adds a prompt field to a form.

NWSAppendScrollableStringField (page 88)

Appends a form field into which scrollable text can be entered.

NWSAppendStringField (page 92)

Adds a form field containing an editable string.

NWSAppendToForm (page 95)

Adds a customized field to a form.

NWSAppendToList (page 99)

Adds an element to the current list.

NWSAppendToMenu (page 101)

Adds an option to a menu.

NWSAppendToMenuField (page 103)

Adds an option to a menu field.

NWSAppendUnsignedIntegerField (page 105)

Adds a form field that only accepts an unsigned integer as input.

NWSAsciiHexToInt (page 109)

Converts an ASCII-represented hexadecimal number to an integer.

NWSAsciiToInt (page 110)

Converts an ASCII-represented decimal number to an integer.

NWSAsciiToLONG (page 111)

Converts an ASCII-represented number to type LONG.

NWSClearPortal (page 112)

Blanks out a portal.

NWSComputePortalPosition (page 113)

Calculates the screen line and column for positioning a portal given its size.

NWSConfirm (page 115)

Displays a yes/no portal allowing the user to confirm a decision.

NWSCreatePortal (page 117)

Creates a portal.

NWSDeleteFromList (page 122)

Deletes an element from a list.

NWSDeleteFromPortalList (page 124)

Deletes current and marked elements from a list.

NWSDeselectPortal (page 126)

Makes a portal inactive.

NWSDestroyForm (page 127)

Destroys a form.

NWSDestroyList (page 128)

Destroys a list.

NWSDestroyMenu (page 129)

Destroys a menu.

NWSDestroyPortal (page 130)

Destroys a portal.

NWSDisableAllFunctionKeys (page 132)

Disables all function keys.

NWSDisableAllInterruptKeys (page 133)

Disables all interrupt keys.

NWSDisableInterruptKey (page 135)

Disables a specified interrupt key.

NWSDisableFunctionKey (page 134)

Disables a function key.

NWSDisablePortalCursor (page 136)

Disables the portal cursor.

NWSDisplayErrorCondition (page 137)

Displays an error portal listing the routine that resulted in the error condition.

NWSDisplayErrorText (page 140)

Displays an error portal.

NWSDisplayHelpScreen (page 142)

Displays a help portal.

NWSDisplayInformation (page 143)

Displays text in a portal. Palette, video attribute, and behavior can be specified.

NWSDisplayInformationInPortal (page 146)

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 (page 152)

Displays a pre-help message.

NWSDisplayTextInPortal (page 154)

Displays text in an existing portal. Starting portal line for text and indent level can be specified.

NWSDisplayTextJustifiedInPortal (page 156)

Same as above, but text width and video attribute can be specified.

NWSDrawPortalBorder (page 159)

Redraws the border of a portal.

NWSEditForm (page 161)

Displays a form and allows the user to edit it.

NWSEditPortalForm (page 164)

Same as **NWSEditForm**, but help context can be specified.

NWSEditPortalFormField (page 167)

Same as **NWSEditForm**, but help context and first field to highlight can be specified.

NWSEditString (page 170)

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 (page 174)

Displays a string inside a portal and allows the user to edit it using the NWSNUT screen editor.

NWSEditTextWithScrollBars (page 177)

Allows the user to edit scrollable text within a portal.

NWSEnableAllFunctionKeys (page 180)

Enables all function keys.

NWSEnableFunctionKey (page 181)

Enables a function key.

NWSEnableFunctionKeyList (page 182)

Enables a list of function keys.

NWSEnableInterruptKey (page 183)

Enables an interrupt key and associates that key with a routine.

NWSEnableInterruptList (page 185)

Enables a list of interrupt keys.

NWSEnablePortalCursor (page 187)

Enables the portal cursor.

NWSEndWait (page 188)

Removes a "please wait" portal.

NWSFillPortalZone (page 189)

Fills a specified region of the portal with a character.

NWSFillPortalZoneAttribute (page 192)

Fills a specified region of the portal with the video attribute.

NWSFree (page 195)

Frees memory allocated with **NWSAlloc (page 64)**.

NWSGetADisk (page 197)

Displays a portal prompting the user to insert the specified floppy disk into the disk drive.

NWSGetDefaultCompare (page 199)

Obtains the default compare function.

NWSGetFieldFunctionPtr (page 200)

Obtains the customized routines for the specified field.

NWSGetHandleCustomData (page 202)

Obtains the function for handling developer-defined data.

NWSGetKey (page 204)

Reads one key from the keyboard buffer.

NWSGetLineDrawCharacter (page 206)

Gets a line drawing character.

NWSGetList (page 207)

Returns the pointers for the current list.

NWSGetListHead (page 208)

Returns the first element in the current list.

NWSGetListNotifyProcedure (page 211)

Obtains the routine to be called when a list element is highlighted.

NWSGetListSortFunction (page 213)

Returns a pointer to the currently set list sort function.

NWSGetListTail (page 214)

Returns the last element in the current list.

NWSGetMessage (page 215)

Retrieves the specified message.

NWSGetNUTVersion (page 217)

Returns the version of NWSNUT the NLM is using.

NWSGetPCB (page 218)

Obtains the portal control block of a portal.

NWSGetScreenPalette (page 220)

Returns the current screen palette.

NWSGetSortCharacter (page 221)

Returns the weighted value used for sorting a given character.

NWSInitForm (page 223)

Initializes a form.

NWSInitializeNut (page 227)

Sets up NWSNUT context for your NLM.

NWSInitList (page 231)

Initializes a list.

NWSInitListPtr (page 234)

Initializes a list that is appended to another list or form.

NWSInitMenu (page 235)

Initializes a menu.

NWSInitMenuField (page 238)

Initializes a menu field for a form.

NWSInsertInList (page 240)

Inserts an element into a list.

NWSInsertInPortalList (page 242)

Inserts an element into a list using a specified insertion routine.

NWSIsAnyMarked (page 245)

Indicates whether any elements in the current list are marked.

NWSIsdigit (page 247)

Tests whether a character is an ASCII representation of a decimal number.

NWSIsxdigit (page 249)

Tests whether a character is an ASCII representation of a hexadecimal number.

NWSKeyStatus (page 251)

Indicates whether a key is waiting in the keyboard buffer.

NWSList (page 252)

Displays a list and allows the user to perform list operations.

NWSMemmove (page 256)

Copies bytes from one buffer to another.

NWSMenu (page 257)

Displays a menu and allows the user to choose options from it.

NWSModifyInPortalList (page 260)

Modifies the text field of a list element.

NWSPopHelpContext (page 263)

Pops a help context off of the help stack.

NWSPopList (page 265)

Pops a set of list pointers from the list stack.

NWSPopMarks (page 267)

Retrieves the marked status of all list elements in the current list.

NWSPositionCursor (page 268)

Positions the cursor relative to the entire screen.

NWSPositionPortalCursor (page 270)

Positions the portal cursor within the portal.

NWSPromptForPassword (page 271)

Enables a console operator to input a password to an NLM, with optional forced verification.

NWSPushHelpContext (page 274)

Saves a help context onto the help stack.

NWSPushList (page 276)

Pushes the current list pointers onto the list stack.

NWSPushMarks (page 278)

Saves the marked status of all list elements in the current list.

NWSRemovePreHelp (page 279)

Removes a pre-help portal.

NWSRestoreDisplay (page 280)

Clears the screen.

NWSRestoreList (page 281)

Takes the specified list from the save stack and makes it current.

NWSRestoreNut (page 283)

Cleans up resources allocated by NWSNUT for your NLM.

NWSRestoreZone (page 284)

Saves data in a buffer to the screen.

NWSSaveFunctionKeyList (page 287)

Saves a list of function keys.

NWSSaveInterruptList (page 288)

Saves a list of interrupt keys.

NWSSaveList (page 289)

Saves the current list into the specified slot in the save stack.

NWSSaveZone (page 291)

Saves a defined area on the screen to a buffer.

NWSScreenSize (page 293)

Calculates the screen size.

NWSScrollPortalZone (page 294)

Scrolls the portal display area the specified direction and number of lines.

NWSScrollZone (page 296)

Allows a console operator to scroll the contents of a zone in a defined screen area, thus creating new lines.

NWSSelectPortal (page 299)

Makes a portal active.

NWSSetDefaultCompare (page 300)

Sets the default compare function.

NWSSetDynamicMessage (page 302)

Stores a dynamic message into the MessageInfo structure.

NWSSetErrorLabelDisplayFlag (page 304)

Sets the flag that determines whether **NWSDisplayErrorCondition (page 137)** and **NWSDisplayErrorText (page 140)** display NLM name and version information.

NWSSetFieldFunctionPtr (page 305)

Sets the customized routines for a field.

NWSSetFormRepaintFlag (page 308)

Repaints the form to show changes made to the form but not yet reflected on the screen.

NWSSetHandleCustomData (page 310)

Sets the function for handling developer-defined data.

NWSSetList (page 312)

Makes the specified list current.

NWSSetListNotifyProcedure (page 313)

Sets the routine to be called when a list element is highlighted.

NWSSetListSortFunction (page 315)

Enables the use of a customized list sort function.

NWSSetScreenPalette (page 317)

Sets the screen palette.

NWSShowLine (page 320)

Displays text at a specified screen location.

NWSShowLineAttribute (page 322)

Identical to **NWSShowLine (page 320)**, but also allows screen attribute specification.

NWSShowPortalLine (page 324)

Displays a line of text. Portal line and column can be specified.

NWSShowPortalLineAttribute (page 326)

Displays text with a specified screen attribute.

NWSSortList (page 328)

Sorts list elements.

NWSStartWait (page 329)

Displays a "please wait" portal.

NWSStrcat (page 331)

Appends a copy of one string to the end of another.

NWSToupper (page 332)

Returns the uppercase value of the specified byte.

NWSTrace (page 333)

Displays an information portal and waits for Esc to be pressed.

NWSUngetKey (page 335)

Inserts a key into the keyboard buffer.

NWSUnmarkList (page 337)

Removes marks from a list.

NWSUpdatePortal (page 338)

Redraws a virtual portal to show changes made since creation or last update.

NWSViewText (page 339)

Displays text within a portal.

NWSViewTextWithScrollBars (page 341)

Displays scrollable text within a portal.

NWSWaitForEscape (page 344)

Waits for the Esc key to be pressed.

NWSWaitForEscapeOrCancel (page 345)

Waits for the Esc or Cancel (F7) key to be pressed.

NWSWaitForKeyAndValue (page 347)

Waits until the user presses one of the keys in a specified set.

2 NWSNUT Tasks

This documentation describes common tasks associated with NWSNUT.

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 illustration in "The NWSNUT Environment" on page 15). 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" on page 18
- "Positioning the Portal" on page 19
- "Screen Palette" on page 19

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 screen ****/
```

The example shows just one way to write to a portal. See "Basic Portal Functions" on page 22 for a summary of functions used for writing to portals.

For more information, see the following:

- "Selecting Portals" on page 19
- "The Portal Cursor" on page 20
- "Scrolling the Portal" on page 20
- "Line Drawing Characters" on page 20

4. Destroy the portal.

```
NWSDestroyPortal (portalNumber, handle);
```

NWSDestroyPortal (page 130) destroys the portal and cleans up all resources used by the portal.

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.

   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 (page 240) or **NWSInsertInPortalList (page 242)** 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 (page 252) 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 can 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. 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 (page 234)** 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.

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 (page 101) 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 (page 257)** 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 (page 257)**, 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 (page 129) frees the nodes and menu pointers for the menu.

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 (page 223) 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 Type Functions" on page 31.

For more information, see the following:

- "Form Field Type Functions" on page 31
- "Field Structure" on page 27
- "Prompt Fields" on page 30
- "Menu Fields" on page 30
- "Custom Fields" on page 31
- "Form Editing Functions" on page 32

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" on page 32).

4 Destroy the form.

```
NWSDestroyForm (handle);
```

NWSDestroyForm (page 127) frees all of the fields in the current form and reinitializes form pointers.

3 NWSNUT Functions

This alphabetically lists the NWSNUT functions and describes their purpose, syntax, parameters, and return values.

NWSAlert to NWSAppend* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSAlert

Displays an alert portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

message

(IN) Specifies the message identifier of the alert message.

Return Values

The following table lists return values and descriptions.

(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 (page 60)

NWSAlertWithHelp

Displays an alert portal with access to help screens.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

(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 (page 58)

NWSAlignChangedList

Aligns the list display line with the portal frame.

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.11, 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 122), **NWSGetListIndex (page 209)**

NWSAlloc

Allocates memory for NWSNUT purposes.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 195)

NWSAppendBoolField

Appends a boolean choice field to a form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 352) 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.

Value	Meaning
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 (page 223)

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendCommentField

Appends a comment field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (text must be available as long as the form is used).

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

The following table lists return values and descriptions.

NULL	Unable to append field to form
not NULL	Pointer to FIELD item that has been appended to form

Remarks

The comment field pointed to by *prompt* is a text string that appears in a form, but is not available for editing by the user. It is often used as a prompt. This text is not copied but only pointed to, and must therefore be available as long as the form is used.

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 (page 352) structure is described in "NWSNUT Structures" on page 351.

See Also

NWSAppendPromptField (page 86), **NWSInitForm (page 223)**

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendHexField

Appends a hexadecimal field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 352) structure. See **NWSAppendBoolField (page 65)** 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 (page 105),
NWSAppendIntegerField (page 76), **NWSInitForm (page 223)**

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendHotSpotField

Appends a hot spot field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the string to be displayed in the form to mark the hot spot.

SpotAction

(IN) Points to 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

The following table lists return values and descriptions.

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 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 (page 65)** or nwsnut.h for values.

The *SpotAction* parameter sets the *fieldControl* field in the FIELD (page 352) 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 (page 223)

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendIntegerField

Appends an integer field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 352) structure. See **NWSAppendBoolField (page 65)** 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 (page 70), **NWSAppendUnsignedIntegerField (page 105)**, **NWSInitForm (page 223)**

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendMenuField

Appends a menu field to a form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 65)** 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 (page 103), **NWSInitMenuField (page 238)**

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendPasswordField

Appends a password field to a form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Specifies 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

The following table lists return values and descriptions.

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 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 (page 271)

NWSAppendPromptField

Appends a prompt field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 68), **NWSInitForm (page 223)**

Example

See the example for **NWSInitForm (page 223)**.

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, 5.x

Platform: NLM

Service: NWSNUT

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 (page 352) 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

The following table lists return values and descriptions.

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 uppercase entries to lowercase)
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 =
EF_MASK	Causes each input character to be echoed as an asterisk (available on NWSNUT versions 4.04 and later—see NWSGetNUTVersion (page 217))

If help is specified for a form field, it is not displayed unless the user presses Enter, followed by F1.

See Also

NWSAppendStringField (page 92), NWSEditString (page 170)

NWSAppendStringField

Appends a string field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 352) structure. See **NWSAppendBoolField (page 65)** 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 (page 223)

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendToForm

Appends a customized field to a form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the routine to format field, (NULL for default).

fControl

(IN) Points to the routine to handle normal key input for this field, (NULL for default).

fVerify

(IN) Points to the routine to verify field contents after editing, (NULL for default).

fRelease

(IN) Points to the routine to free the *fData* and *fXtra* memory and release these parameters (NULL for default).

fData

(IN/OUT) Points to 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) Points to 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

The following table lists return values and descriptions.

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 (page 352) structure. See **NWSAppendBoolField (page 65)** 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 (page 223)

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendToList

Appends an item and its customized data to the current list.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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

The following table lists return values and descriptions.

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 (page 358) structure.

See Also

NWSAppendToList (page 99), **NWSInitList (page 231)**, **NWSList (page 252)**, **NWSSetListNotifyProcedure (page 313)**

Example

See the example for **NWSInitList (page 231)**.

NWSAppendToMenu

Adds an option to the current menu.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 358) that defines a menu item within the current menu.

See Also

NWSDestroyMenu (page 129), **NWSInitMenu (page 235)**, **NWSMenu (page 257)**

Example

See the example for **NWSInitMenu (page 235)**.

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 363) 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 (page 79), **NWSInitMenuField (page 238)**

Example

See the example for **NWSInitForm (page 223)**.

NWSAppendUnsignedIntegerField

Appends an unsigned integer field to the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 352) structure. See **NWSAppendBoolField (page 65)** 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 (page 70), **NWSAppendIntegerField (page 76)**, **NWSInitForm (page 223)**

Example

See the example for **NWSInitForm (page 223)**.

NWSAscii* to NWSDestroy* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSAsciiHexToInt

Converts an ASCII-represented hexadecimal number to an integer.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 110), **NWSAsciiToLONG (page 111)**

NWSAsciiToInt

Converts an ASCII-represented decimal number to an integer.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 109), **NWSAsciiToLONG (page 111)**

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, 5.x

Platform: NLM

Service: NWSNUT

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 (page 109), **NWSAsciiToInt (page 110)**

NWSClearPortal

Blanks out the specified portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSClearPortal  (
   PCB   * portal);
```

Parameters

portal

(IN) Points to a portal control block.

Remarks

This function blanks out the portal specified by *portal*. The *portal* parameter can be obtained by calling **NWSGetPCB**.

See Also

NWSCreatePortal (page 117), **NWSDestroyPortal (page 130)**, **NWSGetPCB (page 218)**

NWSComputePortalPosition

Calculates the screen line and column for positioning a portal.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the zero-based top row of the portal.

column

(OUT) Points to 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

The following table lists return values and descriptions.

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 (page 117)

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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) Points to an optional parameter for the action procedure.

Return Values

The following table lists return values and descriptions.

-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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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) Specifies 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:

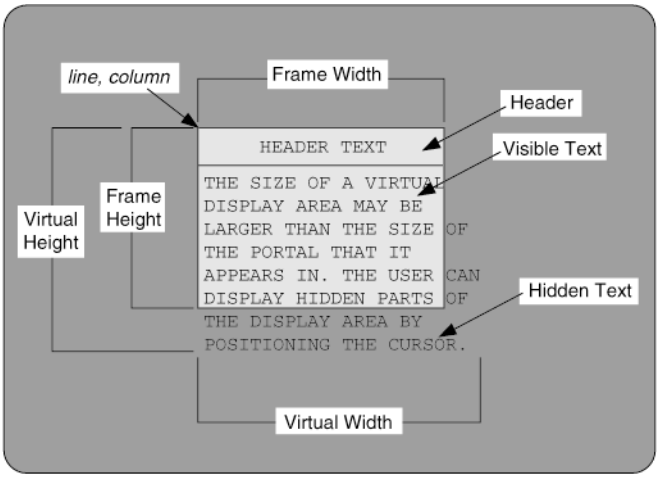
(0xFFFFFFFE)	Unable to allocate memory for PCB, virtual screen, or save area.
(0xFFFFFFFF)	Maximum number of portals already defined.

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.

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 (page 112), **NWSDestroyPortal (page 130)**, **NWSDisplayTextInPortal (page 154)**, **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSUpdatePortal (page 338)**

NWSDeleteFromList

Removes the specified element from the current list.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 62), **NWSAppendToList (page 99)**, **NWSDeleteFromPortalList (page 124)**, **NWSGetListIndex (page 209)**, **NWSInsertInList (page 240)**

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the element to be deleted.

currentLine

(IN) Points to 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

The following table lists return values and descriptions.

-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 (page 99), **NWSDeleteFromList (page 122)**, **NWSInsertInList (page 240)**

NWSDeselectPortal

Deselects the currently active portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function deselects the current portal and unhighlights its border. After this function is called, there is no current portal.

See Also

NWSSelectPortal (page 299)

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function destroys the current form and reinitializes form pointers.

See Also

NWSInitForm (page 223)

Example

See the example for **NWSInitForm (page 223)**.

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function destroys the current list and reinitializes the current list pointers.

See Also

NWSInitList (page 231), **NWSList (page 252)**

Example

See the example for **NWSInitList (page 231)**.

NWSDestroyMenu

Destroys the current menu.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function frees the nodes in the current menu and frees the menu pointers.

See Also

NWSInitMenu (page 235), **NWSMenu (page 257)**

Example

See the example for **NWSInitMenu (page 235)**.

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

The portal index number is returned by **NWSCreatePortal**.

See Also

NWSCreatePortal (page 117)

NWSDisable* to NWSDraw* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSDisableAllFunctionKeys

Disables all function keys.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 134), **NWSEnableAllFunctionKeys (page 180)**, **NWSEnableFunctionKey (page 181)**, **NWSEnableFunctionKeyList (page 182)**, **NWSSaveFunctionKeyList (page 287)**

NWSDisableAllInterruptKeys

Disables all interrupt keys.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function disables all previously defined interrupt keys and destroys the association between the keys and their associated routines.

See Also

NWSEnableInterruptKey (page 183), **NWSEnableInterruptList (page 185)**, **NWSSaveInterruptList (page 288)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 132), **NWSEnableFunctionKey (page 181)**, **NWSEnableFunctionKeyList (page 182)**, **NWSSaveFunctionKeyList (page 287)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function destroys the association between a defined interrupt key and a specified function. That association is established by calling **NWSEnableInterruptKey**.

See Also

NWSDisableAllInterruptKeys (page 133), **NWSEnableInterruptList (page 185)**, **NWSEnableInterruptKey (page 183)**, **NWSSaveInterruptList (page 288)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

The *portal* parameter can be obtained by calling **NWSGetPCB**.

See Also

NWSEnablePortalCursor (page 187)

NWSDisplayErrorCondition

Displays an error box listing the name of the routine that resulted in the error condition, and displays an appropriate error message.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSDisplayErrorCondition  (
   BYTE       *procedureName,
   int         errorCode,
   LONG        severity,
   PROCERROR   * errorList,
   NUTInfo     * handle,
   ...);
```

Parameters

procedureName

(IN) Points to 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.

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 (page 140), **NWSSetErrorLabelDisplayFlag (page 304)**

NWSDisplayErrorText

Displays an error portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 137),
NWSSetErrorLabelDisplayFlag (page 304)

NWSDisplayHelpScreen

Displays a help portal on the screen.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 263), **NWSPushHelpContext (page 274)**

NWSDisplayInformation

Displays text in a portal on the screen.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

(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 (page 146), **NWSDisplayTextInPortal (page 154)**, **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSViewText (page 339)**

NWSDisplayInformationInPortal

Displays text in a portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 143), **NWSDisplayTextInPortal (page 154)**, **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSViewText (page 339)**

NWSDisplayPreHelp

Displays a prehelp portal on the screen.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 279)

NWSDisplayTextInPortal

Displays text in an existing portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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 (page 143), **NWSDisplayInformationInPortal (page 146)**, **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSViewText (page 339)**

NWSDisplayTextJustifiedInPortal

Displays justified text in an existing portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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 (page 143), **NWSDisplayInformationInPortal (page 146)**, **NWSDisplayTextInPortal (page 154)**, **NWSViewText (page 339)**

NWSDrawPortalBorder

Draws a border for the specified portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSDrawPortalBorder  (
   PCB   * portal);
```

Parameters

portal

(IN) Points to the portal control block for the portal to have the border.

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**.

NWSEdit* to NWSFree Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSEditForm

Displays the current form and allows the user to edit it.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 164), **NWSEditPortalFormField (page 167)**, **NWSInitForm (page 223)**

NWSEditPortalForm

Displays the current form and allows the user to edit it.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

–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 (page 161), **NWSEditPortalFormField (page 167)**, **NWSInitForm (page 223)**

Example

See the example for **NWSInitForm (page 223)**.

NWSEditPortalFormField

Displays the current form and allows the user to edit it.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

–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 (page 161), **NWSEditPortalForm (page 164)**, **NWSInitForm (page 223)**

NWSEditString

Allows the user to edit a string in a portal

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (pass NO_MESSAGE if a header is not wanted).

prompt

(IN) Specifies the message identifier for prompt text.

buf

(IN/OUT) Points to the display text on input. 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

NWSEditString 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 *header* parameter specifies the message to be displayed in the header. It can take any of up to 14 dynamic messages that can be defined to correspond to DYNAMIC_MESSAGE_ONE through DYNAMIC_MESSAGE_FOURTEEN, or can take the system message indicated by SYSTEM_MESSAGE. If no header is desired, passing NO_MESSAGE supresses the display of a header at all.

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 uppercase.
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.
EF_MASK	Causes each input character to be echoed as an asterisk (available on NWSNUT versions 4.04 and later—see NWSGetNUTVersion (page 217))
EF_NOCONFIRM_EXIT	Suppresses confirmation portal when <Esc> key is pressed after string data has been edited

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 (page 174)

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the text to display on input (can be changed by the user during the edit process) and needs to be zero-terminated.

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:

Value	Name	Description
<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 (page 170)

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, 5.x

Platform: NLM

Service: NWSNUT

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) Inputs a pointer to the text to display (can be changed by the user during the edit process) and needs to be zero-terminated.

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:

Value	Name	Description
<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 (page 341)**.

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 (page 161), **NWSEditText (page 174)**, **NWSEditPortalForm (page 164)**, **NWSEditPortalFormField (page 167)**, **NWSInitForm (page 223)**

Example

```
ccode = NWSEditTextWithScrollBars(0, 0, 18, 78, DYNAMIC_MESSAGE_ONE,
     data, fileSize+1, DYNAMIC_MESSAGE_TWO, FALSE, SHOW_VERTICAL_SCROLL_BAR |
      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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function reverses the effect of **NWSDisableAllFunctionKeys**.

See Also

NWSDisableAllFunctionKeys (page 132), **NWSDisableFunctionKey (page 134)**, **NWSEnableFunctionKey (page 181)**, **NWSEnableFunctionKeyList (page 182)**, **NWSSaveFunctionKeyList (page 287)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function reverses the effect of **NWSDisableFunctionKey**.

See Also

NWSDisableAllFunctionKeys (page 132), **NWSDisableFunctionKey (page 134)**, **NWSEnableAllFunctionKeys (page 180)**, **NWSEnableFunctionKeyList (page 182)**, **NWSSaveFunctionKeyList (page 287)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function enables the function keys specified by the *keyList* parameter.

See Also

NWSDisableAllFunctionKeys (page 132), **NWSDisableFunctionKey (page 134)**, **NWSEnableAllFunctionKeys (page 180)**, **NWSEnableFunctionKey (page 181)**, **NWSSaveFunctionKeyList (page 287)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 133), **NWSDisableInterruptKey (page 135)**, **NWSEnableInterruptList (page 185)**, **NWSSaveInterruptList (page 288)**

NWSEnableInterruptList

Enables the list of interrupt keys saved in the INTERRUPT structure.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 133), **NWSEnableInterruptKey (page 183)**, **NWSSaveInterruptList (page 288)**

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSEnablePortalCursor  (
   PCB   * portal);
```

Parameters

portal

(IN) Points to a NWSNUT portal control block.

Remarks

This function enables the cursor of *portal*. The *portal* parameter can be obtained by calling **NWSGetPCB**.

See Also

NWSDisablePortalCursor (page 136)

NWSEndWait

Removes the wait portal displayed by **NWSStartWait** .

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function removes a wait portal created by **NWSStartWait**.

See Also

NWSStartWait (page 329)

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 192)

Example

```
#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 the screen*/
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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 189)

Example

```
#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 screen */
   NWSGetKey (&type, &value, handle);
                                          /* wait for a key to be pressed */
   NWSDestroyPortal (portalNumber, handle);
}
```

NWSFree

Frees memory allocated by **NWSAlloc** .

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function frees memory allocated by **NWSAlloc** at *address*.

See Also

NWSAlloc (page 64)

NWSGet* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

LONG NWSGetADisk  (
   BYTE     *volName,
   BYTE     *prompt,
   NUTInfo   * handle);
```

Parameters

volName

(IN) Points to the floppy volume name.

prompt

(IN) Points to the prompt to display.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

The following table lists return values and descriptions.

`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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the current default compare function.

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 (page 300), **NWSSortList (page 328)**

NWSGetFieldFunctionPtr

Obtains the routines associated with the specified field in a form.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the formatting routine for *fp*.

Control

(OUT) Points to the control routine for *fp*.

Verify

(OUT) Points to the verify routine for *fp*.

Release

(OUT) Points to the memory release routine for *fp*.

Entry

(OUT) Points to the routine to be called for all entries in the form.

customDataRelease

(OUT) Points to the routine to release memory for releasing memory allocated for custom data for *fp*.

Remarks

To set the routines for the field specified by the *fp* parameter, call **NWSSetFieldFunctionPtr**.

See Also

NWSSetFieldFunctionPtr (page 305)

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the custom data held in the NUTInfo structure.

customDataRelease

(OUT) Points to the custom data release function held in the NUTInfo structure.

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 (page 310)

NWSGetKey

Reads one key from the keyboard buffer.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSGetKey  (
   LONG     *type,
   BYTE     *value,
   NUTInfo   * handle);
```

Parameters

type

(OUT) Points to the retrieved key type.

value

(OUT) Points to the retrieved key value.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

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 (page 251), NWSUngetKey (page 335)

NWSGetLineDrawCharacter

Gets a line drawing character.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSGetList  (
   LISTPTR   * listPtr,
   NUTInfo   *handle);
```

Parameters

listPtr

(IN/OUT) Points to 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.

Remarks

This function returns the list pointer to the current list in the *listPtr* parameter.

See Also

NWSGetListHead (page 208), **NWSSetList (page 312)**

NWSGetListHead

Returns the first element in the current list.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 207), **NWSGetListTail (page 214)**

NWSGetListIndex

Returns the index of the specified list element.

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 62), **NWSGetList (page 207)**, **NWSGetListHead (page 208)**, **NWSGetListTail (page 214)**

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the currently defined notify procedure.

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 (page 313)

NWSGetListSortFunction

Returns a pointer to the currently set list sort function.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 315)

NWSGetListTail

Returns the last element in the current list.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

See Also

NWSGetListHead (page 208)

NWSGetMessage

Retrieves a message from the specified message buffer.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 302)

NWSGetNUTVersion

Returns the version of NWSNUT currently loaded on the server.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 370) structure is defined in nwsnut.h.

Fields in the PCB (page 370) structure should not be changed directly. Call NWSNUT functions for creating and manipulating portals.

See Also

NWSCreatePortal (page 117)

NWSGetScreenPalette

Returns the current screen palette.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 317)

NWSGetSortCharacter

Returns the weighted value used for sorting a given character.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

NWSInit* to NWSModify* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSInitForm

Initializes pointers for the current form.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function initializes the pointers in the first FIELD (page 352) 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**.

Fields in the FIELD (page 352) structure should not be changed directly. Call NWSNUT functions for building and manipulating forms.

See Also

NWSAppendBoolField (page 65), **NWSAppendCommentField (page 68)**, **NWSAppendHexField (page 70)**, **NWSAppendHotSpotField (page 73)**, **NWSAppendIntegerField (page 76)**, **NWSAppendMenuField (page 79)**,

NWSAppendStringField (page 92), **NWSAppendToForm (page 95)**, **NWSAppendToMenuField (page 103)**, **NWSAppendUnsignedIntegerField (page 105)**, **NWSEditPortalForm (page 164)**, **NWSInitMenuField (page 238)**

Example

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 ", NULL,
      handle);
   line += 2;
   NWSAppendCommentField (line, 1, "Unsigned Integer Field:", handle);
   NWSAppendUnsignedIntegerField (line, 25, NORMAL_FIELD, &myOtherInteger,
                                  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 parameter*/
   NWSEditPortalForm (FORM_HEADER, 11, 40, 16, 50, F_NOVERIFY, FORM_HELP,
      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 option
      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, NUTInfo
*handle)
{
   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, 5.x

Platform: NLM

Service: NWSNUT

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 (pass -1 to signal no message identifier).

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) Points to a message table containing program messages for your NLM (see Remarks below).

helpScreens

(IN) Points 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

The following table lists return values and descriptions.

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 Also

NWSRestoreNut (page 283)

Example

```
#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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

If you have previously built a list, you must save or destroy it before calling **NWSInitList**. **NWSInitList** initializes a new LISTPTR (page 361) 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 (page 99), NWSDeleteFromList (page 122), NWSDeleteFromPortalList (page 124), NWSDestroyList (page 128), NWSGetList (page 207), NWSInsertInList (page 240), NWSInsertInPortalList (page 242), NWSList (page 252), NWSModifyInPortalList (page 260), NWSSaveList (page 289), NWSRestoreList (page 281), NWSSetList (page 312)

Example

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 *itemNumber,
      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 *itemNumber,
      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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSInitListPtr  (
   LISTPTR   * listPtr);
```

Parameters

listPtr

(IN) Points to the list pointer to be initialized.

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function initializes an MFCONTROL (page 363) structure for a menu.

To build a list of options for the menu, call **NWSAppendToMenu** for each option.

See Also

NWSAppendToMenu (page 101), **NWSMenu (page 257)**

Example

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 "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 *)handle);
   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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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 (page 363) 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 (page 79), **NWSAppendToMenuField (page 103)**

Example

See the example for **NWSInitForm (page 223)**.

NWSInsertInList

Inserts a new list element after the specified list element.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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 (page 99), **NWSInsertInPortalList (page 242)**, **NWSModifyInPortalList (page 260)**

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 99), **NWSInsertInList (page 240)**, **NWSModifyInPortalList (page 260)**

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 267), **NWSPushMarks (page 278)**, **NWSUnmarkList (page 337)**

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

int NWSIsdigit  (
   BYTE   character);
```

Parameters

character

(IN) Specifies the ASCII value of the byte in question.

Return Values

The following table lists return values and descriptions.

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 (page 249)

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

int NWSIsxdigit  (
   BYTE   character);
```

Parameters

character

(IN) Specifies the BYTE in question.

Return Values

The following table lists return values and descriptions.

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 (page 247)

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the element to highlight as default.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

format

(IN) Points to an optional routine to format list elements.

action

(IN) Points to an optional routine to be called after key is pressed.

actionParm

(IN) Points to 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 (page 99), **NWSDestroyList (page 128)**, **NWSInitList (page 231)**

Example

See the example for **NWSInitList (page 231)**.

NWSMemmove

Copies bytes from one buffer to another buffer.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the element to highlight as the default selection.

action

(IN) Points to 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) Points to 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 (page 101), **NWSDestroyMenu (page 129)**, **NWSInitMenu (page 235)**

Example

See the example for **NWSInitMenu (page 235)**.

NWSModifyInPortalList

Modifies the text field of an element.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 99), **NWSInsertInList (page 240)**, **NWSInsertInPortalList (page 242)**

NWSPop* to NWSRestore* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSPopHelpContext

Removes the last element from the help stack.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

0	Success
Nonzero	Failure

Remarks

This function removes the last help context from the help stack.

See Also

NWSDisplayHelpScreen (page 142), **NWSPushHelpContext (page 274)**

Example

```
#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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 276), **NWSRestoreList (page 281)**, **NWSSaveList (page 289)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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, the elements are considered unmarked.

See Also

NWSPushMarks (page 278)

NWSPositionCursor

Sets the position of the cursor relative to the entire screen.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the screen to use for your NLM.

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 the"See Also" list.

NWSPositionCursor can work in harmony with any of the related functions listed in the following "See Also" list.

See Also

NWSEnablePortalCursor (page 187), **NWSDisablePortalCursor (page 136)**, **NWSPositionPortalCursor (page 270)**

Example

```
ccode = NWSPositionCursor(12, 40, handle->screenID);
```

NWSPositionPortalCursor

Positions the cursor for an NWSNUT portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSPositionPortalCursor  (
   LONG   line,
   LONG   column,
   PCB    * portal);
```

Parameters

line

(IN) Specifies the portal line where cursor is to be positioned.

column

(IN) Specifies the portal column where cursor is to be positioned.

portal

(IN) Points to an NWSNUT portal control block.

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, 5.x

Platform: NLM

Service: NWSNUT

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) Specifies designated the maximum length of the password string.

passwordString

(IN) Points to a null terminated string to be used as the password.

verifyPassword

(IN) Specifies enables or disables password verification.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Return Values

The following table lists return values and descriptions.

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 (page 82)

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 a .hlh file and the messages are stored in a .hlp file.

Up to MAXHELP help contexts can be saved to the help stack.

See Also

NWSDisplayHelpScreen (page 142), **NWSPopHelpContext (page 263)**

Example

```
#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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 265), **NWSRestoreList (page 281)**, **NWSSaveList (page 289)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 267)

NWSRemovePreHelp

Removes the current pre-help portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function reverses the effect of **NWSDisplayPreHelp**.

See Also

NWSDisplayPreHelp (page 152)

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSRestoreDisplay  (
   struct ScreenStruct  * screenID);
```

Parameters

screenID

(IN) Specifies the screen to use for your NLM.

Return Values

None.

NWSRestoreList

Makes the specified list the current list.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 265), **NWSPushList (page 276)**, **NWSSaveList (page 289)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 227)

NWSRestoreZone

Restores data in a buffer to the screen.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the screen to use for your NLM.

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 (page 268)**.

See Also

NWSSaveZone (page 291), **NWSScrollZone (page 296)**

NWSSave* to NWSSet* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSSaveFunctionKeyList

Saves the current list of enabled function keys.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function saves saves the enabled function keys into *keyList*.

See Also

NWSDisableFunctionKey (page 134), **NWSEnableFunctionKey (page 181)**, **NWSEnableFunctionKeyList (page 182)**

NWSSaveInterruptList

Saves the context of the current interrupt keys.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 133), **NWSEnableInterruptKey (page 183)**, **NWSEnableInterruptList (page 185)**

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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.

Take care that you do not save another list to the same slot, because the first list is overwritten.

See Also

NWSPopList (page 265), **NWSPushList (page 276)**, **NWSRestoreList (page 281)**

NWSSaveZone

Saves a defined area on the screen to a buffer.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the screen to use for your NLM.

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 (page 268)**.

See Also

NWSRestoreZone (page 284), **NWSScrollZone (page 296)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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**.

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the screen to use for your NLM.

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 (page 268)**.

See Also

NWSSaveZone (page 291), NWSRestoreZone (page 284)

NWSSelectPortal

Selects a portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 117), **NWSDeselectPortal (page 126)**

NWSSetDefaultCompare

Specifies a routine for sorting list elements.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 199)

NWSSetDynamicMessage

Stores dynamic messages in the NWSNUT interface message table.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function does not copy the message text, but sets the appropriate pointer in the MessageInfo (page 362) 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.

NWSSetErrorLabelDisplayFlag

Sets the *displayErrorLabel* field of the NUTInfo structure.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 137), **NWSDisplayErrorText (page 140)**

NWSSetFieldFunctionPtr

Changes functions associated with a field in a form.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the format routine for the field. If no routine is wanted, specify NULL.

Control

(IN) Points to the control routine for the field. If no routine is wanted, specify NULL.

Verify

(IN) Points to the verify routine for the field. If no routine is wanted, specify NULL.

Release

(IN) Points to the memory release routine for the field. If no routine is wanted, specify NULL.

Entry

(IN) Points to a routine to be called for all fields in the form. If no routine is wanted, specify NULL.

customDataRelease

(IN) Points to a routine to be used to release memory allocated for custom data associated with the field. If no routine is wanted, specify NULL.

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.

Routine	Parameters	
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 (page 200), NWSInitForm (page 223)

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSSetFormRepaintFlag  (
   LONG      value,
   NUTInfo   * handle);
```

Parameters

value

(IN) Specifies this 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.

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 (page 161), **NWSEditPortalForm (page 164)**, **NWSEditPortalFormField (page 167)**

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the custom data to be held in the NUTInfo structure.

customDataRelease

(IN) Points to the custom data release function to be held in the NUTInfo structure.

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 (page 202)

NWSSetList

Changes the current list to point to the new list information.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSSetList  (
   LISTPTR   * listPtr,
   NUTInfo   *handle);
```

Parameters

listPtr

(IN) Points to the new list pointers to be made current.

handle

(IN) Points to a NUTInfo structure that contains state information allocated to the calling NLM.

Remarks

This function makes the list specified by *listPtr* the current list. The *listPtr* parameter can be obtained from **NWSGetList**.

See Also

NWSGetList (page 207)

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the routine to be called when *el* is selected.

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 (page 211)

NWSSetListSortFunction

Specifies a custom list sorting function.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 in NUTInfo. *listSortFunction*.

See Also

NWSGetListSortFunction (page 213)

NWSSetScreenPalette

Changes the screen palette.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

The following bits are defined for the *palette* parameter:

NORMAL_PALETTE
INIT_PALETTE
HELP_PALETTE
ERROR_PALETTE
WARNING_PALETTE

OTHER_PALETTE

NWSShow* to NWSWait* Functions

Click any function name in the table of contents to view the purpose, syntax, parameters, and return values for that function.

NWSShowLine

Displays a text string at a specified screen location.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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) Points to the screen to use for your NLM.

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 (page 268)**.

NWSShowLine does not alter the attribute bytes. To specify an attribute in the line to be shown, call **NWSShowLineAttribute**.

See Also

NWSShowLineAttribute (page 322), **NWSShowPortalLine (page 324)**

NWSShowLineAttribute

Identical to **NWSShowLine** , but also allows screen attribute specification.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSShowLineAttribute  (
   LONG                    line,
   LONG                   column,
   BYTE                  *text,
   LONG                   attribute,
   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.

attribute

(IN) Specifies the screen attribute for the text.

length

(IN) Specifies the screen display length in which the string pointed to by *text* appears.

screenID

(IN) Points to the screen to use for your NLM.

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 (page 268)**.

See Also

NWSShowLine (page 320)

NWSShowPortalLine

Places screen output in the specified portal.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 154), **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSShowPortalLineAttribute (page 326)**, **NWSUpdatePortal (page 338)**

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, 5.x

Platform: NLM

Service: NWSNUT

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) Points to 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.

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 (page 154), **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSShowPortalLine (page 324)**, **NWSUpdatePortal (page 338)**

NWSSortList

Sorts the current list alphabetically.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

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 (page 199), **NWSSetDefaultCompare (page 300)**

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, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

To destroy this portal, call **NWSEndWait**. Keys are not disabled by this function.

See Also

NWSEndWait (page 188)

NWSStrcat

Appends a copy of one string to the end of another.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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, 5.x

Platform: NLM

Service: NWSNUT

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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

(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, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

-1	Keyboard buffer is full
0	Key was inserted

Remarks

This function reverses the effect of **NWSGetKey**.

See Also

NWSGetKey (page 204), **NWSKeyStatus (page 251)**

NWSUnmarkList

Removes the mark status from the current list.

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This removes the marked status from all items in a list.

See Also

NWSIsAnyMarked (page 245), **NWSPopMarks (page 267)**, **NWSPushMarks (page 278)**

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, 5.x

Platform: NLM

Service: NWSNUT

Syntax

```
#include <nwsnut.h>

void NWSUpdatePortal  (
   PCB   * portal);
```

Parameters

portal

(IN) Points to the PCB of the portal.

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, 5.x

Platform: NLM

Service: NWSNUT

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 (needs to be zero-terminated).

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 (page 143), **NWSDisplayInformationInPortal (page 146)**, **NWSDisplayTextInPortal (page 154)**, **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSEditText (page 174)**

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, 5.x

Platform: NLM

Service: NWSNUT

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 (needs to be zero-terminated).

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

The following table lists return values and descriptions.

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 (page 143), **NWSDisplayInformationInPortal (page 146)**, **NWSDisplayTextInPortal (page 154)**, **NWSDisplayTextJustifiedInPortal (page 156)**, **NWSEditText (page 174)**, **NWSViewText (page 339)**

NWSWaitForEscape

Waits for the escape key to be pressed.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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.

Remarks

This function waits for the escape key to be pressed before going to the next instruction.

See Also

NWSWaitForEscapeOrCancel (page 345)

NWSWaitForEscapeOrCancel

Waits for the escape or cancel (F7) key to be pressed.

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 4.x, 5.x

Platform: NLM

Service: NWSNUT

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

The following table lists return values and descriptions.

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 (page 344)

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, 5.x

Platform: NLM

Service: NWSNUT

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.

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 e */
```

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};
```

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 (page 344), NWSWaitForEscapeOrCancel (page 345)

4 NWSNUT Structures

This alphabetically lists the NWSNUT structures and describes their purpose, syntax, and fields.

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

Fields in this structure should not be changed directly. Use NWSNUT functions for building and manipulating forms.

element

Points to NWSNUT-internal information.

fieldFlags

Specifies this is set with various "Append" functions (such as **NWSAppendBoolField (page 65)**) 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

Specifies 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

Specifies 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

Specifies 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

Specifies the display attribute for the form field. This can be set when appending a custom field with **NWSAppendToForm (page 95)**.

fieldActivateKeys

Specifies the keys that activate the form field. This field can be set when appending a custom field with **NWSAppendToForm (page 95)**.

fieldFormat

Points to the routine used to format the form field.

fieldControl

Points to the routine that is called when the form field is selected.

fieldVerify

Points to the routine that is called when the form field is selected.

fieldRelease

Points to the routine called to release memory allocated for *fieldData* and *fieldExtra*.

fieldData

Points to 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 (page 95)**.

fieldXtra

Points to additional control information associated with the form field. This field can be set when a custom field is appended to a form with **NWSAppendToForm (page 95)**.

fieldHelp

Specifies the help context for the form field. You can specify the help context when the form is displayed by **NWSEditPortalForm (page 164)** or **NWSEditPortalFormField (page 167)**. A help context can be specified for a field when it is created by **NWSAppendToForm (page 95)**.

fieldAbove

Points to NWSNUT-internal positioning information.

fieldBelow

Points to NWSNUT-internal positioning information.

fieldLeft

Points to NWSNUT-internal positioning information.

fieldRight

Points to NWSNUT-internal positioning information.

fieldPrev

Points to NWSNUT-internal positioning information.

filedNext

Points to NWSNUT-internal positioning information.

fieldEntry

Points to a routine to be called upon entry to each field in the form.

customData

Points to user-defined data to be attached to the form field.

customDataRelease

Contains a routine to release data in *customData*. The parameters match **NWSFree (page 195)** so that **NWSAlloc (page 64)** can be used to allocate memory for *customData*, further guaranteeing that memory is freed.

nutInfo

Points to 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 (page 200)**; to change the routines, call **NWSSetFieldFunctionPtr (page 305)**. These fields can also be set when a custom field is added to a form by **NWSAppendToForm (page 95)**.

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

Specifies the interrupt key value.

Remarks

This structure is used by **NWSEnableInterruptList (page 185)**.

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

Fields in the LIST (page 358) 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

Points to developer-defined data set by calling **NWSAppendToList (page 99)**.

marked

Specifies 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 (page 337)**. To determine whether any items in a list have been marked, call **NWSIsAnyMarked (page 245)**.

flags

Reserved for NWSNUT.

entryProcedure

Points to a routine to call if this list item is selected. This field can be obtained by calling **NWSGetListNotifyProcedure (page 211)** and set by **NWSSetListNotifyProcedure (page 313)**.

extra

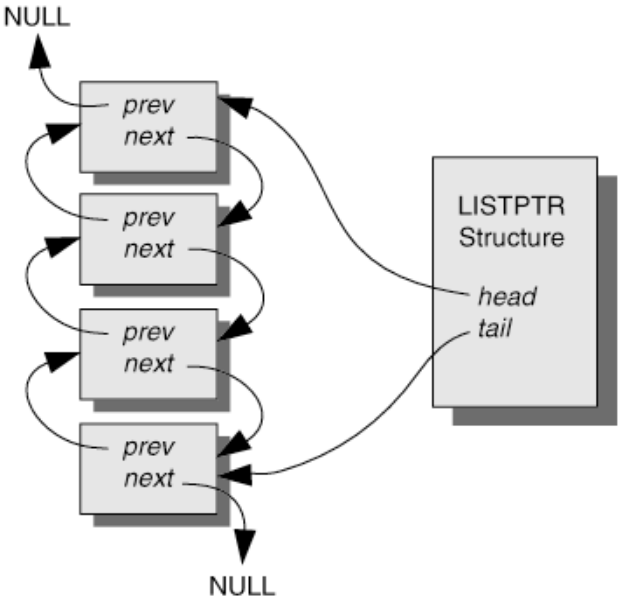
Reserved for NWSNUT.

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:


NULL
prev
next
prev
next
prev
next
prev
next
NULL
LISTPTR
Structure
head
tail

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

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 (page 231)**.

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 (page 302)**.

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

Fields in this structure should not be changed directly. Use NWSNUT functions for building and manipulating menus.

headernum

Specifies the message identifier for the menu header (set by **NWSMenu (page 257)**).

centerLine

Specifies the screen line the menu is centered on (set by **NWSMenu (page 257)**).

centerCoumn

Specifies the screen column the menu is centered on (set by **NWSMenu (page 257)**).

maxoptlen

action

Specifies the routine to be called when an option is selected (set by **NWSMenu (page 257)**).

arg1

Specifies NWSNUT internal; cannot be set.

arg2

Specifies NWSNUT internal; cannot be set.

arg3

Specifies NWSNUT internal; cannot be set.

arg4

Specifies NWSNUT internal; cannot be set.

arg5

Specifies NWSNUT internal; cannot be set.

arg6

Specifies NWSNUT internal; cannot be set.

menuhead

Specifies the list pointer structure for the menu list; cannot be set.

nutInfo

Points to 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

Points to NWSNUT internal; cannot be set

currentPortal

Specifies NWSNUT internal; cannot be set

headerHeight

Specifies NWSNUT internal; cannot be set

waitFlag

Specifies NWSNUT internal; cannot be set

listStack

Points to NWSNUT internal; cannot be set

saveStack

Points to a stack of lists saved with **NWSSaveList (page 289)**. To restore a saved list, call **NWSRestoreList (page 281)**.

nextAvailList

Specifies NWSNUT internal; cannot be set

head

Points to NWSNUT internal; cannot be set

tail

Points to 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 (page 328)**). This function can be obtained by calling **NWSGetDefaultCompare (page 199)**, and set by calling **NWSSetDefaultCompare (page 300)**.

freeProcedure

Points to NWSNUT internal; cannot be set

interruptTable

Points to the interrupt procedures enabled by **NWSEnableInterruptKey (page 183)**. A list of enabled interrupt keys can be saved by calling **NWSSaveInterruptList (page 288)**. A list of interrupt keys can be activated by calling **NWSEnableInterruptList (page 185)**.

functionKeyStatus

Specifies whether function keys are enabled. The following functions control function key status:

NWSDisableAllFunctionKeys (page 132)
NWSDisableFunctionKey (page 134)
NWSEnableAllFunctionKeys (page 180)
NWSEnableFunctionKey (page 181)
NWSEnableFunctionKeyList (page 182)
NWSSaveFunctionKeyList (page 287)

messages

Specifies the MessageInfo structure that holds message information for your NLM. **NWSInitializeNut (page 227)** 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 (page 302)**.

helpContextStack

Specifies this saves help context for your NLM. To save a context to the stack, call **NWSPushHelpContext (page 274)** ; to remove a context from the stack, call **NWSPopHelpContext (page 263)**. The last help context pushed onto the help stack is displayed when F1 is pressed.

currentPreHelpMessage

Specifies the identifier set by **NWSDisplayPreHelp (page 152)**.

freeHelpSlot

Specifies NWSNUT internal; cannot be set

redisplayFormFlag

Specifies NWSNUT internal; cannot be set

preHelpPortal

Specifies NWSNUT internal; cannot be set

helpActive

Specifies NWSNUT internal; cannot be set

errorDisplayActive

Specifies NWSNUT internal; cannot be set

helpPortal

Specifies NWSNUT internal; cannot be set

waitPortal

Specifies NWSNUT internal; cannot be set

errorPortal

Specifies NWSNUT internal; cannot be set

resourceTag

Points to set by **NWSInitializeNut (page 227)** to the value of the *resourceTag* parameter.

screenID

Points to NWSNUT internal; cannot be set

helpScreens

Points to the help file for your NLM. It is set by **NWSInitializeNut (page 227)** 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

Specifies NWSNUT internal; cannot be set

helpHelp

Specifies NWSNUT internal; cannot be set

allocChain

Points to NWSNUT internal; cannot be set

version

Specifies this is set by **NWSInitializeNut (page 227)** to CURRENT_NUT_VERSION, as defined in NWSNUT.H.

MyNutInfoStuff

Specifies NWSNUT internal; cannot be set

moduleHandle

Specifies NWSNUT internal; cannot be set

customData

Points to custom data to be passed within the NUTInfo structure. To obtain this field, call **NWSGetHandleCustomData (page 202)** ; to set it, call **NWSSetHandleCustomData (page 310)**.

customDataRelease

Points to the function used to release memory allocated to *customData*. To obtain this field, call **NWSGetHandleCustomData (page 202)** ; to set it, call **NWSSetHandleCustomData (page 310)**.

displayErrorLabel

Specifies whether **NWSDisplayErrorCondition (page 137)** and **NWSDisplayErrorText (page 140)** displays NLM version information. The default value for this field is 1, causing version information to be displayed. To hide version information, call **NWSSetErrorLabelDisplayFlag (page 304)**. 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

Fields in this structure should not be changed directly. Use NWSNUT functions for creating and manipulating portals.

frameLine

Specifies the top-most line of the portal. Set when the portal is created.

frameColumn

Specifies the left-most column of the portal. Set when the portal is created.

frameHeight

Specifies the frame height in lines. Set when the portal is created.

frameWidth

Specifies the frame width in columns. Set when the portal is created.

virtualHeight

Specifies the virtual height of the portal in lines. Set by calling **NWSCreatePortal (page 117)**.

virtualWidth

Specifies the virtual width of the portal in columns. Set by calling **NWSCreatePortal (page 117)**.

cursorState

Specifies to set when the cursor is enabled (see **NWSDisablePortalCursor (page 136)** and **NWSEnablePortalCursor (page 187)**).

borderType

Specifies the type of the portal border. Set by calling **NWSCreatePortal (page 117)**.

borderAttribute

Specifies the attribute for the portal border. Set by calling **NWSCreatePortal (page 117)**.

saveFlag

Specifies 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 (page 117)**.

directFlag

Specifies 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 (page 117)**.

headerAttribute

Specifies the screen attribute for the header. Set by calling **NWSCreatePortal (page 117)**.

portalLine

Specifies the top-most line of the physical display area within the frame (the area that can be written to).

portalColumn

Specifies the left-most column of the physical display area within the frame (the area that can be written to).

portalHeight

Specifies the number of lines in the physical display area within the frame (the area that can be written to).

portalWidth

Specifies the number of columns in the physical display area within the frame (the area that can be written to).

virtualLine

Specifies the top-most line of the virtual portal. Set by calling **NWSCreatePortal (page 117)**.

virtualColumn

Specifies the left-most column of the virtual portal. Set by calling **NWSCreatePortal (page 117)**.

cursorLine

Specifies the line that the cursor is on. The cursor position can be specified by calling **NWSPositionPortalCursor (page 270)**.

cursorColumn

Specifies the column that the cursor is on. The cursor position can be specified by calling **NWSPositionPortalCursor (page 270)**.

firstUpdateFlag

Specifies NWSNUT internal. Cannot be set

headerText

Points to the text for the header. Set by calling **NWSCreatePortal (page 117)**.

headerText2

Points to NWSNUT internal. Cannot be set

virtualScreen

Points to 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

Points to NWSNUT internal. Cannot be set.

nutInfo

Points to NWSNUT context information.

sequenceNumber

Specifies NWSNUT internal. Cannot be set.

markForReposition

Specifies NWSNUT internal. Cannot be set.

DBCSAction

Specifies NWSNUT internal. Cannot be set.

borderPalette

Specifies the palette used for the border. Set by calling **NWSCreatePortal (page 117)**.

showScrollBars

Specifies NWSNUT internal. Cannot be set.

lastLine

Specifies NWSNUT internal. Cannot be set.

longestLineLen

Specifies NWSNUT internal. Cannot be set.

verticalScroll

Specifies NWSNUT internal. Cannot be set.

horizontalScroll

Specifies NWSNUT internal. Cannot be set.

oldVertical

Specifies NWSNUT internal. Cannot be set.

oldHorizontal

Specifies 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

Specifies the completion code for the error.

errorMessageNumber

Specifies the message number for the error.

Remarks

This structure is used by **NWSDisplayErrorCondition (page 137)**.

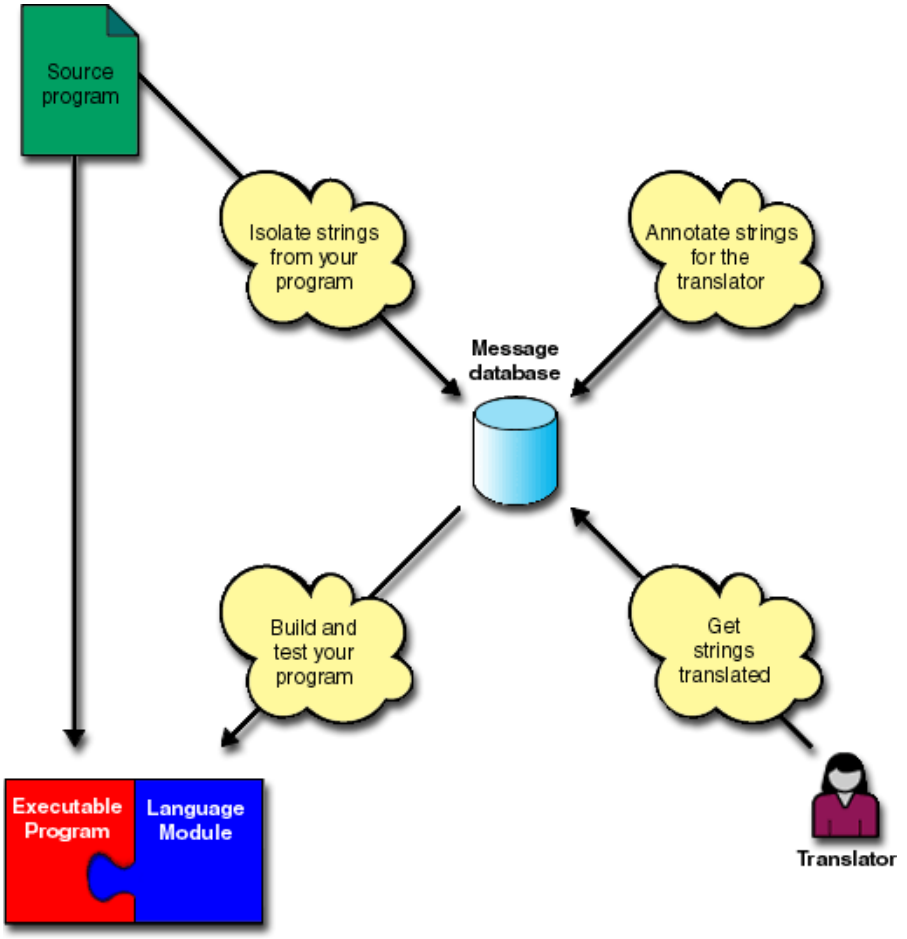
5 NLM Internationalization Concepts

This documentation describes Internationalization NLMs, its functions, and features.

Internationalization is the process of preparing programs for use in more than one language. The core feature of internationalized programs is separation of message and help files from the rest of the code. If designed properly, internationalized programs can be ported easily from one language and locale to another. When the new language is specified, text and related symbols (such as monetary symbols) appear in the language and locale of a user in the specified language area. However, the code providing essential program functionality executes in the same way independent of the language specified.

If you are internationalizing GUI applications (Windows 95, Windows NT, and so forth) to run on NetWare® clients, please refer to the internationalization instructions and tools provided with the relevant platform SDK.

The NetWare message internationalization tools are a set of DOS command-line utilities designed to help you prepare text-based software compatible with the NetWare OS for localization into other languages. The following diagram illustrates the process supported by the tools:



Internationalization Checklist

- User interface resources, such as text strings and help files, are isolated into separate modules. The modules are dynamically selected and bound with the program, based on the user's language. (See “Isolating the User Interface” on page 389.)
- The user interface is localizable. Text strings don't have problematic wording, hard coded linguistic features, or characters that might not be viewable or understandable to the translator. Icons are not linguistically

based. Where there might be ambiguities for the translator, annotations have been prepared. (See “Making the User Interface Localizable” on page 392.)

- Variations in localized text, such the length of text strings and the order of insertion parameters, are dynamically detected and handled by the program. (See “Handling Variations in Localized Text” on page 399.)
- Text formatting conventions, such as sort order, case conversion rules, and time and number notations, are dynamically selected and applied by the program, based on the user's locale information. (See “Formatting Text in a Locale Sensitive Way” on page 401.)
- Double-byte characters in text are dynamically detected and handled by the program, based on the user's character set. This applies to all character input, processing, and output operations. Unicode conversions are made when interfacing with software that uses that standard. (See “Handling Double-Byte Characters” on page 402.)
- Direct interactions with hardware are isolated from the program. Hardware-specific resources like addresses for timers, interrupt controllers, ROM routines, and video memory are isolated into separate modules that are selected and bound with the program based on the user's machine type. (See “Isolating the Hardware Interface” on page 405.)

File Name Changes for MSGLIB

In order for **MSGLIB** (and the **MSGEDIT** tool) to run successfully, certain files must be copied from the \NWSKD\TOOLS directory into the search path, and the names of the copied files must be changed. The following table lists these files and specifies the required name change:

Current Name	New Name
SYS_ERR.DAT	SYS$ERR.DAT
SYS_HELP.DAT	SYS$HELP.DAT
SYS_MGR.DAT	SYS$MSG.DAT
IBM_RUN.DAT	IBM$RUN.OVL
_RUN.OVL	$RUN.OVL

Format Specifiers Supported in Localization

Format specifiers that don't conform to the following definition are not supported in the NetWare® internationalization process as listed below:

String consisting of a % sign followed by	
Any (or any combination) of the following characters: space, +, -, #, 0	
An optional field width, which is a sequence of digits or an asterisk	
An optional precision, which is a period followed by optional digits or an asterisk	
An optional letter (see below)	
A conversion character: d,i,o,u,x,X,f,e,E,g,G,c,s,S,p,n	
Optional letters:	h can precede d,i,o,u,x,X,n
	l can precede d,i,o,u,x,X,n
	L can precede e,E,f,g,G

Examples of unsupported format specifiers include %Fs and %q.

Language IDs

The following table lists the identifiers for the various languages in the NetWare® server and client environments.

Language	ID on Server	ID on Client
French (Canada)	0	FRANCAIS.CAN
Chinese (Simplified)	1	CHINESES
Danish	2	DANSK
Dutch	3	NEDERLAN
English	4	ENGLISH
Finnish	5	SUOMI

Language	ID on Server	ID on Client
French (France)	6	FRANCAIS
German	7	DEUTSCH
Italian	8	ITALIANO
Japanese	9	NIHONGO
Korean	10	HANGUL
Norwegian	11	NORSK
Portuguese (Brazil)	12	PORTUGUE
Russian	13	RUSSKI
Spanish (Latin America)	14	ESPANOL
Swedish	15	SVENSKA
Chinese (Traditional)	16	CHINESET
Polish	17	
Portuguese (Portugal)	18	
Spanish (Spain)	19	

Locale-Sensitive Routines for Text Formatting

The NetWare® OS provides locale-specific text-formatting information (in the LCONFIG.SYS file) for use by NLM applications. The NetWare internationalization functions for character collation, comparison, and numeric formatting make use of this information to achieve locale sensitivity. This is also true of the NWSNUT default character-compare and date/time services. Operations using these functions should behave properly with respect to locale sensitivity.

Note that the LCONFIG.SYS information for Chinese, Japanese, and Korean locales currently doesn't provide a sort order for double-byte characters. However, unless it is critical that your program sort double-byte characters, you should still use the functions mentioned in "Using Functions that Support Parameter Reordering" on page 400.

Message Database Flags and Fields

The following table lists the flags and fields associated with message database strings. If the "MSGEDIT" column is marked, the flag or field is editable using **MSGEDIT**.

Flag/Field	Specifies	MSGEDIT
Doc status	Whether the string passed the documentation (user interface) review. A value of "None" means that no indication has been made using **MSGDOC**.	
Enabling unfriendly	Whether the string has any of the localizability problems documented in ""Making the User Interface Localizable" on page 392 " section of the "Internationalizing DOS Utilities" chapter in this book.	X
Length limit	Length limit, in bytes, entered by the engineer.	X
Length preceded	Whether the string is length preceded.	X
Message number	String's index number in the message database.	
Message type	Type of string: S = C or assembly string M = Message Librarian string C = C character constant	
Notes	Notes entered by engineers, translators, and user interface reviewers.	X
Number of parameters	Number of insertion parameters in the string.	
Original length	Length of the untranslated string.	
Original text	Text of the untranslated string.	
Parms	Insertion parameter descriptions. If you enter a description here and then change the number of parameters in the source string, the next time your run **MSGEXTR** or **MSGIMP** a tilde (~) is inserted at the start of this field. Delete the tilde when you update the parameter description.	X
Should be translated	Whether the string should be translated.	X

Flag/Field	Specifies	MSGEDIT
Source	Source file and line number the string came from.	
Source verified	Whether the string has been inspected for the localizability problems documented in “Making the User Interface Localizable” on page 392.	X
Translated text	Text of the translated string.	
Void translation	(Unused field)	

Message Librarian (MSGLIB.EXE)

MSGLIB is a menu utility that lets you create and edit message libraries for your NWSNUT-based NLM programs. Message libraries are special source code libraries into which you isolate user interface text strings for localization purposes.

A message library consists of two files: an .MLC file that contains the literal user interface text strings, and an .MLH file that defines symbolic names for the literal strings.

In your program, you reference the strings by their symbolic names. As for the literal strings, you import them into a message database (.MDB file), where they can be translated and exported as a separate loadable message module (.MSG file).

String Expansion Factors

The following table lists the expansion factors currently used by the NetWare® message internationalization tools for text strings that will be translated.

NOTE: While these are the expansion factors that are currently "enforced" by the tools, they don't provide for enough expansion. You should design your program using the expansion factors described in “Handling Variations in Localized Text” on page 399.

Characters in English String	Expansion Factor
1-10	3.0
11-20	2.0
21-30	1.8

Characters in English String	Expansion Factor
31-50	1.6
51-70	1.4
75+	1.3

Transfer File Keywords

The following table lists the keywords used in transfer files that are exported from or imported to message databases using MSGXFER (page 449). The keywords are case insensitive. Each should be on a line by itself. The first five keywords comprise the transfer file header; the rest are used to describe individual strings and their translations.

Keyword	On Export, Specifies	On Import, Specifies
$PROGRAM	Name of the program the transfer file is for, as read from the message database label.	(Same as on export—must match the program name in the message database label, or the transfer file won't import.)
$VERSION	Version number of the program the transfer file is for, as read from the message database label. If the database label doesn't specify a program version number, this line isn't inserted in the transfer file.	(Same as on export—must match the program version number in the message database label, or the transfer file won't import.)
$DATE	Transfer file creation date.	(Same as on export, unless changed by the translator.)
$LANGUAGE	(Not present on export.)	Language of the translation. MSGXFER (page 449) issues a warning if this keyword is missing, but it is otherwise ignored.
$COMMENT	(Not present on export.)	Number of lines of comment text that follow, if any. MSGXFER (page 449) displays the comment text during the import process.
$MESSAGE	Index number for the message database string.	(Same as on export—the translator shouldn't change it.)

Keyword	On Export, Specifies	On Import, Specifies
$TEXT	Number of text lines in the message database string that follows. The line breaks in the message string correspond to those hard coded in the source code by the programmer.	(Same as on export— the translator shouldn't change it.)
$TRAN	(Not present on export.)	Number of text lines in the translation that follows. MSGXFER (page 449) preserves the line breaks used by the translator. If the number of insertion parameters in the translation differs from that of the original string, MSGXFER (page 449) won't import the translation. MSGXFER (page 449) issues a warning if this keyword is missing, or if the line count of the translation differs from that of the original string.
$PARMS	Number of text lines in the insertion-parameter descriptions that follow (if any), as read from the message database "Parms" field.	(Same as on export—the translator shouldn't change it.)
$NOTES	Number of text lines of notes to the translator that follow (if any), as read from the message database "Notes" field.	(Same as on export—the translator shouldn't change it.)
$PROBLEM	(Not present on export.)	Number of text lines of problem annotation (made by the translator) that follow, if any. MSGXFER (page 449) adds the problem annotation to the string's "Notes" field in the message database.
$LENGTH	Length limit for the translation, as read from the message database "Length limit" field.	(Same as on export—the translator shouldn't change it.) MSGXFER (page 449) won't import the translation if it exceeds this limit. The hard returns in the translation don't count as part of the length.

Use of Single Byte Characters for NetWare Designations

By design decision, NetWare® doesn't support double-byte characters in server names, volume names, Y/N responses, or drive letters. Server names include names of NetWare servers, print servers, mail servers, and any other servers that advertise a service on the network. In these contexts, you can assume that all characters are single-byte.

Internationalization Tools

Most Commonly Used Internationalization Tools

MSGMAKE (page 439)	Creates an empty message database (.MDB file) to hold your program's user interface text strings
MSGEXTR (page 426) or MSGIMP (page 435)	Extracts the text strings from your source program and puts them in the message database. Use MSGIMP (page 435) if the strings are in Message Librarian files; otherwise, use MSGEXTR (page 426).
MSGTRAN (page 447)	Translates the strings in the message database into a fake language of your choice, which lets you do simulated localization testing.
MSGEXP (page 423)	Creates a binary message module (.MSG file) from the strings in the message database. Your program uses the message module in execution.
MSGEDIT (page 422)	Lets you annotate the strings in the message database for the translator
MSGCHECK (page 418)	Scans the message database for potential localization problems

Complete Summary of Internationalization Tools

The following table summarized all message internationalization tools in this SDK.

Tool	Description
MSGCHECK (page 418)	Scans message database strings for potential localization problems
MSGDUMP (page 420)	Displays or prints the contents of a message database
MSGEDIT (page 422)	Allows annotation of message database strings for the translator

Tool	Description
MSGEXP (page 423)	Creates an output file from the strings in a message database
MSGEXTR (page 426)	Copies quoted strings from source files into a message database
MSGMAKE (page 439)	Lists the number of strings in a message database, by source file
MSGFLAGS (page 430)	Sets or clears specified flags for all strings in a message database
MSGGLOSS (page 431)	Lists all the words used in a message database's strings
MSGGREP (page 433)	Finds occurrences of a string in a message database
MSGIMP (page 435)	Copies Message Librarian strings into a message database
MSGLABEL (page 437)	Labels a message database
MSGMAKE (page 439)	Creates and labels an empty message database (.MDB file)
MSGPFIX (page 441)	Fixes parameter count errors in a message database
MSGPURGE (page 442)	Purges a source file's strings from a message database
MSGRENAM (page 443)	Renames a source file in a message database
MSGSTATS (page 444)	Gets statistics on message database string characteristics
MSGTRAN (page 447)	Creates fake translations in a message database
MSGXFER (page 449)	Exports message database strings for translation

Summaries of the Tools

This table summarizes the tools you will most commonly use for internationalization.

MSGMAKE (page 439)	Creates an empty message database (.MDB file) to hold your program's user interface text strings
MSGEXTR (page 426) or MSGIMP (page 435)	Extracts the text strings from your source program and puts them in the message database. Use MSGIMP (page 435) if the strings are in Message Librarian files; otherwise, use MSGEXTR (page 426).
MSGTRAN (page 447)	Translates the strings in the message database into a fake language of your choice, which lets you do simulated localization testing.

MSGEXP (page 423)	Creates a binary message module (.MSG file) from the strings in the message database. Your program uses the message module in execution.
MSGEDIT (page 422)	Lets you annotate the strings in the message database for the translator
MSGCHECK (page 418)	Scans the message database for potential localization problems

The following table summarized all message internationalization tools in this SDK.

Tool	Description
MSGCHECK (page 418)	Scans message database strings for potential localization problems
MSGDUMP (page 420)	Displays or prints the contents of a message database
MSGEDIT (page 422)	Allows annotation of message database strings for the translator
MSGEXP (page 423)	Creates an output file from the strings in a message database
MSGEXTR (page 426)	Copies quoted strings from source files into a message database
MSGFILES (page 429)	Lists the number of strings in a message database, by source file
MSGFLAGS (page 430)	Sets or clears specified flags for all strings in a message database
MSGGLOSS (page 431)	Lists all the words used in a message database's strings
MSGGREP (page 433)	Finds occurrences of a string in a message database
MSGIMP (page 435)	Copies Message Librarian strings into a message database
MSGLABEL (page 437)	Labels a message database
MSGMAKE (page 439)	Creates and labels an empty message database (.MDB file)
MSGPFIX (page 441)	Fixes parameter count errors in a message database
MSGPURGE (page 442)	Purges a source file's strings from a message database
MSGRENAM (page 443)	Renames a source file in a message database
MSGSTATS (page 444)	Gets statistics on message database string characteristics
MSGTRAN (page 447)	Creates fake translations in a message database
MSGXFER (page 449)	Exports message database strings for translation

6 NLM Internationalization Tasks

This documentation describes common tasks associated with Internationalizing NLMs.

Isolating the User Interface

Because the user interface of the program may be translated into several languages, you will want to isolate it into a separate module that can be replaced at load time, based on the user's language.

1. Set up the internationalization tools.

   Put the internationalization tool in your search path somewhere and run BT.BAT (included with the tools). This loads Btrieve, which is required for the tools to work.

2. Create a message database.

   Run **MSGMAKE**. Note that the first parameter to this and all message tools is the name of the message database (without the .MDB extension). For example:

   ```
   msgmake myprog "My Program" 1 4 0
   ```

   <Enter>

   This creates an empty message database (MYPROG.MDB) to hold the user interface text strings for a single program ("My Program", Version 1, Language 4, Translation 0).

3. Put the user interface text strings in the message database.

   Make sure the empty message database is in the same directory as the source files that contain the user interface strings.

If the user interface strings are in Message Librarian (.MLC and .MLH) files, run **MSGIMP** on them. For example:

```
msgimp myprog mlc myprog.mlc
```

<Enter>

If the user interface strings are in .C files, run **MSGEXTR** on each C file. For example:

```
msgextr -mmyprog -w -smyprog.c
```

<Enter>

Make sure the empty message database is in the same directory as the source files that contain your user interface strings.

MSGIMP indexes the strings in the message database using the numbers defined in the message header file (MYPROG.MLH), but doesn't change the .MLC file. On the other hand, **MSGEXTR** writes some changes to the .C file. For example:

Change Inserted by MSGEXTR:

```
printf( MSG ("My Program, Version 1.0\n", 1));
printf(MSG ("This is printed on the screen.\n", 2));
```

Notice that a call to the "MSG" macro has been inserted around each quoted string. The number that appears as the second argument to the MSG macro represents the string's index number in the message database.

Either way, the quoted strings are copied from the source file (MYPROG.MLC or MYPROG.C) into the message database (MYPROG.MDB).

If you modify any strings in your source files after running **MSGIMP** or **MSGEXTR**, update the message database by rerunning **MSGIMP** or **MSGEXTR** on those files. As long as you use the same command as before, the previously extracted strings will be handled properly. We recommend that you always use the **-W** option with **MSGEXTR**.

4 Create the English message module.

From the directory containing the message database, run MSGEXP. For example:

```
msgexp myprog default cwsysmsg myprog.msg
```

<Enter>

This creates the binary message module (MYPROG.MSG) from the English strings in the message database (MYPROG.MDB). The **DEFAULT** and **CWSYSMSG** parameter options specify the type of strings in the message database and the format for the binary message module, respectively.

5 Create the English help module.

If your program uses the NetWare® NLM™ User Interface (NWSNUT), use the Help Librarian utility (HELPLIB.EXE) to create the binary English help module (for example, MYPROG.HLP).

6 Link the English message and help modules.

If the English message and help modules are the default user interface for your program, link them with the MESSAGES and HELP options in the linker definition file. Do not link language modules with your program other than the default, but load them separately if they are needed.

7 Code the program to access the appropriate message and help modules.

The NLM loader automatically loads and initializes the message and help modules that correspond to the user's language (specified by the **LANGUAGE** command).

7a If your program uses NWSNUT, access the individual resources in the message and help modules using standard NWSNUT functions, such as

NWSGetMessage
NWSPushHelpContext
NWSPopHelpContext

7b If your program doesn't use NWSNUT, define the **MSG** macro to be an index into an array of messages. Initialize the message array during startup by calling **LoadLanguageMessageTable**. For example:

```
#define MSG(s, id)     myprogMesgTable[id]
...

/* during program initialization... */

BYTE    **myprogMesgTable;

   LoadLanguagMessageTable(&myprogMesgTable, &msgCnt,
     &languageID);
```

```
/* Later, during processing, we detect an error and
   so print a message ... */

NWprintf(MSG("MYPROG: Insufficient memory.\r\n", 132));
```

Making the User Interface Localizable

Ensure that the user interface of the program does not have characteristics that hinder the localization process.

1. Eliminate "problem" characters from the text.

   Ensure that the user interface of your program does not have characteristics that hinder the localization process.

   The following types of characters can cause problems during localization. Eliminate them from your user interface text (with the exceptions noted):

Character Type	Problem
Control characters	Bell, carriage return, and line-feed are permissible. However, other control characters might display differently to the translator. Even if they display correctly, the translator might not know what to do with them.
Line-draw characters	Line-draw characters might appear as different characters in another code page—do not hard code them into the user interface.
Characters above 0x07E	Characters in this range vary across code pages, and thus might appear differently to the translator. Even if they display correctly, the translator might not know what to do with them.
Format specifiers	The NetWare® message internationalization tools convert format specifiers to reorderable tokens (such as <1>, <2>, and so forth) for the translator.

2. Use programming techniques that are not specific to one language.

   For example,

   ```
   printf("%d user%c", count, count != 1 ? 's' : '');
   ```

   depends on making a word plural by adding an "s"—many languages form plurals in other ways. Instead, use a separate call to display each variation of the text string, as in the following:

   ```
   if (count == 1)
       printf("%d user", count);
   ```

```
else
    printf("%d users", count);
```

3. Avoid using fragmented text strings.

   Don't construct text strings from fragments.

   Text strings constructed at run time from smaller string fragments cause problems for localization. For example, in the following code

```
printf("Invalid character found in file %s, ", fileName);
printf("line %d.\n", lineNum);
```

   The translator sees each **printf** as a separate string, and might not know that they go together. Even if the translator figures out that the fragments go together, he or she might not be able to translate the message correctly because of differences in word order from one language to another.

   To allow the translator to reorder words and insertion parameters as desired, make sure you display each message in its entirety, using a single call. For example:

```
NWprintf("Invalid character found in file %s, line %d.\n",
 fileName, lineNum);
```

4. Use clear an consistent wording.

   Resolve wording problems of the types listed below:

Type	Examples	Problem	Solution
Culture-specific phrases	"John Doe"	No translations for such phrases	Use other descriptive words
Noun clusters	"Volume mount problem list overflow"	Several possible interpretations	Add prepositions and articles to clarify intent
Abbreviations	"int"	Is it "integer" or "interrupt"?	Spell out words whenever possible
Acronyms	"ECB"	Several possible interpretations	Make a note to the translator (see below)
Technical jargon	"skulk" "promiscuous"	The specialized meaning can elude the translator	Make a note to the translator (see below)

Type	Examples	Problem	Solution
Inconsistent terminology	"Not enough memory" "Insufficient RAM"	The translator won't know whether the terms are used synonymously or to make a distinction	Use only one word or phrase for a given concept

Annotating NLM Programs

You should provide several kinds of annotations in your message database. If you have not created a message database, see "Isolating the User Interface" on page 389.) To make the annotations, follow this procedure:

1. Run **MSGEDIT** from the directory containing your message database.

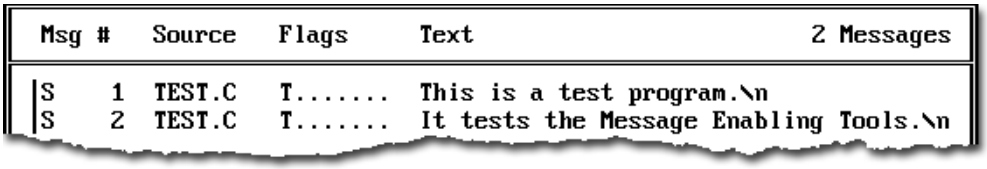
   Type

   ```
   msgedit
   ```

   and press <Enter>. This command displays a menu of messaged databases in the current directory, as illustrated in the following screen shot:

2. To display list of strings in your message database, from the options menu choose "Examine Message Database" as illustrated in the following screen shot:

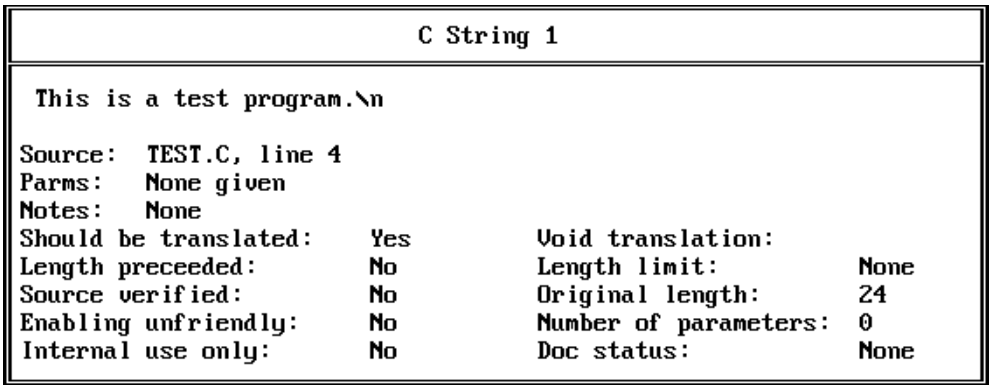
A list of the strings in your message database appears as in the following screen shot:



3. Make annotations as explained in the following topics:
   - "Verifying Translation Readiness" on page 395
   - "Identifying Text That Should Remain in English" on page 396
   - "Identifying Length Limits" on page 397
   - "Describing Insertion Parameters" on page 397
   - "Annotating Context" on page 398
   - "Clarifying Ambiguous Words" on page 399

Verifying Translation Readiness

1. Bring up the strings in your database, as explained in "Annotating NLM Programs" on page 394, and check each string to make sure it meets the requirements in "Making the User Interface Localizable" on page 392.
2. For every string that meets the above criteria, highlight the string with the F5 key, press <Enter>, and when the following screen shot appears, set "Source Verified" to Yes.



3. For every string that fails to meet the criteria in Step 1, highlight and choose the string as explained in Step 2, and set "Enabling unfriendly" to Yes.

4. Before sending your program to a translator, make every string is accurately set to "Source Verified."

Identifying Text That Should Remain in English

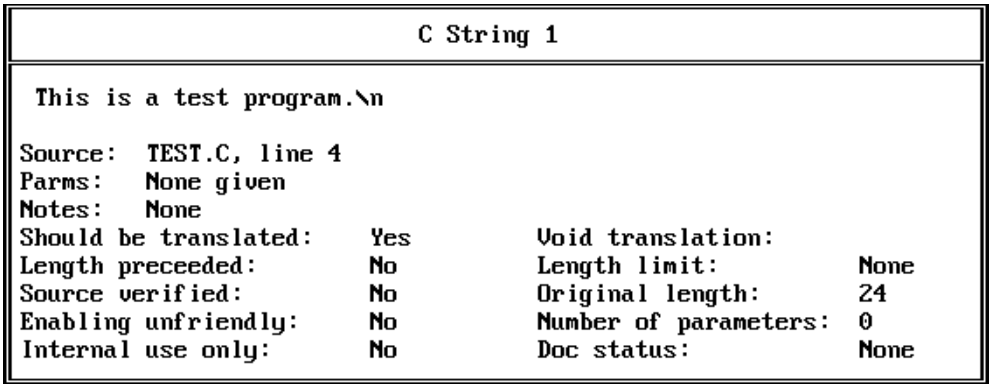
1. Decide which text should remain in English, based on the following table:

Text type	Reason for leaving in English
Debugging messages	The user should never see these.
Abend messages	Technical support groups in various countries have requested that these not be translated.
Program keywords	These are words that your program or other programs parse for, such as command names and parameters, configuration parameters, the username GUEST, the group name EVERYONE, and so forth.
Product names	Most product names, such as NetWare®, should remain in English.
Function names	In general, putting function names in user interface text isn't very user friendly. But if for some reason you need to do it, you should typically identify them as text that should remain in English.

2. Using the marking method explained in "Verifying Translation Readiness" on page 395, mark strings to be completely or partially left in English as explained in the following steps.

3. If a string should remain entirely in English, set the "Should be translated" field of the "Multiple Messages Flags" screen to No.

4. If a string should be partially translated, choose it specifically from the string list. A screen will appear as illustrated in following screen shot:



5 In the "Notes" field, explain which words should remain in English, but leave the "Should be translated" field set to Yes.

Identifying Length Limits

1 If a string is less than 40 characters, specify your program's length limit for the string in the "Length Limit" field of the screen outlined in the final step of “Identifying Text That Should Remain in English” on page 396. Otherwise, the tools will default to the length limits explained in “String Expansion Factors” on page 383.

These expansion limits are confining, and might force a poor translation. For more information, see “Handling Variations in Localized Text” on page 399.

2 If a string is over 40 characters but your program doesn't allow it to expand to the limit discussed above, document the length limit imposed by your program.

Describing Insertion Parameters

The translators see string insertion parameters (format specifiers) as numbered tokens. For example:

```
<1> of <2> sectors bad on <3>.
```

1 For each string that have an insertion parameter, enter a description in the "Parms" field.

If you don’t enter a description, the translator won't know what the insertion parameters stand for.

The descriptions should use the numbered tokens that the translator will see, even though you don't see them when viewing the string in **MSGEDIT**. For example:

```
<1> = number of bad sectors <2> = total number of sectors
<3> = volume name
```

Annotating Context

Strings often have different translations in different contexts. For example, "Restore menu" in German can be either "Menü Wiederherstellen" or "Wiederherstellungsmenü", depending on whether it is a command or the title of a menu.

1. Indicate the context of each string in the "Notes" field (see the screen above).

   When translating, the translator typically doesn't view the strings in the context of the running program.

For long messages whose context is obvious, annotating the context might not be necessary. But it is especially critical for shorter strings that could be interpreted in several ways.

In addition, because the translators read annotations written by many different engineers, we recommend that all engineers use the following terms for the various NWSNUT user interface contexts.

Term	Meaning
screen header	The top two or three lines on the screen that give the program name, version, current date, and so on.
screen footer	The bottom line on the screen that shows the available editing and function keys.
context-sensitive help string	The string of help text that appears above the screen footer. This text changes as you scroll through the different parts of the screen.
menu title	The text in the top section of a menu.
menu item	A line of text in the lower section of a menu.
list title	The text in the top section of a list box. This text is sometimes divided into more than one column.

Term	Meaning
list item	An item in a list box.
form title	The text in the top section of a form.
field name	The text that labels a field in a form. A field name is sometimes called a prompt.
status message	An asynchronous message that provides information to the user.
error message	An asynchronous message that reports an error to the user.

Clarifying Ambiguous Words

1 Define special technical jargon and acronyms for the translator. Do this in the "Notes" field. For example:

```
Promiscuous is a mode of operation in which a LAN card
picks
up all the packets that pass by on the wire, rather than
just
those addressed to the LAN card.
```

Handling Variations in Localized Text

1 Design your applications to the expansion factors shown in the above table, regardless of whether or not they are enforced by the tools you are using.

The NetWare® message internationalization tools currently use the expansion factors documented in “String Expansion Factors” on page 383, which don't allow for enough expansion. The program must be able to accommodate variations in the length and insertion-parameter order of localized text strings.

2 Use the engineering guidelines described in the following subtasks:

- “Designing Displays to Accommodate Text Expansion” on page 400
- “Designing Screen Layouts to Accommodate Text Expansion” on page 400
- “Using Functions that Support Parameter Reordering” on page 400

Designing Displays to Accommodate Text Expansion

For each instance in which your program retrieves and displays a user interface text string, ensure that there is enough room for the string in the memory buffer and on the screen. Allocate the needed room at run time based on the localized string length, or build in enough extra room from the start.

To estimate the amount of room that a given string will need when localized into the various languages, multiply its English length by the expansion factor specified in the following table. These expansion factors account for 95% of the cases. The remaining 5% of the cases required more expansion room.

Characters in English string	Expansion Factor
1-5	3.1
6-25	2.2
26-40	1.9
41-70	1.7
75+	1.5

The NetWare® message internationalization tools currently use the expansion factors documented in “String Expansion Factors” on page 383, which don't allow for enough expansion. Design your applications to the expansion factors shown in the above table, regardless of whether they are enforced by the tools you are using.

Designing Screen Layouts to Accommodate Text Expansion

For each instance in which your program writes anything on the screen, determine the overall size, location, and layout based on expanded string lengths and the available screen space. You can make these determinations at run time based on current conditions, or account for them when designing your program.

Using Functions that Support Parameter Reordering

The standard C **printf** family of functions doesn't support parameter reordering.

1 To print a string that has multiple insertion parameters, use one of the following NetWare® internationalization functions, defined in NWLOCALE.H:

NWprintf
Syntax (page 324)
NWvprintf
NWvsprintf

In addition, the following NWSNUT functions also support parameter reordering:

NWSAlert
NWSAlertWithHelp
NWSDisplayErrorText
NWSDisplayErrorCondition
NWSTrace

Formatting Text in a Locale Sensitive Way

Your program must be able dynamically to adapt text formatting operations to the specific requirements of the user's locale (language and culture). Such operations include sorting, upper casing, lower casing, and the formatting of dates, times, numbers, and currency values for display.

1 Use the engineering guidelines described in the following information topics and subtasks:

"Locale-Sensitive Routines for Text Formatting" on page 381

"Designing Case-Conversion Routines to Skip over Double-Byte Characters" on page 401

"Eliminating Locale-Specific Programming Techniques" on page 402

Designing Case-Conversion Routines to Skip over Double-Byte Characters

By NetWare® design decision, double-byte characters should never be targeted for case conversion, even if the characters represent Roman (English) letters. Historically, most case-conversion routines don't check whether a given byte is part of a double-byte character.

1. Ensure that your case-conversion routines skip over double-byte characters. For information on how to detect double-byte characters, see “Handling Double-Byte Characters” on page 402.

Eliminating Locale-Specific Programming Techniques

1. In addition to using the locale-sensitive functions mentioned in “Using Functions that Support Parameter Reordering” on page 400, use locale-sensitive algorithms when evaluating characters without calling a library function.

   For example, when checking to see whether a character should be targeted for case conversion, check not only to see whether a letter lies between `a' and `z' or `A' and `Z'. Check also for letters with accents, diereses, or other diacritical marks. It is safe, however, to use single-quoted letters if they are being passed to NetWare® internationalization functions.

2. Do not use 0x7F to mask the high-order bit when you are evaluating characters.

   This practice is often based on the assumption that the evaluated characters are either lower ASCII or NetWare augmented wildcards (wildcards with the high bit set). Localized and user-entered strings can contain numerous characters whose high bit is set, including single-byte accented Roman letters, single-byte Japanese phonetic characters, and a vast number of double-byte (Asian) characters. Use high-bit masking only in contexts where the high bit is truly insignificant to your task.

Handling Double-Byte Characters

Chinese, Japanese, and Korean text contains a mixture of single-byte and double-byte characters.

1. Code your program so its text handling operations can detect the presence of double-byte characters when those languages are active. The following subtasks describe the engineering guidelines:

   - “Using "Double-Byte Aware" String-Handling Functions” on page 403
   - “Eliminating Single-Byte Specific Programming Techniques” on page 404
   - “Converting to and from Unicode as Needed” on page 405

Note that NetWare® server names, volume names, Y/N responses, and drive letters must be single byte letters as explained in "Use of Single Byte Characters for NetWare Designations" on page 386.

Using "Double-Byte Aware" String-Handling Functions

1. For character-level string operations like parsing, searching, comparing, wrapping, truncating, and so on, use routines that are sensitive to the presence of double-byte characters.

Such routines query the NetWare® operating system to determine whether a double-byte character set is currently being used. If it is, they determine the range of character codes that is reserved for the leading byte of double-byte characters, and use that information to detect double-byte characters in strings.

The NWSNUT input, display, and editing functions meet this requirement, although they require that you specify string lengths in bytes, not characters. The NetWare internationalization functions defined in NWLOCALE.H are also double-byte aware. Here's an example that illustrates the use of the **NWLstrcspn** and **NWSGetSortCharacter (page 221)**. This routine strips carriage return and line feed characters from a string.

Stripping Carriage Return and Line Feed Characters from a String

```
LONG DisplayStringCopy( /* routine to strip CR LF from string */
          void *sourceAddress,
          void *destinationAddress,
          LONG numberOfBytes)
{
  LONG indexCR, minIndex, len, offset, offset1, i;
  char copy[255];
  char charset[3] = { 13, 10, '\0' };

  CMovB( (BYTE *)sourceAddress, copy, numberOfBytes );
  copy[ numberOfBytes ] = 0;

  /* skip over all CR and LFs */
  len = numberOfBytes;
  offset = 0;
  offset1 = 0;
  i = 0;

  while ( i <  numberOfBytes ) {

    /* find first occurrence of CR or LF */
```

```
    indexCR = NWLstrcspn( (char *)copy+offset,
                        charset );

    if ( indexCR == (numberOfBytes - offset) ) {
      CMovB( (char *)sourceAddress+offset,
             (char *)destinationAddress + offset1,
             numberOfBytes - offset );
      i += numberOfBytes - offset;
       offset += numberOfBytes - offset;
      offset1 += numberOfBytes - offset;
    }
    else {
      minIndex = indexCR;

      /* copy everything up to CR or LF */

      CMovB( (char *)copy+offset,
             (char *)destinationAddress + offset1,
             minIndex );
      offset1 += minIndex;

      /* skip over CR or LF */
      if ( NWCharType( copy[indexCR] ) == NWDOUBLE_BYTE ) {
        minIndex+=2;
        len -= 2;
      }
      else {
        minIndex++;
        len-;
      }
      i += minIndex;
      offset  += minIndex;
    }
  }
  return( len );
}
```

Eliminating Single-Byte Specific Programming Techniques

Not only should you use the double-byte aware functions mentioned above, but you should also use double-byte aware algorithms when performing character-level string operations "manually" (without calling a library function).

1. When "manually" processing a string that might contain double-byte characters, always start by checking whether the first byte of the string

falls within the lead-byte range for double-byte characters. (Any other byte in the string could be the trailing byte of a double-byte character.)

2. After testing the first byte, advance one or two bytes (depending on the result) to get to the next character.

In this manner, you can safely proceed from character to character until you find the target character.

IMPORTANT: For the purposes of isolating double-byte bugs, it is better to use the functions discussed in "Using "Double-Byte Aware" String-Handling Functions" on page 403 rather than writing your own double-byte aware code.

Converting to and from Unicode as Needed

If your program interfaces with NDS™, you will have to supply and receive text in Unicode format.

1. Refer to the Unicode book for references to functions with which you can perform conversions between code pages and Unicode.

Isolating the Hardware Interface

If your program will be used on personal computers with different hardware architectures, you should isolate your program's hardware access operations.

1. Use NetWare® functions whenever possible.

   The easiest way to isolate hardware dependencies is to access the hardware indirectly through a generic software interface. For example, rather than directly updating video memory, call a NetWare function to do it. Hardware access operations done in this way are portable across all architectures supported by NetWare.

2. If direct hardware access is required, isolate it into a separate NLM™ application.

   If your program is a device driver, isolate the hardware-specific portion of your code into a separate NLM.

Dealing with Hardware Resource Constraints

1. Your program's dependencies on processor speed and available memory should be reasonable. Keep in mind that server machine configurations in some parts of the world might be slower and have less memory.

Testing Your Program

When you have completed the engineering procedures in this documentation, your program should be adaptable for use in each NetWare® market. Translation shouldn't require any reengineering or assistance from you. To verify that this is true, complete the testing procedure outlined in the following table. Detailed instructions are given after the table.

Step	Reason
Scan the source code	*Locale-sensitive formatting:*
	Dates, times
	Case conversion
	Double-byte character handling:
	Path delimiters
	Wildcards
Scan the message database	*Localizability:*
	Absence of problem characters
	Appropriate length limits
	Absence of wording/linguistic problems
	Annotations
Use fake message modules	*User-interface isolation:*
	Completeness of message module
	Opening of message module by language
	Localizability:
	Absence of string concatenations
	Localized text handling:
	String expansion allowance
	String insertion-parameter reordering
	Double-byte character handling:
	Text display, truncation, wrapping

Scanning the Source Code

1 Scan your source code for internationalization readiness problems. Some of these problems are listed in the table below.

Problem	Scan for	Explanation
Character Matching	memchr strchr strcspn strpbrk strrchr strstr strtok	These functions assume that each byte is a separate character. When matching characters in localized or user-entered strings, use the functions mentioned above in “Handling Double-Byte Characters” on page 402.
Monocasing	memicmp strcmpi stricmp strncmpi strnicmp strupr strlwr toupper tolower	These functions don't skip over double-byte characters; they also omit accented letters from processing. When converting case in localized or user-entered strings, use the functions mentioned above in “Formatting Text in a Locale Sensitive Way” on page 401.
Truncation	strncat strncpy strnset	These functions might split double-byte characters. When editing localized or user-entered strings, use the functions mentioned above in “Handling Double-Byte Characters” on page 402.
Date & Time	date time	Most standard C run-time date/time functions aren't locale sensitive. Use instead the functions mentioned above in “Formatting Text in a Locale Sensitive Way” on page 401.

Problem	Scan for	Explanation
Single Letters	`a'-`z' `A'-`Z'	If single-quoted letters are being used to determine whether a byte should be monocased, or if they are being compared to potential double-byte characters, you might have a problem. Make sure you use the techniques mentioned above in “Formatting Text in a Locale Sensitive Way” on page 401.
Backslash	`\\' \\\\ 5c	The backslash character code (0x5C) shows up as the second byte of several Japanese and Traditional Chinese double-byte characters. When searching localized or user-entered text for the backslash character, use the method described above in “Handling Double-Byte Characters” on page 402.
High Bit Masking	7f	If hexadecimal 7F is being used to mask bytes of localized or user-entered strings, you might have a problem. Make sure you follow the guideline mentioned above in “Formatting Text in a Locale Sensitive Way” on page 401.
printf	printf	The standard C **printf** family of functions doesn't allow string insertion parameters to be reordered. When displaying localized strings that have multiple insertion parameters, use the functions mentioned above in “Handling Double-Byte Characters” on page 402.

Scanning the Message Database

1. From the directory containing your message database, run **MSGCHECK**. For example:

   ```
   msgcheck myprog
   ```

   <Enter>

   This scans the message database (MYPROG.MDB) for strings that have various problems, such as unsupported characters, overly stringent length limits, undescribed insertion parameters, and known wording or programming problems (where you have set the "Enabling unfriendly" flag to Yes).

2. Optionally, also run **MSGSTATS** with the -N parameter:

```
msgstats myprog /m -N
```

<Enter>

This scans the message database (MYPROG.MDB) for strings with empty "Notes" fields. Unless a string is a long message whose user interface context is obvious, the "Notes" field should at least contain an indication as to the user interface context of the string.

Using Fake Message Modules

Using fake language modules allows you to test most language swapping capabilities without having hire a translator. One useful fake language is Pig-Latin, which has the following advantages:

- "Translation" into Pig-Latin is fairly easy
- Pig-Latin messages can be read by anyone who reads English
- Pig-Latin strings are considerably longer than the regular English originals, allowing you to test for length as well as for insertion of the correct translated string.

1. Create and scan a fake language database according to the instructions in the following tasks:
   - “Creating the Fake Message Modules” on page 409
   - “Scanning the Message Database” on page 408
2. Run the following tests on the fake language database:
   - “Testing for String Isolation” on page 410
   - “Testing for String Concatenation” on page 411

Creating the Fake Message Modules

1. In the server's SYS\SYSTEM\NLS directory, create numbered subdirectories to hold the English and the fake language modules. (Novell® uses 4 for the English module and 99 for European languages.)
2. From the directory containing your message database, run **MSGLABEL** to relabel the message database for the fake language. For example, if you set 88 as the subdirectory for Pig-Latin, the syntax would be as follows:

   ```
   msglabel myprog "My Program" 1.0 99 0
   ```

   <Enter>

This labels MYPROG.MDB as the message database for My Program, Version 1.0, Pig-Latin translation version 0.

3. Run MSGTRAN with the CONCAT option. For example:

   ```
   msgtran myprog concat
   ```

   <Enter>

   This generates a fake translation in the message database (MYPROG.MDB), consisting of the original strings surrounded by curly braces ({ }).

4. Run **MSGEXP** with the /T option:

   ```
   msgexp myprog default nlm nls\concat\myprog.msg /t
   ```

   <Enter>

   This creates a message module (MYPROG.MSG) from the translated messages in the message database (MYPROG.MDB), and puts the message module in the NLS\CONCAT directory. The DEFAULT and NLM parameters specify the type of strings in the message database and the format for the message module, respectively.

5. Repeat the above procedure to create message modules containing expanded strings, reordered strings, and wide Roman (Japanese double-byte) character strings, respectively. Use the options of the **MSGTRAN** tool, and put the message modules in the corresponding NLS subdirectories.

Testing for String Isolation

1. Set the server language to Pig-Latin by typing "LANGUAGE 88" at the system prompt (assuming you created 88 as the subdirectory for Pig-Latin).
2. Copy the message module from NLS\CONCAT to NLS\88, and load your program.
3. Verify that the correct message module (the one with curly braces { } around each string) has been opened.
4. Display as many user interface strings as possible.

   If you see strings that don't have curly braces, you know that they haven't been isolated into the message module. Unless the strings represent text that should remain in English, they should be isolated into the message module, as explained above in "Isolating the User Interface" on page 389.

Testing for String Concatenation

Do this test simultaneously with "Testing for String Isolation" on page 410.

1. Check for strings that have more than one set of curly braces—these strings have been constructed at run time from string fragments. This can be a problem for localization, as mentioned above in "Making the User Interface Localizable" on page 392.

Testing for String Expansion

1. Copy the message module from NLS\EXPAND to NLS\88 (assuming you created 88 as the subdirectory for Pig-Latin) and reload your program. Verify that the correct message module (the one with the expansion characters (....@) appended to each string) has been opened.
2. Display as many user interface strings as possible.

   If you see strings whose terminating `@' character has been truncated, you know that your program hasn't allowed enough expansion room for the strings. For information on designing for string expansion, see "Handling Variations in Localized Text" on page 399.

Testing for Parameter Reordering

1. Copy the message module from NLS\REVERSE to NLS\88 (assuming you created 88 as the subdirectory for Pig-Latin) and reload your program. Verify that the correct message module (the one with the strings in reversed word order) has been opened.
2. Display all the user interface strings that have multiple insertion parameters in a single text line, and verify that the program has reversed the parameters. If not, see "Handling Variations in Localized Text" on page 399.

Using MSGLIB

Starting MSGLIB

1. Copy the following files from the \NWSKD\TOOLS directory into the search path and rename the files. The following table lists the files and specifies the required name change:

Current Name	New Name
SYS_ERR.DAT	SYS$ERR.DAT
SYS_HELP.DAT	SYS$HELP.DAT
SYS_MGR.DAT	SYS$MSG.DAT
IBM_RUN.DAT	IBM$RUN.OVL
_RUN.OVL	$RUN.OVL

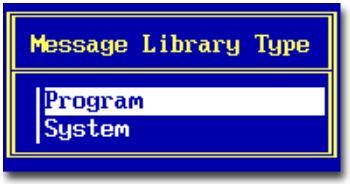
2 At the system prompt, type "MSGLIB" (without quotation marks) and press the <Enter> key. The main menu appears as in the screen shot below.

You are concerned only with the first, second, and last menu options — "system" message libraries do not apply to NLM programming. You can press the <F1> key for context sensitive help.

Creating a New Message Library

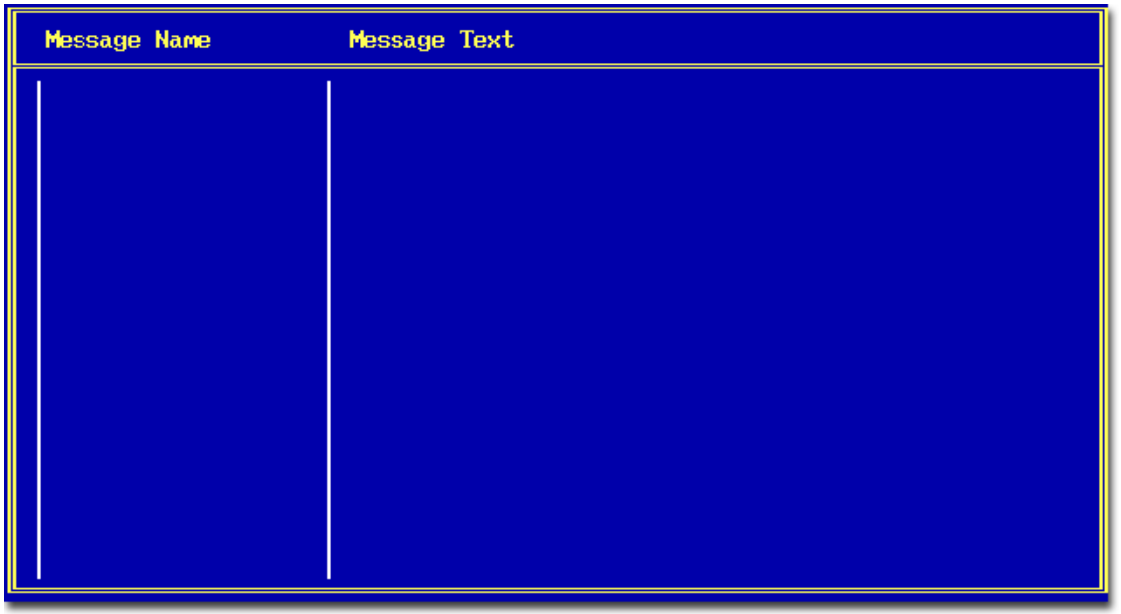
1 From the main MSGLIB menu, choose "Create New Message Library."

A menu appears as in the following screen shot:



2 Choose "Program."

A blank editing screen like the one below appears.



3 To enter a text string in the new message library, press <Insert>.

MSGLIB prompts you for the name of the new string.

4 Type the symbolic name you will use in your program for the string and press the <Enter> key.

MSGLIB limits you to 40 characters.

5 Type the literal text, including any format specifiers, and press <Enter>.

The string appears as an entry in the new message library.

6 Repeat the previous three steps to enter the desired strings. When you have entered all strings, press <Esc>.

MSGLIB prompts you whether to save the message library.

7 Choose "Yes" to save the message library, and enter the desired path and filename under which to save it. Do not specify a filename extension—**MSGLIB** automatically creates .MLH and .MLC files.

Editing an Existing Message Library

1. From the main MSGLIB menu, choose "Edit Existing Message Library."
2. From the menu that appears, choose "Program."

   MSGLIB prompts you for the name of the message library to open, as in the screen portion in the screen shot below.

3. Type the path and filename of the desired message library (without an extension), or press <Insert> to see a list of message libraries in the current directory. After typing or selecting the filename, press <Enter>.

   The message library appears in the editing screen.

4. To add a new string, use the <Insert> key as described in "Creating a New Message Library" on page 412. To delete one or more strings, select them and press <Delete>. To edit a string, select it and press <Enter>.

   Note that **MSGLIB** only lets you edit the literal text. To change the symbolic name, you must create a new string with the desired name, and delete the old one.

5. When you have finished editing, press <Esc> to exit. Choose "Yes" when prompted to save your changes. MSGLIB prompts you whether to use the existing filename. Press <Enter> to confirm the name, or enter a new filename under which to save the changes.

Printing a Message Library

1. From the main MSGLIB menu, choose "Print Existing Message Library."
2. From the menu that appears, choose "Program."

   MSGLIB prompts you for the name of the message library to print.

3. Type the path and filename of the desired message library (without an extension), or press <Enter> to see a list of message libraries in the current directory. After typing or selecting the filename, press <Enter>.

   MSGLIB prompts you for the name of the output file.

4 To print to a text file, enter the desired path and filename. To send the output to a printer, enter the desired device name (for example, LPT1).

7 NLM Internationalization Tools Reference

This documentation alphabetically lists the tools for Internationalizing NLMs and describes their purpose, syntax, and parameters.

MSGCHECK

Scans message database strings for potential localization problems

Syntax

MSGCHECK *messageDB*

Parameters

messageDB

Specifies the path and filename of the message database to scan. If you omit the extension, .MDB is assumed.

Remarks

MSGCHECK scans message database strings for potential localization problems, including the following:

- Characters that might not be viewable or understandable to the translator. This includes control characters (except for carriage return, line feed, and bell), line-draw characters, other characters above 0x7E, and obscure format specifiers (see the appendix for details). Only strings that will be translated are scanned for these characters.
- Flag or field settings that indicate the string isn't ready for localization. This includes any of the following: Source verified = No, Enabling unfriendly = Yes, Length limit value that is too short (based on the string expansion factors listed in the appendix), empty Parms field, Parms field entry that starts with a tilde (indicating that the number of parameters has changed since the entry was made), multiple Message type values exist in the database.

Replace *messageDB* with the path and filename of the message database to scan. If you omit the extension, .MDB is assumed.

Example

msgcheck myprog <Enter>

This scans MYPROG.MDB in the current directory. Here's an example of what the output looks like:

```
Messages with control characters (i.e. form feed, bell)
C0004
1 possible errors

Messages not flagged "Source verified"
C0006
1 possible errors

Messages flagged "Enabling unfriendly"
C0004 C0006
2 possible errors

Messages with possible wrong parameter descriptions
C0004
1 possible errors


Total messages 2, potential errors 5
```

MSGDUMP

Displays or prints the contents of a message database

Syntax

MSGDUMP *messageDB* [*sortBy*]

Parameters

messageDB

Specifies the path and filename of the message database to be displayed. If you omit the extension, .MDB is assumed.

sortBy

Optional. Specifies the basis for sorting the output:

-S	Sort the output by source file and message number.
-F	Sort the output by numeric value of message flags (not very useful).
-T	Sort the output by message text.
-M	Sort the output by message type.

Remarks

MSGDUMP displays or prints the contents of a message database.

Replace *messageDB* with the path and filename of the message database to be displayed. If you omit the extension, .MDB is assumed.

Example

msgdump myprog <Enter>

This prints MYPROG.MDB to the screen. Here's an example of what the output looks like:

```
C String:      1      TEST.C, line 4     Flags: T
Length limit:  None
Original text:
  This is a test program.\n
____________________
C String:      2     TEST.C, line  5     Flags: T
Length limit:  None
Original text:
  It tests the message internationalization tools.\n
__________________-
2 messages found
```

MSGEDIT

Allows annotation of message database strings for the translator

Syntax

MSGEDIT [*extension*]

Parameters

extension

Optional. Specifies the extension of the message databases to be opened.

Remarks

MSGEDIT annotates message database strings for the translator. See ""Making the User Interface Localizable" on page 392. "

Replace *extension* with the extension of the message databases to be opened. Don't type a period before the extension. If you omit the extension, MDB is assumed.

Example

msgedit <Enter>

MSGEXP

Creates an output file from the strings in a message database

Syntax

```
MSGEXP  messageDB  inputType  outputType  outputFile  [
options ]
```

Parameters

messageDB

Specifies the path and filename of the message database to export from. If you omit the extension, .MDB is assumed.

inputType

Specifies the source of strings to export:

CSTRING	Export only strings that came from C and assembly files.
MSGLIB	Export only strings that came from Message Librarian files.
DEFAULT	Export whichever string type the message database contains. (If both types are present, an error is printed.)

outputType

Specifies the format of file to create:

NLM	Create a binary message module for an NLM application.
CWSYSMSG	Create a binary message module for a DOS program.
MLH	Create a Message Librarian header file.
MLC	Create a Message Librarian source file.
INC	Create a Message Librarian assembly include file.

outputFile

Specifies the path and filename for the output. If you are creating a binary message module, by convention you should give it the same name as your executable program, but with a .MSG extension.

options

Specifies a variety of switches that are optional except with UNIX m4 files and programs using small memory models:

/T	Export the translated strings, not the original strings.
/I	Don't export strings that are flagged Internal use only.
/E	(Used only when creating a binary message module.) Support the issuing of an explicit error message when unused message slots are referenced; for example, <<<Bad message 235>>> instead of just <<<Bad message>>>, where 235 is the unused slot number.
/Z	Unload Btrieve when done.
dataSize	(Used to override the default data sizes for string insertion parameters when creating a binary message module.) Replace *dataSize* with one of the following: /3 Watcom 386 data sizes. This is the default for NLM message modules.
pointerType	(Used to override the default pointer type when creating a binary message module.) Replace *pointerType* with one of the following: /N Near pointers. This is required if your program uses a small memory model.

Remarks

From the strings in a message database, **MSGEXP** creates one of the following:

- Binary message module (.MSG file)
- Message Librarian file
- UNIX m4 file

When creating the output, **MSGEXP** converts the numbered tokens in translated strings back to format specifiers. If an error occurs in this process, run **MSGPFIX** on the message database.

Example

msgexp test cstring nlm test.msg /t <Enter>

This creates a binary NLM message module named TEST.MSG using the translated strings in TEST.MDB.

MSGEXTR

Copies quoted strings from source files into a message database

Syntax

```
MSGEXTR   -M  messageDB  [options ] [ -S  sourceFile ]
```

Parameters

messageDB

Specifies the path and filename of the target message database. If you omit the extension, .MDB is assumed.

options

Optional. Specifies a variety of file extraction options, usable more than once:

initType	Replace *initType* with one of the following:	
	-I	Create the specified message database. (If it already exists, an error message is issued.)
	-II	Reinitialize (clear out) the specified message database. If it doesn't exist, create it.
-T *msgMacro*	Use *msgMacro* instead of **MSG** as the message macro name.	
-W	Insert the message macros in the original source files. The **-N** and **-O** options override this option.	
-O *outputFile*	Insert the message macros in a copy of the source file, and save the copy to the path and filename specified by *outputFile*. This option affects the next source file only. If you omit this option and don't specify the **-W** or **-N** option, the modified source code is saved to NAME.MSG, where NAME is the original source filename.	
-N	Don't save the message macros anywhere. The **-O** and **-W** options override this option.	
-R	Don't extract single-quoted strings, such as `A'. If you omit this option, single-quoted strings are extracted but not tagged in the source code. They are extracted so you can track them for internationalization purposes.	
-D *string*	Don't extract instances of *string*. The string must not contain white space.	

-X	Extract pretagged strings only. The MSG macro must already be there.
-E	If an untagged string in the source program already exists in the message database, tag it as the existing string in the database. Extract and tag subsequent instances of the same string as new strings in the database. (Use this option if you prefer to keep the tagged version of your source program separate.)
-C	Perform source-file parsing in a way that allows nesting of C comments, as in the following example: /*This is a comment /*with first one*/ /*then another*/ nested subcomment.*/
-F	Set the Length preceded flag to No for each new string added to the message database. This option prevents **MSGEXTR** from checking whether strings are length preceded. (The option is provided because the checking algorithm can be "fooled" by strings whose first byte equals the string length minus one.)
-L *length*	Set the Length limit field to length for each new string added to the message database. If the string is longer than *length*, set the Length limit field to None.
-Z	Unload Btrieve when done.

sourceFile

Optional. Specifies the path and filename of the source files to be processed. See "Remarks."

Remarks

MSGEXTR copies quoted strings from source files into a message database. For information on how to use this tool, see “Isolating the User Interface” on page 389.

MSGEXTR also lets you create or reinitialize the message database before copying strings from the source files. However, typically you should use **MSGMAKE** to create the message database, as it lets you label the database at the same time. If you use **MSGEXTR** to create or initialize the database, a default program named "Unknown" with a language ID of zero is created.

You can repeat the pattern [*options*] **-S** *sourceFile* as often as desired. Options take effect in the order they appear on the command line—only source files that come after an option are affected by it. A subsequent option can override a previous one.

Replace *sourceFile* with the path and filename of the source file to be processed. To specify multiple source files, code multiple **-S** options in your command line. The processing performed on each source file depends on the options specified before it on the command line. You can omit the **-S** option when using the *initType* option.

Example

msgextr -mmyprog -w -smyprog.c <Enter>

This copies the quoted strings from MYPR0G.C into MYPROG.MDB, and inserts a call to the **MSG** macro around each string if one isn't already present.

MSGFILES

Lists the number of strings in a message database, by source file

Syntax

MSGFILES *messageDB*

Parameters

messageDB

Specifies the path and filename of the message database to be listed. If you omit the extension, .MDB is assumed.

Remarks

MSGFILES lists the number of strings in a message database, by source file.

Example

msgfiles myprog <Enter>

This lists the number of messages in MYPROG.MDB. For example:

```
MYPROG.C:       1 messages found
TEST.C:         2 messages found
————————————-
Grand Total:    3 messages found
```

MSGFLAGS

Sets or clears specified flags for all strings in a message database

Syntax

MSGFLAGS *messageDB* *flagSettings*

Parameters

messageDB

the path and filename of the message database to be processed. If you omit the extension, .MDB is assumed.

flagSettings

Specifies settings for various flags:

T	Should be translated
L	Length preceded
V	Source verified (can only be cleared)
U	Enabling unfriendly
I	Internal use only

Remarks

MSGFLAGS sets or clears specified flags for all strings in a message database.

For *flagSettings*, you must type a plus (+) or minus (-) sign before each flag to indicate whether to set or clear the flag.

Example

msgfiles myprog -t +i <Enter>

This sets the Should be translated flag to No and the Internal use only flag to Yes for each string in MYPROG.MDB.

MSGGLOSS

Lists all the words used in a message database's strings

Syntax

MSGGLOSS *messageDB* [*ignoreList*] [*options*]

Parameters

messageDB

Specifies the path and filename of the message database to be processed. If you omit the extension, .MDB is assumed.

ignoreList

Optional. Specifies the path and filename of a word list to be excluded from the output. This is a text file with white space between each word.

options

Optional. Specifies several options for word listing and cross referencing:

/A	Sort the word list alphabetically.
/F	Sort the word list by frequency of use.
/M	Suppress the message listing.
/X	Suppress message cross references.
/S	Suppress the message listing and cross references. (List only the words.)

Remarks

MSGGLOSS lists all the words used in a message database's strings. This can be useful to localizers for terminology management. You can specify words to be excluded from the list.

Example

msggloss myprog ignore.txt <Enter>

This lists the words used in the strings in MYPROG.MDB, excluding the words specified in IGNORE.TXT. For example:

```
Enabling  C0003

Message   C0003

My        C0001

Program   C0001 C0002

test      C0002

tests     C0003

Tools     C0003

Version   C0001

8 words in glossary
————————————-
C0001 My Program, Version 1.0
————————————-
C0002 This is a test program.
————————————-
C0003 It tests the message internationalization tools.
————————————-
Total of 3 messages
```

MSGGREP

Finds occurrences of a string in a message database

Syntax

```
MSGGREP  messageDB   searchString  [ searchType ]
```

Parameters

messageDB

Specifies the path and filename of the message database to be searched. If you omit the extension, .MDB is assumed.

searchString

Specifies the string you are searching for. If it contains white space, enclose it in double quotes.

searchType

Optional. Specifies various kinds of searches:

-O	Search only the original strings.
-T	Search only the translated strings.
-N	Search only the Notes fields.
-P	Search only the Parms fields.

Remarks

MSGGREP finds occurrences of a string in a message database. You can search the original strings, the translated strings, the Notes fields, or the Parms fields. The search is case insensitive.

Example

msggrep myprog program <Enter>

This finds and displays all occurrences of the word program in MYPROG.MDB. Here's an example of the output:

```
C String:      1      MYPROG.C, line 4       Flags: T
Length limit:  None
Original text:
  My Program, Version 1.0\n
____________________
C String:      2      TEST.C, line 4         Flags: T
Length limit:  None
Original text:
 This is a test program.\n
____________________
3 messages tested, 2 matched
```

MSGIMP

Copies Message Librarian strings into a message database

Syntax

```
MSGIMP  messageDB   fileType  inputFile
```

Parameters

messageDB

Specifies the path and filename of the message database to receive the strings. If you omit the extension, .MDB is assumed.

fileType

Specifies the type of input file:

MLC	The input file is an MLC file.
MLH	The input file is an MLH file.
SYS$MSG	The input file is a SYS$MSG.DAT or SYS$MSG.MLH file.

inputFile

Specifies the path and name of the input file.

Remarks

MSGIMP copies Message Librarian strings into a message database. Both the .MLC and .MLH file must be in the source directory, although you should reference only one of them on the command line (it doesn't matter which one).

The strings are indexed in the message database using the numbers defined in the .MLH file. The #define names for the strings are entered as notes in the database. Neither the .MLC nor .MLH file is modified.

Example

msgimp myprog mlc myprog.mlc <Enter>

or

msgimp myprog mlh myprog.mlh <Enter>

Either of the above copies the strings from MYPROG.MLC into the message database MYPROG.MDB, and indexes the strings in the database using the numbers defined in MYPROG.MLH.

MSGLABEL

Labels a message database

Syntax

```
MSGLABEL  messageDB  [ newLabel ]
```

Parameters

messageDB

Specifies the path and filename of the subject message database. If you omit the extension, .MDB is assumed.

newLabel

Optional. Specifies descriptive information about your program. If you use this parameter, **MSGLABEL** updates the message database label using the specified information. If you omit this parameter, **MSGLABEL** displays the database's current label.

progName	Name of the program the message database is for. The name is truncated to 12 characters. If it contains white space, enclose it in double quotes.
progVersion	Version number of the program. The number is truncated to 6 characters. If it contains white space, enclose it in double quotes.
tranLang	Language ID for the message database. This parameter is ignored for client programs (specify any value as a placeholder). See "Language IDs" on page 380.
tranVersion	Version number of the translation contained in the message database. The number is truncated to 6 characters. If it contains white space, enclose it in double quotes.

Remarks

MSGLABEL labels a message database. To *label* means to specify the program the message database is for, and the translation it contains. You can also use **MSGLABEL** to display a message database's label.

Example

msglabel myprog "My Program" 1 15 1 <Enter>

This labels MYPROG.MDB as the message database for My Program, Version 1, Swedish translation version 1. To display this label, run **MSGLABEL** again, omitting the *newLabel* parameter:

msglabel myprog <Enter>

Here's the output:

```
Program:      My Program
Version:      1
Language ID:  15
Translation:  1
```

MSGMAKE

Creates and labels an empty message database (.MDB file)

Syntax

MSGMAKE *messageDB* *label*

Parameters

messageDB

Specifies the path and filename of the message database to be created. If you omit the extension, .MDB is assumed.

label

Specifies descriptive information about your program:

progName	Name of the program the message database is for. The name is truncated to 12 characters. If it contains white space, enclose it in double quotes.
progVersion	Version number of the program. The number is truncated to 6 characters. If it contains white space, enclose it in double quotes.
tranLang	Language ID for the message database. This parameter is ignored for client programs (specify any value as a placeholder). See "Language IDs" on page 380.
tranVersion	Version number of the translation contained in the message database. The number is truncated to 6 characters. If it contains white space, enclose it in double quotes.

Remarks

MSGMAKE creates and labels an empty message database (.MDB file). To *label* means to specify the program the message database is for, and the translation it contains.

Example

msgmake myprog "My Program" 1 4 (None) <Enter>

This creates an empty message database MYPROG.MDB, and labels it as the database for My Program Version 1, English. **MSGMAKE** displays the following:

```
MYPROG.MDB created
Program:      My Program
Version:      1
Language ID:  4
Translation:  (None)
```

MSGPFIX

Fixes parameter count errors in a message database

Syntax

MSGPFIX *messageDB*

Parameters

messageDB

Specifies the path and filename of the message database to be fixed. If you omit the extension, .MDB is assumed.

Remarks

MSGPFIX fixes parameter count errors in a message database. During string extraction, the **MSGEXTR** and **MSGIMP** tools store in the message database a parameter count for each extracted string. In some cases, the parameter count can be wrong. **MSGPFIX** fixes these errors.

NOTE: Don't use **MSGPFIX** unless you encounter a parameter error when using one of the other tools. Typically, these errors appear when you create translated message modules using the **MSGEXP** tool.

Replace *messageDB* with the path and filename of the message database to be fixed. If you omit the extension, .MDB is assumed.

Example

msgpfix myprog <Enter>

This fixes any parameter count errors in MYPROG.MDB.

MSGPURGE

Purges a source file's strings from a message database

Syntax

MSGPURGE *messageDB* *sourceFile*

Parameters

messageDB

Specifies the path and filename of the target message database. If you omit the extension, .MDB is assumed.

sourceFile

Specifies the name of the source file whose strings are to be purged.

Remarks

MSGPURGE purges a source file's strings from a message database. You should do this if you have deleted the source file from your program.

Replace *messageDB* with the path and filename of the target message database. If you omit the extension, .MDB is assumed.

Replace *sourceFile* with the name of the source file whose strings are to be purged.

Example

msgpurge myprog test.c <Enter>

This purges all the strings for TEST.C from MYPROG.MDB.

MSGRENAM

Renames a source file in a message database

Syntax

MSGRENAM *messageDB* *oldFilename* *newFilename*

Parameters

messageDB

Specifies the path and filename of the message database to be updated. If you omit the extension, .MDB is assumed.

oldFilename

Specifies the filename to be changed.

newFilename

Specifies the new filename.

Remarks

MSGRENAM renames a source file in a message database. You should do this whenever you rename a source file whose strings have already been extracted into the message database. Otherwise, when you rerun **MSGEXTR** or **MSGIMP**, the renamed file will be treated as a new (additional) file in the message database.

Replace *messageDB* with the path and filename of the message database to be updated. If you omit the extension, .MDB is assumed.

Replace *oldFilename* with the filename to be changed.

Replace *newFilename* with the new filename.

Example

msgrenam myprog test.c test1.c <Enter>

This renames TEST.C to TEST1.C in MYPR0G.MDB.

MSGSTATS

Gets statistics on message database string characteristics

Syntax

MSGSTATS *messageDB* [**/M**] *characteristics*

Parameters

messageDB

Specifies the path and filename of the message database to be queried. If you omit the extension, .MDB is assumed.

characteristics

Specifies options corresponding to various database flag and field settings:

c	C or assembly string
D	Passed documentation (user interface) review
I	Internal use only
L	Length preceded
l	Length limit specified
M	Message Librarian string
m	Multiple insertion parameters
N	Notes entered
P	Parameters described
p	At least one insertion parameter
R	Rejected in documentation review
s	Length limit too short (based on the expansion factors listed in “String Expansion Factors” on page 383.)
T	Should be translated

U	Enabling unfriendly
V	Source verified
X	Already translated

Remarks

MSGSTATS gets statistics on message database string characteristics.

Specify */M* to list the strings that have the specified characteristics. If you omit this option, only the number of strings with the specified characteristics is displayed.

The options are case sensitive, and correspond to various message database flag and field settings. You must type a plus (+) or minus (-) sign before each option to indicate whether to check for the presence or absence of the characteristic. **MSGSTATS** reports only those strings that have the combined set of specified characteristics.

Example

msgstats myprog +c -X <Enter>

This reports the number of C and assembly strings in MYPROG.MDB that haven't been translated. For example:

```
Looking for messages that:
     Are C strings
     Have not been translated

3 Messages tested, 3 matched
```

To display the three matching messages reported above, you would type

msgstats myprog /m +c -X <Enter>

Here's the output in this case:

```
Looking for messages that:
     Are C strings
     Have not been translated

C String:      1    MYPROG.C, line 4    Flags: T
Length limit:  None
```

```
Original text:
 My Program, Version 1.0\n
————————————————-
C String:      2    TEST1.C, line 4     Flags: T
Length limit:  None
Original text:
 This is a test program.\n
————————————————-
C String:      3    TEST1.C, line 5     Flags: T
Length limit:  None
Original text:
 It tests the message internationalization tools.\n
————————————————-
3 Messages tested, 3 matched
```

MSGTRAN

Creates fake translations in a message database

Syntax

```
MSGTRAN  messageDB   tranType
```

Parameters

messageDB

Specifies the path and filename of the message database to be translated. If you omit the extension, .MDB is assumed.

tranType

Specifies a variety of fake translation types:

ASESWAP	Reverse the case of all letters.
CASEUPPER	Make all letters uppercase.
CASELOWER	Make all letters lowercase.
WIDEROMAN	Change all letters and digits to double-byte Roman characters from the Shift-JIS (Japanese) character set. This helps test double-byte characters while retaining the readability of messages.
PIGLATIN	Convert words to Pig Latin. This expands each word by at least three characters.
EXPAND	Expand strings using the expansion factors listed in the appendix. Strings are expanded by appending periods and an @ sign to the end.
GERMAN	Convert strings to German-like text by umlauting the umlautable vowels and changing `ss'.
EUROPE	Convert strings to European-like text by putting diacritics on every letter possible.
CONCAT	Enclose each string in curly braces ({ }). If your program concatenates strings, the braces make it obvious.
REVERSE	Reverse the word order in each message, on a line-by-line basis.
BACKWARDS	Reverse the letter order in each word.

SWEDCHEF	Convert messages to pseudo-Swedish text.
RUSSIAN	Convert messages to pseudo-Russian text (using code points from code page 866).
NULL	Remove translations from the message database.

Remarks

MSGTRAN creates fake translations in a message database. This can be useful for testing internationalization. **MSGTRAN** translates the strings in the message database into a fake language, and replaces insertion parameters with numbered tokens. It translates only the strings that **MSGXFER** would export for translation.

You can also use **MSGTRAN** to remove translations from a message database.

Example

msgtran myprog wideroman <Enter>

This converts the strings in MYPROG.MDB to double-byte Roman characters. To remove the translation from the database, type as follows:

msgtran myprog null <Enter>

MSGXFER

Exports message database strings for translation

Syntax

MSGXFER *messageDB* *messageType* *xfrFile* *xfrDirection*

Parameters

messageDB

Specifies the path and filename of the message database to be used. If you omit the extension, .MDB is assumed.

messageType

Specifies which strings to transfer:

CSTRING	Transfer only strings that came from C and assembly files.
MSGLIB	Transfer only strings that came from Message Librarian files.
DEFAULT	Transfer whichever string type the source contains. (If both types are present, nothing is transferred.)

xfrFile

Specifies the file to be exported or imported. By convention, the file extension should be .XFR.

xfrDirection

Specifies whether the transfer file is to be imported or exported:

-O	Export from the message database.
-I	Import to the message database.

Remarks

MSGXFER exports message database strings for translation. The export process produces a transfer file, which is a text file containing the strings and certain keywords, which are defined in the appendix. Strings that have the Enabling unfriendly flag set to Yes or the Should be translated flag set to No aren't exported. Insertion parameters are represented as numbered tokens (<1>, <2>, and so on) in the exported strings.

You can also use **MSGXFER** to import translations to a message database. The input is a copy of the transfer file with translations added by the translator. During the import process, you are notified of the following: missing translations, translations with the wrong number of insertion parameters, translations that exceed specified length limits, and problems noted by the translator.

Example

msgxfer myprog cstring myprog.xfr -o <Enter>

This exports the C and assembly strings from MYPROG.MDB to a transfer file named MYPROG.XFR. Here's an example of what the transfer file looks like:

```
$PROGRAM=My Program
$VERSION=1
$DATE=03/03/1992

$MESSAGE=1
$TEXT=2
Mask <1> is not supported because
it conflicts with mask <2>
$PARMS=2
<1> is number of first mask
<2> is number of second mask

$MESSAGE=2
$TEXT=1
Press <ENTER>  to continue:
$NOTES=1
Don't translate "<ENTER>".
$LENGTH=26
```

Here's an example of what the transfer file looks like on import, with the portions added by the translator emphasized:

```
$PROGRAM=My Program
$VERSION=1
$DATE=03/29/1992 $LANGUAGE=Espanol $COMMENT=1 Translated by
Reyna Aburto.

$MESSAGE=1
$TEXT=2
Mask <1>  is not supported because
it conflicts with mask <2>
$TRAN=2 No se da suporte a la mascara <1>  porque hay conflicto
con la mascara <2>
$PARMS=2
<1> is number of first mask
<2> is number of second mask

$MESSAGE=2
$TEXT=1
Press <ENTER> to continue:
$TRAN=1 Pulse <Enter> para continuar:
$NOTES=1
Don't translate "<ENTER>".
$PROBLEM=2 I cannot keep the translation under 29 characters
without using funny abbreviations
$LENGTH=26
```

Here's how you would import translations from the above transfer file into the message database:

msgxfer myprog cstring myprog.xfr -i <Enter>

Revision History

The following table lists changes made to the NLM User Interface Developer Components documentation:

Release Date	Revision Description
September 2002	Updated the descriptions of several functions in “Sorting Lists” on page 25.
February 2001	Added NLM Internationalization Tools chapters (moved from *Internationalization*).
September 2001	Added text alternatives for graphics.
June 2001	Made changes to improve document accessibility.
	Added revision history.