Message System Design [2]

The UNICOS message system consists of a set of tools to build message text files into catalogs, to retrieve messages from catalogs, and to format messages to be issued to the user. Under the message system, all messages and explanations reside in a binary message catalog maintained on disk. No messages need to appear within program code.

This section describes each element of the message system from a design perspective. All terms and concepts involved in the message system are introduced.

The procedures for using the message system in a product are described in Chapter 3, page 27. That section presents a sample procedure for using message system tools in a program.

2.1 Overview

The elements of the message system are as follows:

- Message text files (*group*`.msg`)
- Message and explanation catalogs (*group*`.cat` and *group*`.exp`)
- Catalog creation utilities (`caterr`(1) and `gencat`(1))
- Message retrieval library functions (`catopen`(3), `catclose`(3), `catgetmsg`(3), and `catgets`(3))
- Message formatting library function (`catmsgfmt`(3))
- Explanation viewing utility (`explain`(1))
- Explanation extraction utility (`catxt`(1))
- Catalog search path utility (`whichcat`(1))
- User environment variables or locales for language, catalog path, and message format (`LANG`, `NLSPATH`, `MSG_FORMAT`, and `CMDMSG_FORMAT`)

These elements are described briefly in the following paragraphs. Figure 1 shows the relationships among these elements.

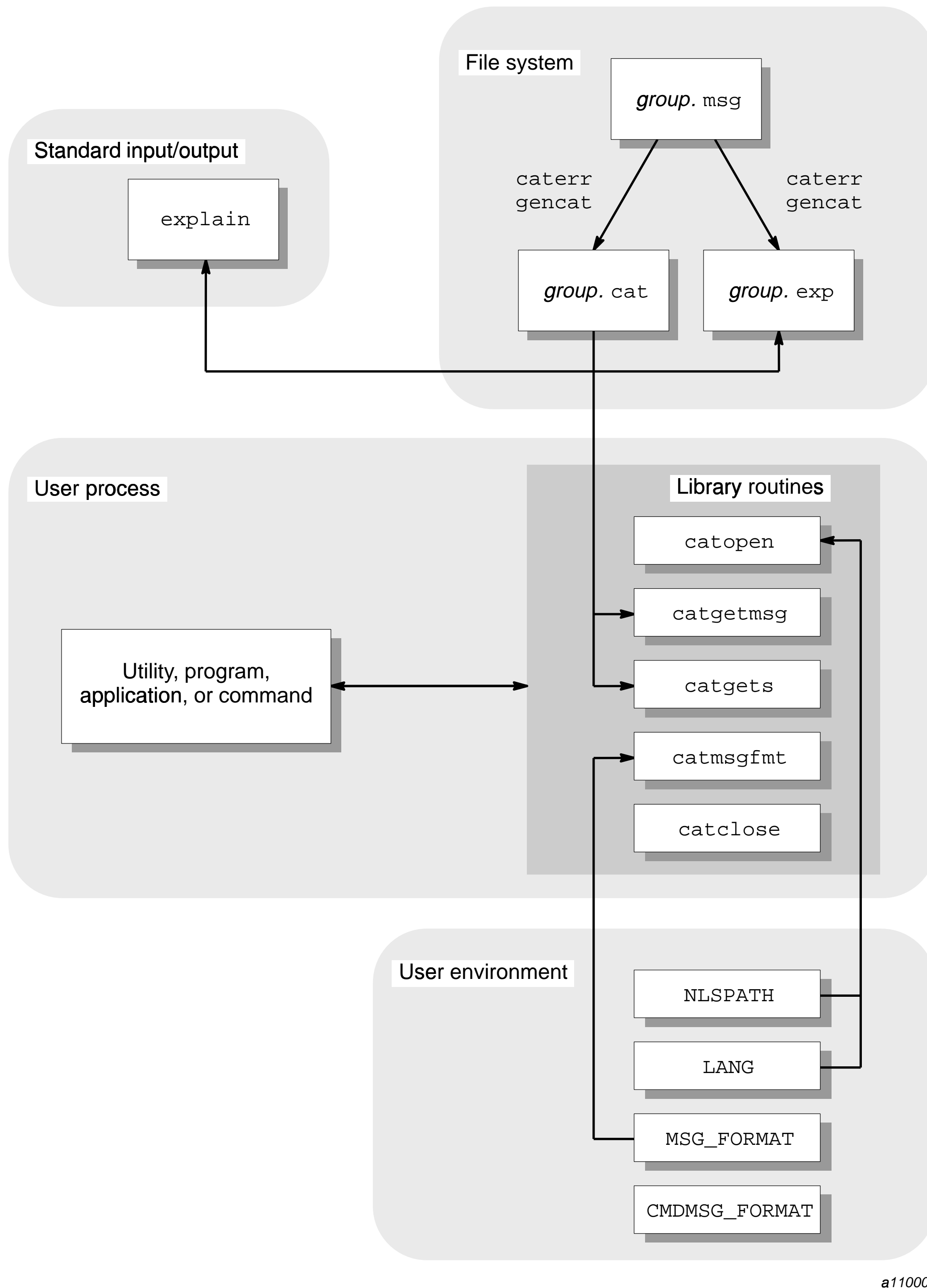

Figure 1. Message system overview

Under the message system, programs issue messages from catalogs. Each software product has a catalog of messages and a catalog of explanations. The source format of these catalogs is maintained in a *message text file* within the source directory tree for the product. The message system contains tools to build a *message catalog* and an *explanation catalog* from the message text file. The message text file and the two catalogs all use the *group code*, which identifies the product or product group, as part of the file name. Catalogs can be built from a message text file, either from the command line or from within an `nmake`(1) makefile. Catalogs are installed in the `/lib` or `/usr/lib` directory trees.

Programs gain run-time access to the message catalogs through library functions. These functions open and close catalogs, retrieve messages from a catalog, and format messages according to a user-specified pattern.

Users receive information from the online explanation catalog by using the `explain`(1) utility.

Users control the type and format of information output with an error message by setting the `MSG_FORMAT` and `CMDMSG_FORMAT` environment variables. They control the directory from which the error messages are retrieved by setting the `NLSPATH` environment variable. Users can determine which catalogs are being accessed and what catalog search path is being traversed by using the `whichcat`(1) utility.

If the messages are available in multiple languages, users control the language in which they receive messages by setting the `LANG` environment variable or the `LC_MESSAGES` locale.

Programmers, administrators, or support personnel who want to extract explanations from the message source file for publication can use the `catxt`(1) utility to extract explanations and to insert important message formatting macros.

For a complete description of the library functions, utilities, environment variables, and files that constitute the message system, see the man pages.

Each of the following subsections describes part of the message system.

2.2 Message text files

The message text file contains the source text for both the messages issued to users from a program and the message explanations available to users through the `explain`(1) utility. The message text file is the source for all messages and explanations to be processed and delivered by the rest of the message system. Figure 2 illustrates how the message text file is processed by and for other elements of the message system.

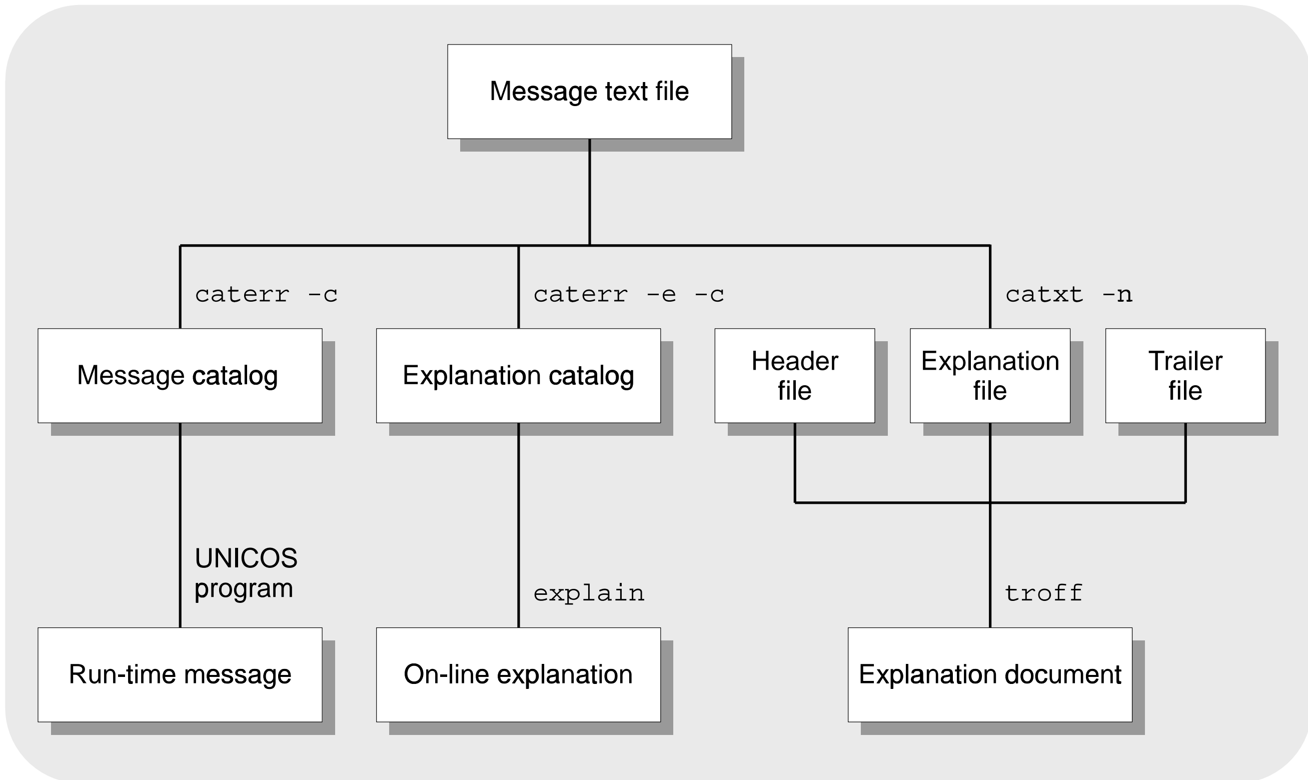

Figure 2. Processing the message text file

The message text file should be named as follows:

group.msg

This name is required to satisfy rules for catalog names and `nmake`(1) implicit rules. *group* is the group code that identifies your product. Several programs can use the same group code or a single program can use several group codes. The group code helps users determine the source of the message. The `.msg` suffix distinguishes a message text file from a message catalog (`.cat` suffix) or explanation catalog (`.exp` suffix).

The group codes `local`, `Local`, `LOCAL`, and all group codes that begin with `Z` (uppercase only) are reserved for site use. Catalogs that Cray Research supplies do not use these group codes.

The message text file can contain the following four basic types of information:

- Message text, preceded by the `$msg` tag
- Explanation text containing `nroff` formatting codes, preceded by the `$nexp` tag

- Plain ASCII explanation text, preceded by the `$exp` tag
- Comments, consisting of `$<space>` *text*, `$<tab>` *text*, or `$<newline>`

The following subsections describe the text associated with each type of tag.

2.2.1 Message text

A `$msg` tag precedes each message in the message text file. This tag is used by the catalog utilities to identify the associated text as a user message to be included in the message catalog. Each message entry must also include the message number.

The following subsections discuss these aspects of writing message text:

- Numbering of messages
- Ordering of messages
- Variables in messages
- Special characters in messages

2.2.1.1 Numbering of messages

Each message contained in the message text file must have a message number. The two types of message numbers are as follows:

- Literal numbers
- Symbolic names

Literal message numbers are integers that follow the `$msg` tag. Combined with the group code, the message number provides a unique message identifier for messages issued using the message system.

A typical message with a literal number appears as follows:

```
$msg 6 The daemon is unable to migrate the file.
```

Rather than literal message numbers, it is recommended that you use symbolic message names (that is, a symbol instead of a number). The purpose of symbolic names is to provide a cross-reference capability between message names and numbers.

A typical message with a symbolic name appears as follows:

```
$msg DGR_UTM The daemon is unable to migrate the file.
```

To use symbolic names, you must perform the following steps:

1. Create an include file to map the symbolic names to literal numbers.
2. Specify the include file in the message text file.
3. Use the -s option of the caterr(1) catalog generation utility when you generate the message and explanation catalogs from the message text file. (For a complete description of the caterr utility, see Section 2.3.3, page 18.)

Suppose you have a message text file (xyz.msg) that contains the following message definitions:

```
$msg EMLEVPAR Missing parameter to MLEV routine
$msg EMLEVPMI Parameter to MLEV routine must be a positive integer
```

An include file (xyzcodes.h) can be created to map the symbolic names to literal numbers. This include file would appear as follows:

```
#define EMLEVPAR 500   /* Missing parameter to MLEV routine */
#define EMLEVPMI 501   /* Parameter to MLEV routine must be positive *
```

You must add the following line before the first message in the message text file:

```
#include "xyzcodes.h"
```

A message catalog (xyz.cat) can be created from the message text file that contains the symbolic names by using the following utility:

```
caterr -s -c xyz.cat xyz.msg
```

The -s option calls the cpp(1) C language preprocessor, which maps the symbols to numbers based on the definitions in the include file. These include files also can be included in C source code files to provide access to the same symbolic message names.

Symbolic error codes can be created in any language, if the compiler for that language has a capability comparable to #include. In some cases, the cpp(1) utility might not be appropriate to do the symbolic-to-numeric mapping, because it processes only C-style include files; instead, a stand-alone program may be required to do the mapping.

Whether you use literal or symbolic message names, separate the $msg tag from the message number with at least one space or tab. If you use more than one space or tab, the file is still processed correctly, but the extra spaces or tabs are removed during text-to-catalog processing.

Separate the message number from the message text with one space. If you use more than one space, all spaces after the first are processed as leading spaces in the message text.

2.2.1.2 Ordering of messages

Messages must appear in ascending order, but they are not required to be consecutive. For example, all three of the following message numbering systems are acceptable:

Example 1:

```
$msg 1 Message one
$msg 2 Message two
$msg 3 Message three
```

Example 2:

```
$msg 100 Message one
$msg 101 Message two
$msg 102 Message three
```

Example 3:

```
$msg 150 Message one
$msg 160 Message two
$msg 170 Message three
```

Space is not allocated in the message file for each possible number in the sequence. Therefore, messages numbered as shown in example 2 or 3 require the same storage space as messages numbered as shown in example 1.

2.2.1.3 Variables in messages

Many messages contain variables that are supplied at run time. Variables can be included in messages by using the `printf`(3) format codes (for example, `%s`, `%d`, and `%f`) in the message that appears in the message text file. The message is returned from the catalog with the code embedded. You construct a print statement that supplies the proper value for the variable at run time.

> **Note:** Use single quotation marks (`' '`)to enclose user-supplied strings (such as file names and user IDs) that are referred to as tokens. The use of quotation marks highlights for users information that is specific to the situation and reduces the possibility of variables being interpreted with a literal meaning. It is not a requirement to use quotation marks to enclose numeric values, language keywords, or other literal replacement strings.

A typical message text entry might appear as follows:

```
$msg 100 Unknown account name '%s'.
```

When printed at run time for a user who has entered `abcd` as an account name, the message appears as follows:

```
Unknown account name 'abcd'.
```

For an example of code to retrieve a message and modify it, see Section 3.3, page 30.

2.2.1.4 Special characters in messages

Messages that extend past the length of one physical line in the message text file must contain a continuation character (\) at the end of each continued line of message text source. The last line of the message text must not end with a \ character because it is not continued.

The following example illustrates a message whose source exceeds one line:

```
$msg 104 A report modification option was used \
in the command line, but a report was not \
requested.
```

You can embed special characters within the text of the message by using escape sequences (initiated with the \ character). Table 1 lists the escape sequences that are allowed in messages and unformatted explanation text.

Table 1. Special characters used in messages and explanations

Sequence	Character
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
\ *nnn*	ASCII character corresponding to the octal value *nnn*

Use newline characters within a multiline message to indicate where the lines should break on the screen.

Any characters other than those listed in Table 1 are passed through without the backslash (for example, `\q` produces `q`).

Although special characters are sometimes necessary in the message text, they make it difficult for users to control the layout of the error message through the MSG_FORMAT and CMDMSG_FORMAT environment variables. For more information about how the message system formats messages, see Section 2.5, page 21.

2.2.2 Explanation text

Each message entry should have a corresponding explanation. The message system accepts the following two types of explanations:

- Formatted explanations that contain formatting macros
- Unformatted explanations that consist of plain ASCII text

> **Note:** Message explanations originating within the Cray Research Software Division (as opposed to on site) are formatted using the nroff message macros contained in the tmac.sg file (see msg(7D)). The option to use unformatted ASCII message explanations exists for the convenience of customer programmers who want to use the message system, but do not want to typeset the explanations for hard-copy printing.

The following subsections discuss details specific to formatted and unformatted explanation text.

2.2.2.1 Formatted explanation text

An $nexp tag at the beginning of the text identifies formatted explanation text. Each formatted explanation text entry must include the message number and the text of the message with variable names inserted in place of variable symbols.

Use the message macros provided with the message system tools to mark up the explanation text. (The message macros are defined in the /usr/lib/tmac/tmac.sg file and are described on the msg(7D) man page.) The message macros are collections of nroff(1) and troff(1) text formatting directives defined for use with the message system.

It is a convention to use italics for variable names in formatted message text. (Italic characters usually appear as underscored or reverse video text online.)

An nroff explanation does not require continuation characters at the end of lines.

An explanation might appear after markup as follows:

```
$nexp 100
The account name \&'\*Vacid\*C' is not recognized.
.PP
```

```
The account ID (\fIacid\fR) specified with the
-a option is not a known account name. Verify
that the ID you entered is a valid account ID on
the system.
.ME
```

For instructions on marking up explanations for publication in the format of Cray Research message documentation, see Chapter 4, page 37.

2.2.2.2 Unformatted explanation text

Sites that elect not to typeset their explanations, but that want to create a message catalog for site-specific software, can use unformatted message explanations.

Each unformatted explanation text entry begins with an `$exp` tag and includes the message number and the text of the message, with variable names inserted in place of variable symbols. A continuation character (\) must appear at the end of each continued line of a multiline unformatted explanation. The last line of the explanation must not end with a \ character because it is not continued. Use angle brackets (< >) for variable names in unformatted explanation text. (Italics are usually used for variables, but italics are not available in unformatted text.)

You must specify the locations of newline and other special characters in an unformatted explanation. Table 1, page 10, summarizes the special characters. If you do not specify newline characters, none are used. This could render the explanation unreadable.

The unformatted version of the explanation presented on page 11 would appear in the text file as follows:

```
$exp 100 The account name '<acid>' is not recognized.\n\
\n\
The account ID (acid) specified with the\n\
-a option is not a known account name. Verify\n\
that the ID you entered is a valid account ID on\n\
the system.\n
```

2.2.3 Comment text

The `$` tag indicates that all remaining text on the source file line is a comment. A space or tab must appear between the `$` and the first character of the comment, or the `$` must appear as the only character on the line. Comments cannot consist of more than one line.

The following example shows four comment lines:

```
$ The following text contains the messages
$ and explanations for the ja(1) command.
$ These messages are part of the "acct"
$ software group.
```

Use comments rather than blank lines to create white space in the source file. Blank lines are significant to the `nroff` and `troff` text formatters and can create extra vertical spacing in online and printed explanations.

2.2.4 Combining text types in a file

The only rule governing how you can combine the four types of text in a message file is that messages and explanations must appear in ascending numerical order. One common arrangement is for messages and explanations to appear in an alternating order.

The following example illustrates this arrangement:

```
$msg 100 Text of message 1
$nexp 100
Text of message 1 with variables inserted
.PP
The explanation for message 1
.ME
$
$msg 101 Text of message 2
$nexp 101
Text of message 2 with variables inserted
.PP
The explanation for message 2
.ME
```

Another possible arrangement is to group all of the messages together, followed by all of the explanations.

The following example illustrates this arrangement:

```
$ Messages
$msg 100 Text of message 1
$msg 101 Text of message 2
$
$ Explanations
$nexp 100
Text of message 1 with variables inserted
```

```
.PP
The explanation for message 1
.ME
$nexp 101
Text of message 2 with variables inserted
.PP
The explanation for message 2
.ME
```

Any other arrangement in which messages and explanations are presented in ascending order can be processed successfully by the catalog generation tools. For the purposes of arranging the catalog, formatted and unformatted explanations are interchangeable.

Comments can appear before, after, or between any of the other text types (that is, `$msg`, `$nexp`, and `$exp`) but cannot appear within them.

2.3 Message and explanation catalogs

The message system uses message catalogs and explanation catalogs. *Message catalogs* contain the text of user messages issued by the program or programs of a particular software group. The message catalog is the run-time source of messages issued to users. Typically, *explanation catalogs* contain copies of each message in the message catalog, along with an accompanying explanation of the cause of the message, and actions suggested to remedy the error condition.

Both types of catalogs are generated from the message text file. When you have created a message text file, run the `caterr`(1) utility, using the message text file as input, to convert the message text file into the form that is used by the message system library functions. When invoked with the `-c` option, `caterr` calls a utility named `gencat`(1) to build a binary message catalog or a binary explanation catalog. (For more information on `caterr` and `gencat`, see Section 2.3.3, page 18.) To produce a message catalog and an explanation catalog from one message text file, you must run `caterr` twice.

The following subsections discuss the location of message and explanation catalogs and explain how to use `caterr` to build them.

2.3.1 Catalog search path

The `LANG` and `NLSPATH` environment variables and the `LC_MESSAGES` category determine the search path on the disk for the message and explanation catalogs. (The

acronym NLS refers to the X/Open Native Language System on which the UNICOS message system is based.)

The use of environment variables and categories to determine the catalog search path gives users and program developers control over which catalogs the message system library functions access.

2.3.1.1 The LANG variable

The `LANG` environment variable and the `LC_MESSAGES` category identify the user's requirements for native language, territory, and coded character set. These components are specified in a string of the following form:

language[*_territory*[*.codeset*]]

The string `En` is the designation for the English language. Other language, territory, and code set designations (if any) are defined and supported locally.

The value of *language* is part of the internal value of the `NLSPATH` environment variable.

2.3.1.2 The NLSPATH variable

The `NLSPATH` environment variable contains the message system search path; that is, the message system searches for catalogs in the directories specified by the value of `NLSPATH`. If the catalog is not found on the user search path (or if the user does not define `NLSPATH`), the internal value of `NLSPATH` is searched.

In addition to string literals, `NLSPATH` can contain any of the following variable fields:

Field	Description
`%N`	The value of the *name* parameter passed to `catopen`. This is the same as the group code.
`%L`	The value of the `LANG` environment variable or the `LC_MESSAGES` category.
`%l`	The language component of the `LANG` environment variable or the `LC_MESSAGES` category. This component determines the language in which messages are displayed.
`%t`	The territory component of the `LANG` environment variable or the `LC_MESSAGES` category.

%c The code set component of the LANG environment variable or the LC_MESSAGES category.

The file name specified in the NLSPATH environment variable must be the name of the message catalog (not the explanation catalog) to be referenced. For example, to specify that the message system should search the *group* .cat file in the /usr/tmp directory, specify the following NLSPATH definition:

```
/usr/tmp/%N.cat
```

The message system replaces %N with the group code you pass to catopen(3) or explain(1). For example, if your group code is lib, the message system would search for a message catalog called /usr/tmp/lib.cat.

The explain utility changes the .cat suffix to .exp before searching for the explanation catalog. Therefore, using the NLSPATH defined in the previous example and a group code of lib, explain would search for the explanation catalog named /usr/tmp/lib.exp.

> **Note:** You must always use %N for the catalog name in the definition of the NLSPATH environment variable. If you hard code the catalog name, the message system tries to retrieve all messages from the catalog you specify. For example, if you set the NLSPATH environment variable to /usr/tmp/lib.cat, the message system searches this catalog for errors from any product. This could cause a library message to be issued in a situation in which another product's message should have been issued. Using the %N variable as the catalog name prevents this error.
>
> Also, you must never specify the explanation catalog in the NLSPATH environment variable. If you specify the path name /usr/tmp/%N.exp in NLSPATH, the message system will access the explanation, rather than the message when it retrieves the message by using the catgetmsg(3) or catgets(3) function. Use the .cat (not the .exp) suffix in NLSPATH declarations.

If the message system searches the paths specified by the NLSPATH variable and does not find the file it is looking for, or if the user has not defined NLSPATH, the message system searches its internally specified path. This path is defined as follows:

```
/usr/lib/nls/%l/%N.cat:/lib/nls/%l/%N.cat:/usr
/lib/nls/En/%N.cat:/lib/nls/En/%N.cat
```

Most message and explanation catalogs are located on disk in the /usr/lib/nls/En directory. Catalogs that must be present for the system to work when the /usr/lib file system is not mounted are located in the /lib/nls/En directory. Thus, if the LANG or LC_MESSAGES language designation variable is set to an unsupported value, the English catalog is still searched. Users with an unset or incorrectly set LANG environment variable or LC_MESSAGES category always receive messages in English.

To determine which catalog is returning a message or explanation, use the whichcat(1) utility. This utility verifies that the expected catalog is being referenced.

The syntax `whichcat -l` returns a list of the path names that are searched when looking for the catalog. If no message or explanation catalog is found, this usage can help you to determine why.

2.3.2 Catalog names

Message catalogs are named by group code with a `.cat` suffix added (for example, the messages for the library group are in a catalog named `lib.cat`). Explanation catalogs are named by group code with a `.exp` suffix added (for example, the explanations for the `lib` group are in a catalog named `lib.exp`). This naming convention is required to satisfy rules for catalog names and nmake(1) implicit rules.

The catopen(3) function references the NLSPATH environment variable when determining the message catalog to open. For example, if the user has not set NLSPATH, and neither LANG nor LC_MESSAGES is set, and you pass the catalog name `lib` to `catopen`, `catopen` tries to open the catalog named `/usr/lib/nls//lib.cat`. If that catalog does not exist, `catopen` tries to open the catalog named `/lib/nls//lib.cat`, `/usr/lib/nls/En/lib.cat`, and then `/lib/nls/En/lib.cat`. If no catalog exists, an error condition has been encountered. For information about the different types of catalog errors you may encounter and recommendations for handling them, see Section 2.4.1, page 20.

The explain(1) user utility references the NLSPATH environment variable when determining the explanation catalog to open. For example, a user enters one of the following utilities:

```
explain lib1001

explain lib-1001
```

Using the internal values of NLSPATH and either LANG or LC_MESSAGES for `%l`, the `explain` utility searches for the following catalogs in succession:

```
/usr/lib/nls/%l/lib.cat
/lib/nls/%l/lib.cat
/usr/lib/nls/En/lib.cat
/lib/nls/En/lib.cat
```

You can change the value of NLSPATH so that the message catalogs can be located in any directory. You may want to change the value of NLSPATH when you are developing code, locate the message catalog in a local directory, and change NLSPATH to point to that local directory.

2.3.3 Generating catalogs

Use the caterr(1) utility to convert your message text file to a binary message catalog and a binary explanation catalog. You must invoke caterr twice to generate both types of catalogs.

The syntax for caterr is as follows:

```
caterr [-c catfile] [-e] [-s[-P cpp_opts]] [-Y x,pathname] [msgfile]
```

The caterr utility processes the message text file (*msgfile*) to prepare it for conversion to a catalog. (If *msgfile* is not specified, the input is read from stdin.) The conversion to a catalog is actually performed by a second utility called gencat(1). However, you can use the -c option to caterr to instruct caterr to call gencat automatically. If you use the -c option, caterr outputs the catalog and names it *catfile*.

It is recommended that you use caterr with the -c option. (The gencat utility exists as a separate utility to maintain compatibility with the X/Open standards for message catalog processing. There is no advantage in calling gencat separately.) By default, caterr looks for gencat in the /usr/bin/gencat file.

By default, the caterr utility generates a message catalog. To generate the explanation catalog, use the -e option.

Message text files can contain symbolic message codes instead of message numbers. (For a definition of symbolic message codes, see Section 2.2.1.1, page 7.) The -s option to caterr calls the C preprocessor (cpp(1)) to process the symbolic codes in the message text file into message numbers according to a mapping defined in an include file specified in the message text file. The -P suboption to the -s option passes the contents of a string enclosed in quotation marks to cpp for processing. Use the -P suboption if you need to pass options and parameters to cpp from the caterr command line.

If $nexp explanation tags are encountered in the message text file, the caterr utility calls the text formatting utility nroff as part of its processing of the message text file. nroff uses message macro definitions to format the explanation text. By default, caterr looks for nroff in the /usr/bin/nroff file and for the message macros in the /usr/lib/tmac/tmac.sg file.

The -Y option lets you specify the version of nroff, gencat, and the tmac.sg message macros that caterrcalls. This option is needed primarily when caterr is used in the system generation environment. For examples of using the -Y option, see the caterr(1) man page.

The following example uses `caterr` to generate a message catalog named `lib.cat` from the message text file `lib.msg`:

```
caterr -c lib.cat lib.msg
```

The following example uses `caterr` to generate an explanation catalog named `lib.exp` from the message text file `lib.msg`:

```
caterr -e -c lib.exp lib.msg
```

Remember to invoke `caterr` twice to generate both a message and an explanation catalog. For more information about generating catalogs, see the `caterr`(1) and `gencat`(1) man pages.

2.4 Retrieving messages

To access the message catalog from your program, use the `catopen`(3), `catclose`(3), `catgetmsg`(3), and `catgets`(3) library functions. For the details of calling these functions, see the man pages.

To retrieve a message from the catalog, open the catalog by using `catopen` and then retrieve the message by using either `catgetmsg` or `catgets`. The nature of your program and the type of messages it issues determines which of these two functions you use. If the program usually issues fatal messages and then aborts, you should use `catgetmsg`. If the program issues many messages and continues processing, you should use `catgets`.

The two functions are used in separate situations because they use system resources differently. `catgetmsg` reads into a user buffer the message corresponding to the message ID that you pass to it. `catgets` reads the entire set into an internal buffer. This has the effect of reading in the entire catalog, because Cray Research message catalogs are structured as a single set.

Because of this difference, `catgetmsg` is more efficient in situations in which only a few messages are issued, where error conditions are usually fatal, or where there are many messages and a program cannot afford the increased size at run time. Library functions and most utilities are examples of programs that should use `catgetmsg`.

The `catgets` function is more efficient in situations in which many messages are issued during the execution of the program. It is unnecessary to access the disk each time a message is read from the catalog, because all of the messages are in a buffer. Compilers are an example of programs that can gain an advantage from using `catgets`.

If catgetmsg or catgets fails because the message catalog identified by the catalog descriptor is not available or because the requested message is not in the catalog, a pointer to a null ("") string is returned.

When you are finished with a message catalog, close it by using the catclose library function.

2.4.1 Retrieval errors

It is possible that an error might occur during your attempt to open the message catalog or to retrieve a message. The message system library functions let you write your code assuming that the message retrieval will succeed. If the retrieval does not succeed, your program can continue processing despite the failure.

You do not need to perform a specific check to determine whether a catopen function fails, because the next catgets or catgetmsg will fail if the catalog is not available.

If you issue a correct catgetmsg or catgets function, you can encounter only two types of errors:

- The catalog is unavailable.
- The catalog is available, but the requested message is not available.

The catgets function returns a pointer to the default string *s*, which you passed to catgets, in response to either of these errors.

The catgetmsg function returns a pointer to a null ("") string in response to either of these errors. You can create a default message by placing it into the buffer used by catgetmsg. If the catgetmsg function fails, your default message will be undisturbed.

This default message capability allows (but does not require) your program to distinguish between these two types of failures. As with almost any call to a library function, you must decide on the level of fault tolerance or error recovery appropriate to your program.

The __catopen_error_code() internal routine also is available to help you diagnose the cause of a failed catopen call. (A failed catopen call is one which returns a value of -1.)

The __catopen_error_code routine returns a nonzero value indicating the reason for the failure. A return value less than 0 indicates that the problem is an error internal to the program. A return value greater than 0 indicates that the problem is a system error.

The internal error codes have symbolic names defined in the nl_types.h header file. These names and definitions are as follows:

Error name	Description
NL_ERR_ARGCNT	catopen was called with less than two arguments.
NL_ERR_ARGNULL	The *name* argument to catopen is NULL.
NL_ERR_MALLOC	catopen was unable to allocate memory (using malloc(3)) for internal structures.
NL_ERR_HEADER	catopen was unable to validate the message catalog file header as a valid message catalog file.
NL_ERR_VERSION	catopen found an invalid version number in the message catalog file header.

System error codes are the system return values defined in the errno.h header file. (These codes are documented on the intro(2) man page.) System error codes are generated in the following cases:

- catopen was unable to successfully open (using open(2)) any of the message catalog files specified in the NLSPATH environment variable search path.
- catopen was unable to successfully read from (using read(2)) or set the read/write file pointer to (using lseek(2)) the message catalog file header and set directory.

2.5 Formatting messages

The message system can format a message before you print it. The message is formatted according to the format pattern specified by the user in the MSG_FORMAT and CMDMSG_FORMAT environment variables. For details about the difference between these two message formatting environment variables, see the explain(1) man page.

The MSG_FORMAT and CMDMSG_FORMAT environment variables hold a pattern constructed from the following replaceable characters:

Character	Description
%C	Command name
%D	Debugging information
%G	Group code
%M	Message text
%N	Message number

`%P` Position of the error

`%S` Severity

`%T` Time stamp

If any of the `%` fields is not present in the variable definition, the corresponding message field is not printed.

The format of the time stamp (`%T`) is equivalent to that produced by the `cftime`(3) function and can be overridden by the `CFTIME` environment variable. For details about time-stamp formats, see the `strftime`(3) man page, which documents the `cftime` function.

The `MSG_FORMAT` and `CMDMSG_FORMAT` environment variables also accept `printf`(3) escape sequences. Table 2 lists these special character sequences.

Table 2. Special characters accepted by `MSG_FORMAT` and `CMDMSG_FORMAT`

Description	Symbol	Sequence
Newline character	NL (LF)	`\n`
Horizontal tab	HT	`\t`
Vertical tab	VT	`\v`
Backspace	BS	`\b`
Carriage return	CR	`\r`
Form feed	FF	`\f`
Audible alert	BEL	`\a`
Backslash	\	`\\`
Question mark	?	`\?`
Single quote	’	`\'`
Double quote	"	`\"`
Octal number	*ooo*	*ooo*
Hexadecimal number	*hh*	\\x*hh*

The escape *ooo* consists of the backslash followed by 1, 2, or 3 octal digits, which are taken to specify the value of the desired character. A common example of this construction is \0, which specifies the null character. The escape \\x*hh* consists of the backslash, followed by `x`, followed by hexadecimal digits, which are taken to specify the value of the desired character. There is no limit on the number of digits, but the behavior is undefined if the resulting character value exceeds that of the largest character.

Any characters other than those listed in Table 2 are passed through without the backslash (for example, `\q` produces `q`).

In most cases, end your `MSG_FORMAT` and `CMDMSG_FORMAT` specification with a newline character (`\n`) so that any output that follows begins on a new line.

If `MSG_FORMAT` is not defined, messages are formatted according to the following default format:

```
%G-%N %C: %S %P\n  %M\n
```

For the default format of the `CMDMSG_FORMAT` variable and the order of precedence of variable evaluation, see the `explain`(1) man page.

This pattern produces a message of the following format:

groupname-msgnumber   command: severity   position
The text of the message

For example, library message number 1001, which is in the `lib` group and has a severity level of unrecoverable, would print as follows:

```
lib-1001  a.out: UNRECOVERABLE
  A READ operation tried to read past the
end-of-file
```

Because no position is specified, `%P` is replaced with a null ("") string.

Use of `MSG_FORMAT` and `CMDMSG_FORMAT` lets users control the message format. This gives users a common format to work with from product to product and allows the construction of more robust scripts to process messages. Users can format messages in a way that a script accepts, rather than changing the script to use the message format imposed by the program.

If you issue a message with replaceable parameters embedded in it, substitute the parameters in the message before passing it to the `catmsgfmt`(3) message formatting function. For example, the following message:

```
The account name 'account' is not recognized.
```

might be returned from the catalog as follows:

```
The account name '%s' is not recognized.
```

Before passing the message string to `catmsgfmt`, replace the `%s` character with its value. One way this can be done is by using the `sprintf` function (see `printf(3)`).

In the following example, the first line of code inserts the value of the `parameter` variable into the message in the buffer to which `p` is a pointer. The result is placed in `buf2`. The second line resets the pointer `p` to point to the modified string.

```
(void) sprintf(buf2, p, parameter);
p = buf2;
```

After parameter replacement, you can call `catmsgfmt` to format the message. `catmsgfmt` returns a pointer to the buffer that contains the formatted message. You can then print the message in any way and to any device that you choose.

The `catmsgfmt` function exists as a convenience to those who want to issue messages in the format specified by `MSG_FORMAT`. If you have a need for complex or program-specific formats, you can control the message formatting yourself with the output functions for the programming language you use.

> **Note:** Be cautious in creating hard-coded message formats. Users quickly grow accustomed to the flexibility of an environment variable and may create software that depends on a particular message format under the assumption that they can control message formats by using the `MSG_FORMAT` environment variable.

2.6 Special message types

Special considerations exist for working with certain types of messages. The following subsections discuss issuing the following message types by using the message system:

- System messages
- Version messages
- Usage messages

2.6.1 System messages

System messages are drawn from the `sys_errlist[]` structure. These messages are indexed by error number (`errno`) and are used by many programs throughout the system. The `sys_errlist[]` structure also is contained in a message catalog with the group code of `sys`. The text of standard system error messages appears in this catalog. An explanation catalog that contains explanations for the system messages also is provided. Your program can draw the text for system error messages from the catalog by using `sys` as the group code and the value of `errno` as the message number.

> **Note:** Be sure to save the value of `errno` to a variable before calling the message system. Otherwise, the value of `errno` may be reset during message processing and you could issue an inappropriate error message.

2.6.2 Version messages

A version message states the version of the product issuing the message. When issuing a version message from the message system, observe the following rules:

1. Pass the version number to be stated in the message from the calling program rather than coding it into the message text file. This is important; if the version number is coded into the message text file, the version message will return the version of the message catalog, rather than the version of the product.
2. Use the techniques described in Section 2.4.1, page 20, to ensure that the version message is always issued, even if the message catalog is unavailable for some reason. This is important because a discrepancy between the version of the product and the version of the message catalog cannot be investigated unless the version of the product is accurately reported by the code.

2.6.3 Usage messages

A usage message provides a summary of the correct syntax for a utility. The explanation for a usage message does not have to describe the utility's syntax in full detail. Instead, it is sufficient to refer the reader to the man page for the utility. The man page describes the syntax of the utility in complete detail.

If the usage message contains a complex syntax that is difficult to reproduce in the explanation, it is acceptable to restate the message simply as "Usage error" in the explanation. For example, the following portion of a message text file defines the full usage message to be issued by the `docexec` code, but abbreviates the message to "Usage error" in the explanation.

```
$msg 100 Usage: \n\
  docexec \n\
  docexec -i\n\
  docexec -b ifile [-o docname] [-l]\n\
  docexec -a docname -t doctitle -n number [-c catname] [-l]\n\
  docexec -d docname [-l]\n\
  docexec -g\n
  docexec -l\n\
$nexp 100
Usage error
.PP
Either an incomplete command line or an unrecognized option
was entered. For details about the \*Cdocexec\fR options, enter
the following command line:
.CS
     man docexec
```

```
.CE
.ME
```

2.7 User access to the message system

The message system provides users with online access to message explanations through the `explain`(1) utility. The syntax of the `explain` utility is as follows:

```
explain msgid
```

The user supplies the *msgid* (group code and message number) of the message to be expanded. The `explain`(1) utility retrieves the message explanation from the appropriate message catalog and outputs it to standard output.

A sample user session with `explain` appears as follows:

```
% explain dm100
A .keep file is not present for 'user'.

The dmlim(1) command did not find a file named
.keep in the home directory of the specified user.
To exempt files from migration, you must create a
file named .keep in your home directory. It should
contain the names of the files that you wish to
exempt from migration. The file names in this file
may contain standard wildcard characters.
```

The output of `explain` is piped through the pager specified in the `PAGER` environment variable. If `PAGER` is not specified, the default pager `more -s` is used.

For a complete description of the `explain`(1) utility, see the `explain`(1) man page.