UNICOS Message System Programmer's Guide

SG–2121 9.0

Cray Research, Inc.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk +1–612–683–5907
Fax number +1–612–452–0141

New Features

UNICOS Message System Programmer's Guide SG–2121 9.0

This revision of the UNICOS Message System Programmer's Guide, publication SG–2121, supports the UNICOS 9.0 release.

The following features have been added to the message system since the UNICOS 7.0 release when this manual was last printed. Information related to these features is marked in the text with change bars.

- The `catgen`(8) command was renamed `gencat`(1) to comply with the X/Open XPG4 standard for messages.
- The addition of the `CMDMSG_FORMAT` variable for prescribing the format of UNICOS command messages has been documented.
- The role of the `LC_MESSAGES` locale category in message system processing is discussed.
- The `msgfile` argument to the `caterr`(1) command is now optional. If `msgfile` is not specified, `caterr`(1) accepts input from standard input. This enhancement allows users to more easily convert multiple message text files to one catalog.
- The message system now allows a date and time stamp in the error message output format. To use this capability, use the `%T` field in the `MSG_FORMAT` environment variable.
- The `troff`(1) and `nroff`(1) utilities are part of UNICOS beginning with UNICOS 8.0. Formerly, these utilities were part of the separately orderable DWB product. This change removes the message system dependence on DWB.
- A warning about changing and adding messages on a Trusted UNICOS system was added to section 1.

Record of Revision

The date of printing or software version number is indicated in the footer. Changes in rewrites are noted by revision bars along the margin of the page.

Version	***Description***
6.0	January 1991. Original printing.
7.0	June 1992. Reprint with revision to reflect message system features added in the UNICOS 7.0 release.
8.0	January 1994. Revision to reflect message system features added in the UNICOS 8.0 release. This revision is only distributed online through Docview.
9.0	July 1995. Reprint with revision to reflect messge system features added since the last print.

Preface

This document discusses the UNICOS message system from the perspective of a programmer who wants to issue messages from code by using the message system library routines and message system catalogs. It contains information about how to create message and explanation catalogs and how to retrieve messages from those catalogs from within a program.

This information is useful to programmers using the message system, and to individuals (for example, system administrators) who want to understand the design of the UNICOS message system.

Related publications

For information on the message system from the point of view of the system administrator who is installing, maintaining, and updating message catalogs on a Cray Research system see *General UNICOS System Administration*, publication SG–2301

- For information about using the message system to retrieve message explanations, see the *User's Guide to Online Information*, publication SG–2143.

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

To order a manual, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907 or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Conventions

The following conventions are used throughout this manual:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage(`*x*`)`	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files) 4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command line.

The following machine naming conventions may be used throughout this manual:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, and CRAY C98D systems) CRAY EL series (CRAY Y-MP EL, CRAY EL92, CRAY EL94, and CRAY EL98 systems) CRAY J90 series (CRAY J916 and CRAY J932 systems) CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems) CRAY Y-MP E series (CRAY Y-MP 2E, CRAY Y-MP 4E, CRAY Y-MP 8E, and CRAY Y-MP 8I systems) CRAY Y-MP M90 series (CRAY Y-MP M92, CRAY Y-MP M94, and CRAY Y-MP M98 systems)
Cray MPP systems	All configurations of Cray massively parallel processing (MPP) systems, including the CRAY T3D series (CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC systems)
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release
SPARC systems	All SPARC platforms that run the Solaris operating system version 2.3 or later

The default shell in the UNICOS 9.0 release, referred to in Cray Research documentation as the standard shell, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Company Standard XPG4

The UNICOS 9.0 operating system also supports the optional use of the C shell.

The POSIX standard uses *utilities* to refer to executable programs that Cray Research documentation usually refers to as *commands*. Both terms appear in this document.

In this publication, *Cray Research*, *Cray*, and *CRI* refer to Cray Research, Inc. and/or its products.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Docview text-viewer system, which lets you see the text of a manual online. The Docview system is available on the Cray Research mainframe. To start the Docview system, use the `docview`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the `define`(1) command.

- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to Internet, using the following Internet addresses:

  `pubs2009@timbuk.cray.com` (comments on this manual)

  `publications@timbuk.cray.com` (general comments)

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Technical Support Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–612–683–5599.

- Use the postage-paid Reader's Comment Form at the back of the printed manual.

We value your comments and will respond to them promptly.

Contents

Contents

Page

Page

Page

Tables

Introduction [1]

User messages are the most important part of communication between software and its users. Messages tell users when the hardware or software cannot perform as requested. It is vital in these situations to report in accurate detail the circumstances of the problem and the path to a solution.

Cray Research has systems installed worldwide. Our users require not only accurate message information, but also access to the messages so that they can be translated to the native language of users who do not speak English.

The UNICOS message system consists of tools and procedures for issuing messages to users from program code and delivering documentation on those messages in a format that is suitable for translation.

The UNICOS message system is based on the X/Open Native Language System specification as described in the X/Open Company Standard XPG4. Cray Research has provided extensions to the standard that include a more complete set of tools and procedures for working with messages and message explanations.

Any programmer can use the message system tools in a Cray Research UNICOS environment. This manual explains how the message system is designed to work and how you can use it from your programs. The examples in this document assume that you are using the C language; however, any language that can call C library functions can use the message system.

Warning: Trusted UNICOS sites can use the information and procedures outlined in the following subsections to change or add messages. However, for changed messages, you must not alter the original, underlying meaning of the message.

1.1 Message system features

The message system includes the following features, which aid in improving error reporting and problem resolution:

- Message catalogs, located separately from the program code, that contain the text of the messages issued at run time
- Explanation catalogs, located in the same directory as the message catalogs, that contain a discussion of the problem and suggested solutions
- Online access to message explanations through the `explain`(1) command

- User control of the message format through the MSG_FORMAT environment variable
- User control of the language of the message text (where translated messages are supplied) through the LANG environment variable
- Message and explanation text source files distributed with the release to allow local modifications of the message or explanation text
- Published guidelines for writing good messages and good message documentation

1.2 Document outline

Each section of this document discusses an aspect of the message system.

Chapter 2, page 3, describes each part of the message system and the purpose it serves.

Chapter 3, page 27, provides a sample procedure for converting an existing piece of software to use the message system.

Chapter 4, page 37, describes how to print message documentation formatted in the Cray Research message documentation format.

Appendix A, page 55, is an appendix that lists guidelines for writing effective messages and usable message explanations.

Appendix B, page 65, is an appendix that contains an example of a correctly formatted message system section.

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.

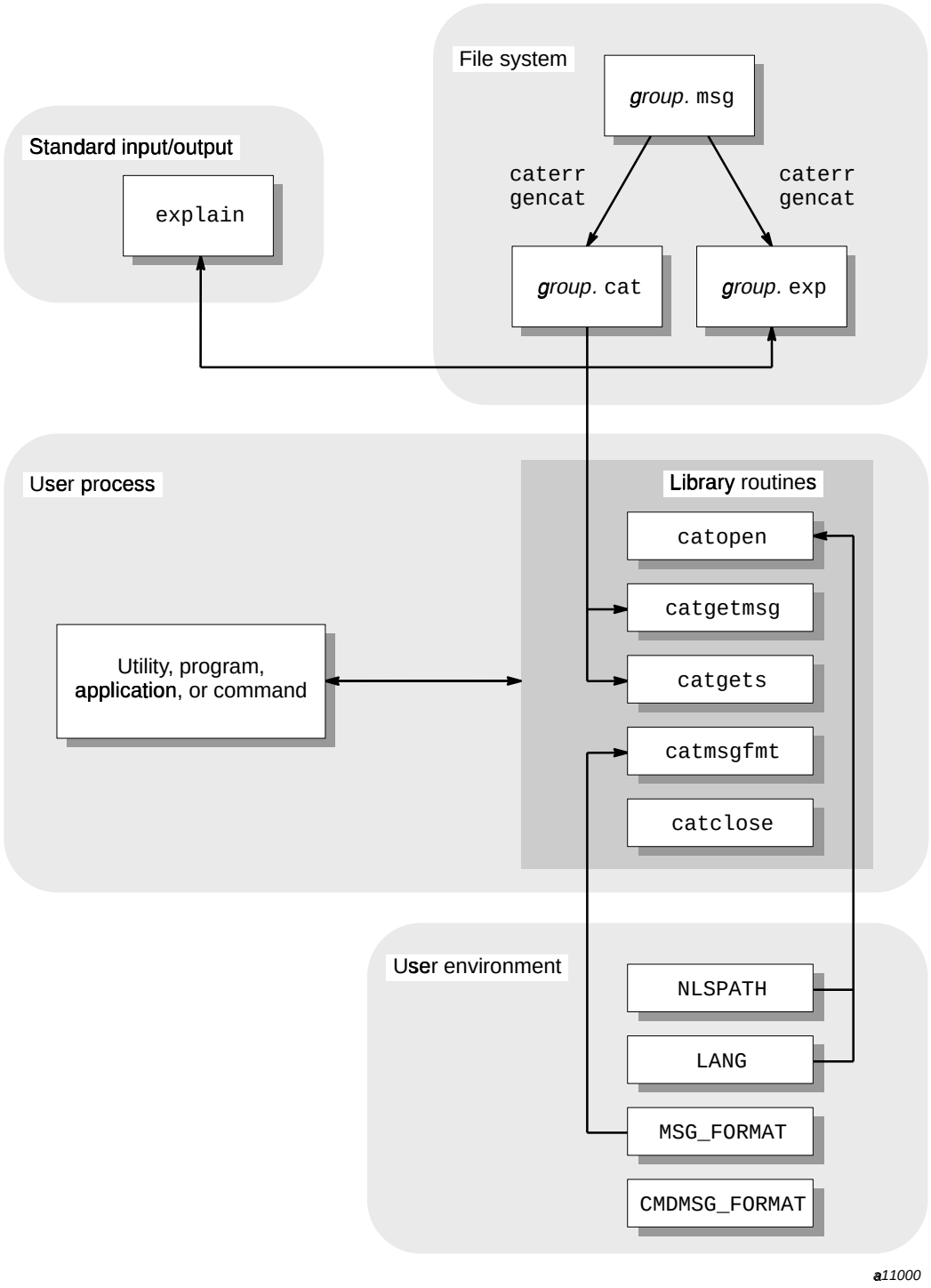

Figure 1. Message system overview

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.

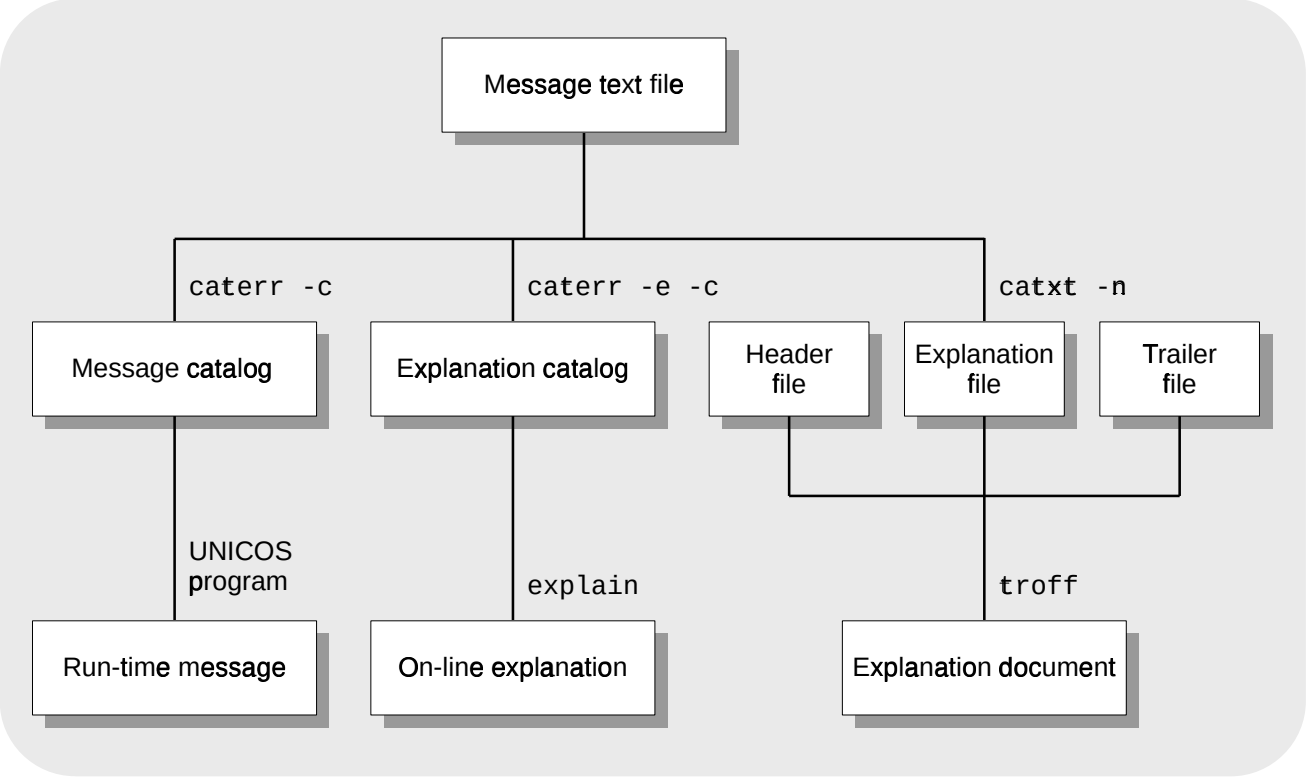

Figure 2. Processing the message text file

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.

Using the Message System [3]

Using the UNICOS message system requires changes to the way messages have traditionally been coded, tested, and documented in most organizations. This section explains how to use the message system as an alternative to coding messages within the program source and addresses some common questions that you may have about message system procedures.

Each subsection describes a step in a sample procedure for approaching the conversion of a program to use the message system. As the principal developer for a product, you must determine how the procedure applies to your product. The procedure also applies loosely to the creation of new code using the message system.

The procedure assumes that the product you are changing is coded in the C language; however, you can use the message system with any language that can interface with C language library functions.

3.1 Planning a conversion

When you are ready to convert a piece of code to the message system, the best first step is to survey the existing code to answer the following questions:

1. Where are the error messages located? Are they contained within one error-processing routine or are they dispersed throughout the program?
2. Are the messages generated using a consistent mechanism? For example, are all messages printed using `fprintf`(3)?

   If the messages are generated consistently, it is easier to extract them to build a message catalog. If the messages are generated by various mechanisms, you must create some method of extracting the messages.
3. What should the software group code be for the product? The group code you choose can be any alphanumeric string. The recommended length of a group code is 3 to 6 characters. Group codes cannot exceed 10 characters.

   The group codes `local`, `Local`, `LOCAL`, and all codes that begin with `Z` (uppercase only) are reserved for site use. Cray Research, Inc. recommends that sites use these codes to ensure that the release software does not contain a message file with the same group code as a local program. Using this naming convention also makes a clear distinction between local messages and release messages.

   The group code for each product must be unique. The `explain`(1) man page lists the group codes in use for the UNICOS operating system.

4. Is there more than one program in the group? It is possible for several programs with a related function to share a group code. For example, the ja(1) command (job accounting) may share a message catalog with other accounting code (for example, Cray system accounting (CSA)). If this is the case, how are message ID numbers divided among the various programs in the group?

   One common solution to this problem is to divide the catalog into ranges (for example, numbers 1 through 1000 are used for the first program in the group, numbers 1001 through 2000 are used for the second program in the group, and so on). You should select ranges that are appropriate for your software group.

3.2 Building a message text file

When you have looked at the code to be converted, chosen the group code to use, and decided on an approach to isolating the messages, you can build the message text file.

Use the following steps to build the message text file:

1. Extract a copy of the messages from the code and write them to a text file. UNICOS text processing utilities such as grep(1), awk(1), and sed(1) are useful for this process. If the messages are generated by a consistent mechanism, this will be an easy task. If they are not, this step will take longer.

2. Add a number or symbolic name to the beginning of each message. Using the convention you decided on during the planning step, number the messages. Each message must have a unique number. The numbers must appear in the text file in ascending order, but they do not have to be consecutive.

3. Edit the copy of the messages that you extracted in step 2. Remove the printing command (fprintf, printf, and so on). Delete variable argument names, the name of the command issuing the message, the severity level, and any quotation marks added for print command syntax. (The message formatting function, catmsgfmt(3), inserts the command name and severity level when it formats the messages.) If you have a newline character (\n) at the end of the message, delete it also. Add the $msg tag to the beginning of the message and place single quotation marks (' ') around variables.

   For example, if your code contains the following message:

   ```
   fprintf(stderr,"ja: Unknown account name
   %s\n", arg)
   ```

   you should edit the message line so that it appears in the text file as follows:

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

Edit each message in the text file in this way. The following listing shows a sample message text file.

```
$ message catalog for ja (part of group 'acct')
$msg 100 Unknown account name '%s'
$msg 101 Unknown group name '%s'
$msg 102 getoptlst() failed
$msg 103 Unknown user name '%s'
$msg 104 report modifying option(s) used without requesting a report
$msg 105 -m option cannot be selected when issuing a report
$msg 106 -m and -t options are mutually exclusive
$msg 107 -h option must be used with -l option
$msg 108 process is not part of a job
$msg 109 can't find TMPDIR in environment
$msg 110 can't make file name
$msg 111 file name exceeds max length
$msg 112 empty or nonexistent job accounting file
$msg 113 no commands seen
$msg 114 '%s' not removed
$msg 115 couldn't get space for selection by name
$msg 116 invalid regular expression for selection by name
$msg 117 couldn't get space for positioning marks
$msg 118 -p option's argument is invalid
$msg 119 cannot position to last entry
$msg 120 unable to position file
$msg 121 error in reading job accounting file
$   Next message is a warning
$msg 122 command flow tree overflow
```

The messages shown in the listing have not been edited to conform with the guidelines presented in Appendix A, page 55. You may want to edit your messages with the guidelines in mind at this point in the procedure, or you may want to complete the conversion of your code, perform preliminary testing, and then return to the message file and concentrate on improving the text of the messages.

4. Use the `caterr`(1) utility to build the text file into a binary catalog. Give `caterr` the name of your message text file and the name of the catalog file you want to produce; `caterr` processes the message text file into a catalog binary. For details of the syntax of the `caterr` utility, see the `caterr`(1) man page.

   For example, to build a catalog file called `/home/me/messages/lib.cat` from a message text file called `lib.msg` in the current directory, issue the following command:

   ```
   caterr -c /home/me/messages/lib.cat lib.msg
   ```

5. Change the NLSPATH environment variable to point to the output of the caterr utility. For example, if the catalog binary file is in the following directory:

   ```
   /home/cypress/me/messages
   ```

   set NLSPATH to the following value:

   ```
   /home/cypress/me/messages/%N.cat
   ```

   This lets you test your program by using a local message catalog.

3.3 Modifying the program source

The next step is to modify your program source to work with the message system. You must modify the program source in the following ways to call the message system correctly:

1. Add a line to include the <nl_types.h> header file in the program. The <nl_types.h> file defines variables used by the message system. For a description of the file, see the nl_types(5) man page.

   The include line appears in the program as follows:

   ```
   #include <nl_types.h>
   ```

2. Add a line to define a message catalog file descriptor:

   ```
   nl_catd  mcfd;
   ```

3. Change the code that issues each message. You can change the code on a message-by-message basis, or you can write a message routine that can be called each time you want to issue a user message. You decide which method works best for you, given the characteristics of your product.

   The following code example illustrates one possible message processing routine designed to be called each time a user message is issued. The code is offered here as an example of an error processing routine. It is specific to one piece of code (ja(1)) and may not work for any other application.

   The following assumptions were made when designing the routine:

   - Only two message severities are used by this code: warning and unrecoverable.
   - No more than one replaceable parameter was used in a single message.
   - Parameters substituted into messages are strings.

- The group code for this product is `acct`, and the command issuing the messages is `ja`.

You could easily modify the code to use more severity levels, more replaceable parameters, and different types of parameters.

```
/*
 *  Retrieve and print error message
 */
#include <stdio.h>
#include <nl_types.h>
#define BUFL 200

processerror (
  int err_num,      /* Message error number                    */
  int fatal,        /* Fatal flag (0 = warning, 1 = fatal)     */
  char *parameter   /* Optional substitution string parameter  */
)

{
        char    *s;             /* Error severity                */
        char    *p;             /* Pointer to error message      */
        char    buf1[BUFL];     /* Error message buffer          */
        char    buf2[BUFL];     /* Error message buffer          */
        char    buf3[BUFL];     /* Error message buffer          */
        nl_catd mcfd;           /* Message catalog file descriptor */
        /* Open the message catalog                              */
        mcfd = catopen("acct", 0);
        p = catgetmsg(mcfd, NL_MSGSET, err_num, buf1, BUFL);

        /* If a parameter was passed in, insert it into the message */
        if (_numargs() >= 3) {
                (void) sprintf(buf2, p, parameter);
                p = buf2;
        }

        /* Set s to the appropriate severity level               */
        if (fatal == 1)
                s = "UNRECOVERABLE";
        else
                s = "WARNING";

        /* Format the message using the catmsgfmt function       */
        (void) catmsgfmt("ja", "acct", err_num, s, p, buf3, BUFL);
```

```
        /* Print the formatted message to stderr                    */
        fprintf(stderr, buf3);

        /* If error is fatal, return error status                   */
        if (fatal == 1)
                return(1);

        return(0);
}
```

With this routine in place, the following line of code could be used to issue error message number 100 as an unrecoverable error with the `optarg` argument to be placed into the message:

```
processerror(100, 1, optarg);
```

The line or lines of code that print each message in the existing code must be changed to call the error processing routine.

3.4 Integrating message system files

Note: If your code is part of the UNICOS system, your completed code and message catalogs must be integrated and built into the UNICOS release. The following subsections describe the steps in the integration procedure. If you are using the message system in an application, these integration steps are not necessary.

3.4.1 Integrating messages into the PL

The message and explanation source should be placed in a file called *group*`.msg`. The file should be added to the UNICOS source manager (USM) source control program library (PL) for your product. Modifications to the message source should follow the same procedures as modification to other source elements of your product.

3.4.2 Building and installing the catalogs

The `nmake`(1) makefile for your product must be modified to build and install the message and explanation catalogs. `nmake` uses implicit rules to handle most of the process automatically. You must explicitly perform the following steps:

1. Name the message source file as *group* .msg; *group* is the group code for your product.
2. Decide where to install your catalogs. If your program must execute at times when only the root file system is available, it is usually installed in /bin. If this is the case, install the catalogs in /lib/nls/En.

   If your program is not required to execute when only the root file system is available, it probably resides in /usr/bin. In this case, install your catalogs in /usr/lib/nls/En.

   If your catalogs will be installed in /lib/nls/En, add the following statement to your makefile:

   ```
   NLSDIR =    $(ROOT)/lib/nls/En
   ```

   If your catalogs will be installed in /usr/lib/nls/En, you do not have to specify a definition for NLSDIR. This variable is predefined as /usr/lib/nls/En.
3. If you use symbolic message names, add the following line to the INIT section of your nmakefile:

   ```
   CATERRFLAGS  += -s
   ```

   This line calls caterr(1) by using the -s option.
4. Add the target names *group*.cat and *group* .exp to the sys or sysgen target list. nmake automatically creates message and explanation catalogs with those names from your *group*.msg file.

   If your program is called sample, and your group code is also sample, your target line might appear as follows:

   ```
   sys: sample sample.cat sample.exp
   ```
5. To install the catalogs, add the following statements to your installsys or installsysgen target:

```
$(CPSET) $(CPSETFLAGS) group.cat $(NLSDIR)/. $(CHMODR) $(OWNER) $(GROUP)
if [-s group.exp ]; then
    $(CPSET) $(CPSETFLAGS) group.exp $(NLSDIR)/. $(CHMODR) $(OWNER) $(GROUP)
fi
```

The message catalog must always be installed unconditionally.

Generation of the explanation catalog depends on the presence of the nroff program on the system. If nroff is present, the explanation catalog is generated and installed. If nroff is not present, a zero-length explanation catalog is produced. This zero-length catalog must not be installed. If the length of the catalog is 0, the -s test in the preceding code segment prevents installation of the explanation catalog.

The message text source file is delivered as part of both source and binary releases; therefore, the *group*`.msg` file must not be specified on any of the `rm` targets. This prevents removal of this file.

The message catalog (*group*`.cat`) file should be specified on either the `rmubin` or `rmrbin` target so that the file can be deleted along with other generated files associated with the product.

The explanation catalog (*group*`.exp`) file can be rebuilt only if `nroff` is available. To prevent removal of this file in cases where `nroff` is not available, add the following statements to the `rmubin` or `rmrbin` target:

```
if whence nroff > /dev/null ; then
    ignore $(RM) $(RMFLAGS) group.exp
fi
```

3.5 Maintaining message system catalogs

Code that uses the message system and the message system catalogs must be maintained from release to release. The following subsections discuss guidelines for adding, deleting, and changing messages.

3.5.1 Deleting a message from a release

You can delete a message from a release by removing its call from your product code. However, you should not remove the message or explanation text from the message text file or catalogs. Retain all of this text so that the message catalogs are upward compatible.

For example, if message number 50 is used for release 6.0 of the product, but not for release 6.1, the 6.0 version of the product will still execute correctly using the 6.1 catalog if message 50 is retained in the 6.1 catalog. Try to retain obsolete messages as long as the release that they support is still in use. If you are in doubt as to whether the release is still in service, do not reuse the message number.

Even if you eventually delete a message from the catalog because the corresponding software is totally obsolete, do not reuse the message number. Reusing message numbers could cause the wrong error message to be issued for an error condition. It is good practice to retire the message number, rather than reusing it.

3.5.2 Adding and changing messages

Adding new messages to a catalog is easy. Simply assign an unused number to the message, add the proper message call to the code, and add the message text and explanation text to the message text file.

If you need to change a message from one release to the next, you can do so by updating the message and the explanation in the message text file. Be cautious when changing the wording of a message so that you do not change the meaning. If you need to change the message in any significant way, create a new message. This policy maintains the upward compatibility of the message catalogs from release to release.

Publishing Message Explanations [4]

This section describes the procedures and guidelines to follow when formatting and processing message explanations for publication in a message document. Following these procedures lets you produce a message document in the style of the Cray Research message documentation. If you do not want to document your messages in that style, you do not need to follow these procedures.

The following topics are discussed in this section:

- Summary of publication procedures
- Message style definition
- Message markup
- Header and trailer files
- Extraction and printing
- Testing online explanations
- Troubleshooting

4.1 Summary of publication procedures

This subsection briefly summarizes the general steps required to format a message text file into both an explanation catalog accessible by using the `explain`(1) utility and a printed document suitable for publication as part of a Cray Research manual. This summary refers you to more detailed information that is contained in later subsections. Use this summary as an overview of the message process or, after you are familiar with the procedure, as a quick reference to the required steps.

1. Edit the message text file for content and format.

   Make changes to the text of the messages and explanations to bring them into conformance with the message guidelines (see Appendix A, page 55).

   Be especially careful of changes that you make to the actual error messages (text tagged with `$msg`). Do not change the order or number of variables in the message without changing the code that passes parameters to the message routines.

   Format the message explanations in conformance with the message style using the macros described on the `msg`(7D) man page. Make sure that the text of each

explanation (text tagged with $nexp) contains a copy of the message it explains. For a procedural guide to the message style, see Section 4.3, page 41.

2. Create or edit the header file for the message section. Create the trailer file.

   Create a file named *group*.head that contains the manual title, manual number, center footer, section title, and introductory paragraphs for the printed message section. The last line of this file must be a .2S macro that begins the 2-column format. See Section 4.4, page 45, for a description and an example of a header file. If this file already exists, make any necessary editing changes.

   Create a trailer file named *group*.trail that contains a .2E macro. This macro must be present to end the 2-column formatting.

3. Extract the explanations from the message text file; place them in a separate file.

   Use the catxt(1) utility to extract the explanations (text tagged with $nexp) from the message text file. Place the resulting text in a file named *group*.nexp. For example, the following command extracts the explanations for the data migration messages (group code dm) from the message text file dm.msg and places the result in the dm.nexp file:

   ```
   catxt -n dm.nexp dm.msg
   ```

   If you use symbolic names, you must use catxt with the -s option. For example, the following command extracts the explanations for the data migration messages (group code dm) from the message text file dm.msg; replaces the symbolic names with numbers based on a list contained in an include file that is specified in the dm.msg file; and places the result in the dm.nexp file:

   ```
   catxt -s -n dm.nexp dm.msg
   ```

   See Section 2.2.1.1, page 7, for a discussion of working with symbolic message names.

4. Print the message section by using text processing utilities on the Cray Research system or on your front end. Print the head, explanation, and trail files by using one command.

   See Section 4.5.2, page 49, for a sample command line to print the message document.

5. Repeat steps 1 through 4 until you are satisfied with the output.

6. Build the explanation catalog from the message text file by using the -c and the -e options of the caterr(1) utility.

The `caterr -c` utility builds binary catalogs that the run-time parts of the message system use. Specifically, a command of the following form builds an explanation catalog called *group*`.exp` from a message text file named *group*`.msg`:

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

For more information about the `caterr` utility, see the man page for the `caterr` utility or Section 4.6, page 49.

7. Set your `NLSPATH` environment variable to point to the explanation catalog you created in the previous step.

   The `NLSPATH` environment variable gives the file name of the catalog that the message system uses to look up explanations. The last node of the path name you specify with this variable must be `%N.cat`.

   For example, if the explanation catalog `dm.exp` (created in the previous step by using the `caterr` utility) is in the `/home/messages/dm` directory, set your `NLSPATH` environment variable as follows:

   ```
   setenv NLSPATH /home/messages/dm/%N.cat
   ```

   For more information, see Section 4.6.2, page 51.

8. Test the explanation catalog by requesting to view the explanations with the `explain`(1) utility.

   For example, to view the explanation for the message with the ID `dm-100`, issue the following command:

   ```
   explain dm100
   ```

   For more information on the `explain` utility, see Section 4.6.3, page 51.

9. Correct any problems in the explanation catalog by editing the message text file and building a new explanation catalog.

10. Repeat steps 6, 8, and 9 until you are satisfied that the online and printed message information is complete, consistent, and correct.

4.2 Message style definition

The following subsections provide information about the style used in message documentation. The message style consists of the following elements:

- Page layout
- Section heading

- Font and point size usage

Section A.2.2, page 63, contains an example of a document section formatted according to the message style.

4.2.1 Page layout

Messages are printed in a 2-column format. The page is divided into 2 columns that are 3.3 inches wide. A 0.25-inch gutter separates the columns vertically. A message ID bar appears at the beginning of each message. This bar, set off by horizontal lines, gives the message ID. The message ID consists of the group code, a dash, and the message number. Message text is in 9-point type.

4.2.2 Section heading

The section title of a message section is formatted the same as a section title in most Cray Research manuals. Introductory information that appears after the title and before the first message begins 1.375 inches below the bar under the section title. This text is in 11-point type and spans the width of the page. (The message style does not use the standard Cray Research publications modified 2-column format.)

4.2.3 Font and point size usage

The message system uses fonts in a manner consistent with the style used in other Cray Research manuals. Different font sizes are used because of the two-column format. Fonts are used as follows:

Font	Description
New Century Schoolbook	The default font. The body of the message explanation is set in 9-point New Century Schoolbook.
`Courier`	Used for all literals, including the copy of the error message that appears in the message explanation. Literals in the body of the explanations (commands, options, file names, and so on) are also in Courier.

Italic Used to denote variables. Variables in messages are discussed in greater detail in Section 4.3.4, page 43.

4.3 Message markup

The message style is achieved by using a set of `nroff`(1) and `troff`(1) macros called *message macros*. The following subsection describes how to use these macros to mark up a section of message documentation correctly.

The markup of a message section is different from that of other `nroff` and `troff` documents, because the message source file is processed not only by `nroff` and `troff`, but also by the `caterr`(1) and `catxt`(1) utilities. Because one of these utilities always processes the message text file before it is piped to `nroff` or `troff`, you must mark up the file to be acceptable to these utilities. This markup is then changed by the utility to be acceptable to `nroff` and `troff`.

4.3.1 Message text file

The message text file contains the marked-up messages and explanations. This file consists of messages, explanations, and comments. Each type of information is denoted by the use of a tag in the file. Messages are tagged with the string `$msg`. Explanations are tagged with either string `$nexp` or `$exp`. Comments are tagged with a dollar sign ($), followed by a space, tab, or carriage return.

4.3.2 Messages

Each message in the file must conform to the following rules of formatting:

- Begins with the string `$msg` followed on the same line by the message number (or symbolic name).
- Appears as a single logical line of text. If the message occupies more than one physical line, each physical line except the last must end with a continuation character (\) to make the message one logical line.
- Conforms to the guidelines for good messages as outlined in Appendix A, page 55.

The following example shows a message in the message text file that is identified by the `msg` tag and a message number:

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

4.3.2.1 Symbolic message names

Instead of using a number to identify each message, you can use a symbolic name. For example, the following message is written using the name DGR_UTM to identify the message in place of the number 6.

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

A header file is used to create a mapping between the symbolic names and the message numbers. For example, the following line could be used in a header file to map the name DGR_UTM to the number 6. The comment indicates the content of the message.

```
#define DGR_UTM  6  /* unable to migrate file*/
```

The name of the header file that contains the mapping must appear in the message text file so that the C language preprocessor (cpp(1)) can replace the symbolic names with the associated numbers before the catalog is generated by the caterr utility. For example, the following line is needed in the dm.msg file to include the dm_msg.h header file:

```
#include "dm_msg.h"
```

For information about printing a message text file that uses symbolic names, see Section 4.5, page 48. For information about creating an explanation catalog from a message text file that uses symbolic names, see Section 4.6, page 49.

4.3.3 Explanations

Each explanation in the file must conform to the following formatting rules:

- Begins with the string $nexp, which must be followed on the same line by the message number (or symbolic name).
- Contains a copy of the message. Like the message itself, the copy of the message must appear as one logical line. If the message occupies more than one physical line, each physical line except the last must end with a continuation character (\) to make the message a single logical line.
- Contains a .PP macro after the copy of the message and before the body of the explanation.
- Contains any of the macros and strings defined on the msg(7D) man page for use within explanations.
- Uses fonts and point sizes as described in Section 4.2.3, page 40.
- Ends with a .ME (message end) macro.

The following example illustrates the markup of one message and explanation pair. A symbolic message name is used in this example.

```
$msg DGR_URH The specified file is not a migrated file.
$nexp DGR_URH
The specified file is not a migrated file.
.PP
The data migration daemon received a request that applies
only to migrated files, but the requested file is not
migrated.  This error usually indicates that the file's
status has been changed by a process other than data
migration.  Perform an \*Cls -l\fR command and examine
the first character in the entry for the file.  If that
character is an "m", the file is migrated.  Inform your
system support staff.
.ME
```

4.3.4 Variables

Many error messages contain variables that contain contextual information when they are issued to the user. For example, the following messages each contain a variable that is supplied at run time:

```
A .keep file is not present for user ID 'mike'.

An attempt to allocate 512 bytes has failed.

Required option -t not specified.
```

In the first message, the user name at the end of the message is a variable. In the second message, the number of bytes is a variable. The option in the third message is a variable.

These messages would appear as follows in the message text file:

```
$msg 100 A .keep file is not present for user ID '%s'.

$msg 200 An attempt to allocate %d bytes has failed.

$msg 300 Required option -%s not specified.
```

Use single quotation marks (' ')around user-supplied strings that are referred to as tokens. Examples of such strings include file names and user IDs. The use of quotation marks highlights the literal information specific to the situation and reduces the possibility of variables being interpreted with a literal meaning. The user

ID in the first example in the preceding messages is in quotation marks because it is a token that is read from the user's environment. The quotation marks help the user to understand that the value should not be confused with standard text.

Not all variables should be enclosed in single quotation marks. Numbers do not need to be quoted (see the second example in the preceding messages). Strings that are to be interpreted literally by the user do not need to be in quotation marks. For example, the option in the third example message does not require quotation marks.

In situations where single quotation marks are used, it is necessary to precede the leading quotation mark with the `troff` string \&. This code protects the string from interpretation by `troff` as the beginning of a comment.

In the explanation of messages that contain variables, it is not possible to show the string or value that users see when they receive the message from the program. That value is unknown until the error occurs.

When you mark up the explanation, choose a word that indicates the nature of the information to be supplied at run time. Put that word in the message in place of the C language variable designator. Knowing the type of the variable will help you to choose an appropriate variable name. The most common variable designators and their variable types are listed in the following table:

Table 3. C language variable designators

Character	Type
%d or %i	Signed decimal
%f	Floating point
%o	Unsigned octal
%s	String (character pointer)
%x	Unsigned hexadecimal

Place the variable name in italic font to indicate that it does not appear literally in the message. Use the *V string to change to italic font (instead of the \fI string). The *V string improves the spacing between the Courier and the italic words.

For example, the markup of the three messages shown previously and their explanations might appear as follows:

```
$msg 100 A .keep file is not present for \&'%s'.
$nexp 100
A .keep file is not present for \*Vuser\fC.
.PP
```

```
The \*Cdmlim\fR\*(11 command did not find a file named \*C.keep\fR in
the home directory of the specified user.  To exempt files from
migration, you must create a file named \*C.keep\fR 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.
.ME
$msg 200 An attempt to allocate %d bytes has failed.
$nexp 200
An attempt to allocate \*Vnumber\fC bytes has failed.
.PP
The command was unable to allocate additional memory.  This message
indicates that your run-time memory allocation is too small to
process the command.  Your system support staff may be able to
increase your run-time memory limit.
.ME
$msg 300 Required option -%s not specified.
$nexp 300
Required option -\*Voption\fC not specified.
.PP
The \*C-t\fR and \*C-j\fR options to the \*Cdmdjournal\fR command
are required.  See the man page for a complete description of the
options to the \*Cdmdjournal\fR command.
.ME
```

4.4 Header and trailer files

Header and trailer files contain macros that are needed for the printed version of the messages, but not for the online version. Because no place exists for these macros in the message text file, they are placed in two special files. The first file is a header file that contains the macros and text that must be processed by `troff` before processing the text of the explanations. The second file is a trailer file that contains a single macro to end the 2-column formatting.

4.4.1 Header file

The header file must contain the following macros and text:

Item	Description
.MT macro	Specifies the title of the manual in which the messages are published. The manual title is printed as the inner page header.
.MN macro	Specifies the manual number of the manual in which the message are published. The manual number is printed as the inner page footer.
.CF macro	Specifies the text that you want to have appear as the center page footer.
.GC macro	Specifies the group code for your messages. The argument to this macro appears in the message ID bar as part of the message identifier.
.ST macro	Specifies the title of the section you are formatting. The section title appears in large type at the beginning of the section and as the outer page header on all succeeding pages of the section.
Intro text	Text that explains the content of the section. This text usually includes the following information: • List of the programs, commands, or routines from which the messages documented in the section are issued. • Significance of the message numbers. For example, if three commands share a catalog, a block of numbers may be assigned to each of the three commands. Thus, messages 1 through 999 are for command 1, messages 1000 through 1999 are for command 2, messages 2000 through 2999 are for command 3. If your messages numbers are divided in this or any other significant way, describe the division in the introductory text. • Sources of additional information about the product or feature.
.2S macro	Starts 2-column format. This macro must be the last macro in the header file.

The following example shows the source markup of a message header file.

```
.MN "SG\-9999"
.MT "\*u Message System Exmple Manual"
.CF "Cray Research, Inc."
.GC "dm"
.ST "\*(Cbdm\fR Messages"
This section documents all error messages issued by the data migration
```

```
feature of UNICOS.  The group code for this feature is \*Cdm\fR.
Each message is listed, along with an extended explanation.  The messages
are arranged by message number, in ascending order.
.PP
The message number helps to indicate the part of data migration that is
issuing the error.  The message numbers are assigned as follows:
.CH 12 "Range" "Message source"
.TL
1\-99
Data migration daemon
.TL
100\-199
\*Cdmget\fR(1) command
.TL
200\-499
\*Cdmput\fR(1) command
.TL
500\-999
\*Cdmlim\fR(1) command
.TL
1000\-1299
\*Cdmdalter\fR(8) command
.TL
1300\-1499
\*Cdmdjournal\fR(8) command
.TL
1500\-1599
\*Cdmdstat\fR(8) command
\&.TL
1600\-
Any data migration command
.PP
The explanation that accompanies each message describes the error in greater
detail and suggests actions for solving the problem.  The explanation may
refer you to documentation that discusses topics related to either the
problem or the solution.  You can also refer to \fIUNICOS System Administration\fR,
publication SG\-2113, for a description of the data migration feature, its
configuration, and its administration.
.2S
```

4.4.2 Trailer file

The trailer file must contain one 2E macro to end 2-column format.

4.5 Extraction and printing

When you have a properly marked-up message text file, you are ready to extract the explanations from it and print the resulting message section.

The following subsections provide detailed steps that you must perform to produce a hard-copy message section.

4.5.1 Extracting the explanations

The message text file contains messages and explanations. Because the documentation that is being produced includes only the explanations, a process is required for extracting the explanations from the message text file. Use the `catxt`(1) utility to perform this extraction.

The syntax of the `catxt` utility is as follows:

```
catxt -n outfile [-s[-P cpp_opts]] infile
```

For example, if you have a marked-up message text file called `dm.msg`, you could extract the explanations from that file and put them in a file called `dm.nexp`, using the following command:

```
catxt -n dm.nexp dm.msg
```

It is a convention to use the suffix `nexp` for explanation files.

If the message text file, `dm.msg` in this example, contains symbolic message names, use the `-s` option as shown in the following command to call `cpp`(1) to map those names to numbers.

```
catxt -n dm.nexp -s dm.msg
```

For more information about using `catxt`, see the `catxt` man page.

In addition to extracting the explanations, the `catxt` utility also replaces the `$nexp` tag with a `.MS` macro (message start). While it is performing this replacement, `catxt` checks that every `.MS` macro has a corresponding `.ME` macro. This pairing is required to ensure proper printing.

If `catxt` finds a missing `.ME` macro, it issues a warning message. The following is an example of the warning message:

```
***WARNING: nexp number 38 does not
have an ending ".ME"***
```

If you receive this message, add a .ME macro to the indicated message in the input file and rerun catxt. Do not make the correction in the output file. All corrections must be made to the message text file so that they are propagated to the online version of the messages, as well as to the hard copy.

4.5.2 Printing the explanations

When you have created the following files, you are ready to print your hard-copy message section:

- Header file (see Section 4.4.1, page 45, for the recommended content of this file)
- Explanation file (see Section 4.5.1, page 48, for a discussion of the utility to create this file)
- Trailer file (see Section 4.4.2, page 47, for the recommended content of this file)

Use the troff(1) text formatting utility to print these files as a hard-copy message section. The troff utility on Cray Research systems is a device-independent text processor (ditroff) that produces output suitable for a PostScript laser printer.

Use the -msg option of the troff utility to use the message macro definitions (see msg(7D)) during text formatting. The troff output can be printed using the lpr(1B) utility with the -n option. The -n option identifies ditroff as the source of the input to lpr.

For example, to print the data migration messages from the dm.head header file, dm.nexp explanation file, and dm.trail trailer file, use the following command:

```
troff -msg dm.head dm.nexp dm.trail | lpr -n
```

Appendix B, page 65, contains an example of the data migration message section that is printed as the result of these commands.

4.6 Testing online explanations

To test that a message text file produces a working explanation catalog, perform the following steps:

1. Build the explanation catalog from the message text file by using the -c and -e options of the caterr(1) utility. (In addition, use the -s option if your message text file uses symbolic message names as described in Section 2.2.1.1, page 7.)

2. Set your NLSPATH environment variable to point to the directory that contains the explanation catalog that you created in the first step.
3. View the explanations in the catalog by using the explain(1) utility.

The following subsections describe these three steps in greater detail.

4.6.1 Building the explanation catalog

The online explanations that a user sees are drawn from a file called the *explanation catalog*, which is a binary file built from the message text file. The explanation catalog is in a form that can be read by the explain(1) utility, which users use to retrieve an online explanation.

The caterr utility builds the explanation catalog from the message text file. Use the -e and -c options of caterr to build an explanation catalog. The -e option specifies that you are building an explanation catalog and not a message catalog. The -c option lets you specify the name of the explanation catalog to be built.

In addition, use the -s option if the message text file contains symbolic message names. The -s option calls the C preprocessor (cpp(1)). The cpp utility, using the include file referenced in the message text file, replaces the symbolic names in the file with the appropriate numbers. (For more information on using symbolic names, see Section 2.2.1.1, page 7.)

The syntax of the caterr utility with these options is as follows:

```
caterr [-s] -e -c catfile infile
```

The *catfile* argument specifies the name of the catalog to be output, and *infile* specifies the name of the message text file to be read as input.

The following example builds an explanation catalog named dm.exp from the message text file dm.msg:

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

The following example builds an explanation catalog named dm.exp from the message text file dm.msg, which contains symbolic message names:

```
caterr -s -e -c dm.exp dm.msg
```

4.6.2 Setting the NLSPATH variable

After you have created an explanation catalog, you must tell UNICOS the location of the catalog by setting the NLSPATH environment variable. This variable gives the search path that the explain utility (used in the next step) uses to find an explanation catalog.

The NLSPATH environment variable must be of a very specific format to work correctly. The file name (last part of the path name after the final slash) must be (literally) %N.cat. (When UNICOS parses this string, the %N is replaced by the group code of the message.)

For example, to test explanations in a catalog named dm.exp in the /home/messages/dm directory, set the NLSPATH variable to the following value:

```
/home/messages/dm/%N.cat
```

In the standard shell, the following statements set NLSPATH to include this string and export the value of NLSPATH:

```
NLSPATH=/home/messages/dm/%N.cat
export NLSPATH
```

In the C shell, the following statement sets NLSPATH to include this string:

```
setenv NLSPATH /home/messages/dm/%N.cat
```

The explain utility reads the NLSPATH variable to determine the name of the message catalog for the file (of the form *group*.cat). It then substitutes the .exp suffix for the .cat suffix in the message catalog name to determine the explanation catalog name. The utility tries to open the file to retrieve the explanation.

4.6.3 Viewing the explanations

Verify that the explanations are displayed correctly for the user by viewing the explanations through the explain utility. Issue the explain utility once for each explanation you want to view.

For example, to view the explanation for the message with the ID dm-100, issue one of the following commands:

```
explain dm100

explain dm-100
```

The explain utility retrieves the explanation and displays it. The following example illustrates how such a session might appear on your screen.

```
% 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.
```

Issue successive `explain` commands until you have tested at least a representative sample of the explanations in the file. Always test the first explanation and the last explanation.

Check the output for correct line breaks, highlighting and underlining (if your terminal is enabled to display them), and completeness of the text (no text missing).

If you find a problem with the catalog, return to the message text file, edit it to eliminate the problem, regenerate the explanation catalog, and test the explanation again. Remember that any change to the message text file is propagated to both the online explanation catalog and the hard-copy message section. After you make a change, check both outputs. Repeat this process until you are satisfied that the message text file is producing usable online text and hard copy.

4.7 Troubleshooting

Table 4 lists common problems that you might encounter when working with the message macros and procedures. The cause of these problems is identified.

Table 4. Common formatting problems and their solutions

Problem	Cause
Output is formatted incorrectly; only one word appears on each line.	Verify that there is a `.2S` macro in the *group*`.head` file. The absence of this macro will cause this problem.
Only part of the copy of the message that appears in the explanation is in Courier font; the remainder is in New Century Schoolbook.	You have used a `\fR` font instead of a `\fC` font after changing fonts for a variable in the copy of the message. Change the `\fR` font code to `\fC`.
The message font changes from Courier to New Century Schoolbook, a blank line is inserted, and the point size gets smaller, even though no font code appears in the source file.	The copy of the message exceeds one logical line. Either join the lines into one or use a continuation character (\) at the end of each physical line except the last one.
A message or explanation is truncated where a variable occurs in the text. Some portion of the source text is missing in the output.	Check for an unprotected single quotation (`'`). Single quotation marks that appear after a space must be preceded by the characters backslash ampersand (\&) to protect them from interpretation by `troff` as the beginning of a comment.
No header bar is appearing on the first page of the section and many messages are being continued over columns and pages.	Either you do not have a header file or your header file does not contain a `.ST` macro. Create a header file and include a `.ST` macro or add a `.ST` macro to your existing header file.

Guidelines for Messages and Explanations [A]

A message consists of two parts: the message that is printed for the user each time the error occurs, and an expanded explanation of the error condition that appears in the message documentation, both online and in the printed manual.

The message and the explanation should both be clear, concise, and focused; however, the message and the explanation may speak to different audiences.

The message is directed to any user who might encounter the error that the message describes. It should contain a brief description of the problem in unambiguous terms.

The explanation is directed to the user who cannot resolve the error using only the information in the message. This user has sought additional information. The explanation should give a more complete description of the problem, suggest actions that will help resolve the problem, and direct the user to sources of information related to the problem and its resolution.

The following subsections give guidelines for writing messages and for writing message explanations.

> **Note:** These guidelines are intended for Cray Research, Inc. software developers who are writing messages for code included in Cray Research, Inc. software releases. Others may want to follow the guidelines to improve the general usability of their messages and explanations.

A.1 Guidelines for messages

A good message provides a user with specific information about the problem that the software has encountered. It conveys the context in which the problem occurred and, when possible, states the problem in a way that implicitly suggests a corrective action. A good message also is written with an awareness of the attitude that is expressed toward the user.

The challenge of writing good messages is conveying as much information as possible concisely. A good rule of thumb for messages is that users who are past the initial learning phase for a product should be able to recognize and correct the problem by using only the information in the message. The explanation exists for users who are new to the product.

Good messages demonstrate the following characteristics:

- Clearly stated

- Specific about the problem
- Respectful of users
- Grammatically correct

These characteristics are important for the usability of the messages in English, and they also improve translatability.

A.1.1 Clear messages

Clear messages state the problem as simply as possible without the use of specialized terminology. A clear message is unambiguous in its description of the problem.

Observe the following guidelines to make messages clear:

1. When describing a problem, use plain English instead of terms familiar only to a limited audience (for example, UNIX system terminology).

   For example, the following message is clear to a programmer familiar with the `stat`(2) system call, but it is not clear to most users:

   ```
   Cannot stat file
   ```

   The following message is preferred:

   ```
   Cannot get the status (with stat(2)) of
   the 'filename' file.
   ```

2. Choose message syntax carefully. Avoid long strings of modifiers.

   For example, the following message:

   ```
   bad swap superblock magic number
   ```

   could be worded more clearly as follows:

   ```
   The checkword number in the swap superblock
   is incorrect.
   ```

3. Use worded explanations rather than programming-language expressions.

   For example, the following message:

   ```
   end bp forw != NULL
   ```

   would be clearer if rewritten as follows:

   ```
   The I/O chain for the pty/tty device has
   failed an internal consistency check.
   ```

Follow these principles wherever possible. Remember that users do not know as much about the system, the program, or the origin of the error as you do.

A.1.2 Specific messages

Messages that are specific give users all of the information needed to correct the problem. Observe the following guidelines to make messages specific:

1. Identify the problem specifically, rather than in a general sense.

   For example, the following message is very vague:

   ```
   I/O error
   ```

   Instead, give information that is definite enough to point to a corrective action.

2. Explain the problem from the user's perspective rather than the system's perspective.

   For example, the following message:

   ```
   No device response
   ```

   would be better stated as follows:

   ```
   Cannot access the device you selected.
   ```

3. Include information specific to the situation.

   Instead of the following message:

   ```
   Terminating job
   ```

   it would be better to write:

   ```
   Terminating job 'job-identifier'.
   ```

4. Include information pertinent to the solution of the problem; do not force users to guess arbitrary limits.

   The following message:

   ```
   Identifier too long
   ```

   would be better stated as follows:

   ```
   The identifier must consist of 14 characters
   or less.
   ```

A.1.3 Respectful messages

The message should respect users and the situation. Respect is shown by adhering to guidelines, as follows.

1. State the problem neutrally or as a deficiency of the system rather than blaming the user for the problem.

   For example, the following message:

   ```
   Illegal expression 'exp' has been specified
   in the input file
   ```

   could be stated more neutrally as follows:

   ```
   Cannot accept the expression 'exp' in the
   input file.
   ```

2. Avoid unnecessarily hostile, violent, or threatening terminology. Terms such as *catastrophic*, *abort*, *illegal*, *kill*, *abandon*, and *disastrous* have a negative effect on users. Rephrase messages containing such words to be more neutral and less threatening.

   It can be difficult to follow this guideline in situations in which accepted UNIX terminology requires the use of a "hostile" word to provide an accurate technical description of the situation. For example, UNIX uses the term *kill* in a technical sense to describe forced job termination. In such a situation, it may not be possible to avoid the term. However, whenever it is within your control, avoid overly dramatic terminology.

3. Avoid introducing attempts at humor.

   Humor in messages has many dangers. Everyone's idea of what is funny differs, especially across cultures. Messages often occur repeatedly, and the humor wears off quickly. Therefore, the best policy is to avoid making messages humorous or cute.

A.1.4 Grammatical messages

A grammatical message is phrased as a complete sentence, is punctuated according to standard usage, contains no truncated words, uses conventional spelling, and contains articles, auxiliary verbs, and prepositions as dictated by standard English usage. Messages written in standard English are less likely to be misinterpreted and can be translated more accurately into foreign languages.

Observe the following guidelines to make messages grammatical:

1. Use capitalization in a standard way; start the message with a capital letter, and capitalize words and abbreviations as they would appear in narrative text.
2. Avoid beginning messages with a special character or a variable. Special characters and variables at the beginning of messages make them difficult to index.
3. Use punctuation in a standard way; include commas and semicolons when appropriate, and end the message with a period.
4. Observe Cray Research trademark and style conventions when using industry terms. Consult a writer if you have questions regarding Cray Research style conventions.
5. Spell out words completely.

   The space and time saved by writing `max` instead of `maximum` is not worth the lack of clarity it creates in the messages. Use only abbreviations that are very widely understood by users of Cray Research systems. For example, it is sensible to use IOS and OWS in messages instead of I/O subsystem and operator workstation. However, it is inappropriate to use MTU for maximum transmission unit in a message whose audience is the end user.
6. Write messages as complete sentences; include a verb and all the needed articles (a, an, the), prepositions, and auxiliary verbs.

The following example illustrates the intention of these guidelines.

The following message:

```
read error
```

would be more grammatical if phrased as follows:

```
An error occurred during an attempt to read
the 'filename' file.
```

The rewritten message is longer, but it is much less likely to be misunderstood or mistranslated. It is more specific, in addition to being more grammatically correct.

A.1.5 Severity levels in messages

Each message issued to users should have a severity level associated with it. The severity level should indicate to users how important the message is to the success of the job. To help users assess the impact an error has on a job, it is important that you use severity level designations in a consistent manner when you develop software.

The following guidelines apply to message severity levels:

- Indicate the message severity level in uppercase letters (for example, WARNING). Use of all uppercase letters calls attention to the severity level.
- Avoid the use of ERROR as a severity level. Messages that users receive are commonly called *error messages*. Therefore, to designate ERROR as a severity level is uninformative and creates an ambiguity in phrases such as *the error message manual*.
- If it does not conflict with third-party vendor constraints on your code, try to restrict your use of severity levels to the following set:

Level	Description
INFO	The system is communicating information to users, usually about the status of a job or process. An informational message requires no user action because a problem was not encountered.
EFFICIENCY	An inefficient use of the software or the hardware is suspected. Users should examine the code for a better way to perform the process.
CAUTION	A possible problem was detected. The output of the program is still usable, but the results may not be what users expect.
WARNING	A probable problem was encountered. The program continues to process from this point, but the output or the results are likely to be incorrect.
FATAL	A definite problem was encountered. The output produced from this point forward is unusable. Output may be suppressed after this point. Use this level of message when a fatal error is encountered in the input, rather than in the processing. If processing encounters an error that is terminal, issue an unrecoverable error. For example, when a compiler encounters an error in the program source it is compiling that renders further execution of the program useless, issue a fatal error message. But, if the compiler program itself encounters a situation in which it can no longer execute, perhaps because of hardware or system software problems, issue an unrecoverable error. Avoid the use of FATAL as a severity level when possible because, although it is standard in some contexts, *fatal* is an inappropriate word to use as a severity level because it is threatening and overused.

UNRECOVERABLE — An error has occurred that renders further processing impossible. The program terminates immediately.

You should issue this level of message only once and terminate processing immediately. See the description of FATAL for an example of the difference between fatal and unrecoverable errors.

These guidelines and suggested severity levels may not apply to all situations. The most important point to remember is that users rely on the message severity to indicate the nature of the problem. Be as consistent and as accurate as possible with this information.

A.1.6 Substitutable strings in messages

Care should be taken to limit the type of information substituted into messages. Only information that users supply should be substituted into error messages. Examples of user-supplied information include variable names, command-line options, and file names. Building messages from other types of substitutable strings can be a serious impediment to a correct translation.

Consider the following message:

Example 1:

```
Cannot find the file 'filename'.
```

You also may want to apply this message to a situation in which two file names are supplied by the user and neither is found. To cover this situation, you should create a second message that is issued when two files are involved in the error.

The second message might appear as follows:

Example 2:

```
Cannot find the files 'filename' and 'filename'.
```

An alternative in this situation would be to modify the first message to accommodate errors on both one or two files. However, doing so would create a potential for translation errors.

For example, consider the case where you change the message in example 1 to read as shown in example 3 in the message catalog:

Example 3:

```
Cannot find the file%s '%s' %s '%s'.
```

Then you replace the first string with `s` to make the word `file` plural, replace the second string with a file name, replace the third string with `and`, and replace the fourth string with another file name.

The result would be a message that would appear to users to be identical to example 2. However, the translator cannot determine which of the replaceable strings is really a user variable and which is part of the message text. Also, the pluralization of `file` by adding a trailing `s` is correct in English, but it would not be correct in most other languages. The insertion of the connective `and` between the file names might also be incorrect in the target language.

Because of the complexities involved in writing for translation, avoid writing messages to appear in multiple contexts. If you must issue a message that is a variation on the syntax of a similar message, write a new message to cover the variation, rather than try to adjust the first message to accommodate all cases.

A.2 Guidelines for explanations

Message explanations exist for the benefit of users who, upon receiving an error message, cannot resolve the problem without the following additional information:

- A more complete description of the problem
- A suggested course of action to solve the problem

A good explanation contains both types of information. The most natural way to format the information is to describe the problem in the first paragraph of the explanation and recommend solutions in the second paragraph. In some cases, more information will be given than comfortably fits in a paragraph. Add additional paragraph breaks as needed.

The following subsections discuss describing problems and solutions to users in message explanations.

A.2.1 Describing the problem

The message explanation provides a more complete description of the problem than the message itself does. Include the following information in the message description:

- Statement about the cause of the problem
- Context surrounding the cause of the problem
- System or job status affected by the problem

- References to documentation discussing related topics

The following message description states the cause of the problem clearly and outlines the status of the job:

```
The catalog specified for reset was not
defined with the RECOVERABLE attribute.
RESETCAT can reset only recoverable
catalogs. The command is terminated. The
catalog and CRA entries have not been
altered. The workfile has not been defined.
```

The description also could refer users to documentation on the RESETCAT command.

A.2.2 Describing the solution

The solution portion of the message explanation presents courses of action that users can pursue in solving the problem. Many messages result from complex or unknown causes. In these cases, users may have to test several conditions or try several solutions before arriving at one that applies to the problem. Most users realize that this is a fact of life. They do not expect a cure-all to be provided in the message explanation. Rather, they are looking for somewhere to start to solve the problem.

Include the following information in the message solution:

- Suggested problem remedies, listed in the order in which users should try them.
- Parameters, files, permissions, and other configuration information that might be related to the problem. Instruct users to check these items.
- Any steps needed to recover from the problem; for example, if the problem requires that a component be restarted after the problem is located, be sure to instruct the user to perform the restart.
- Possible or likely consequences of various courses of action, especially if those consequences are destructive. For example, if a suggested action might damage or destroy data, you must point that out to users. Do not assume that they know.
- References to documentation that discusses utilities, procedures, or configuration information needed to solve the problem.
- Recommendation to seek help if the problem is not a user-level error. In these cases, direct users to contact the system support staff and state the purpose of the contact. The following phrase can be used in this situation:

```
If none of the suggested actions resolve
the error condition, contact your system
```

```
support staff and request that
fill in an action the support staff should perform .
```

The following paragraph is an example of the solution portion of a message explanation:

```
To recover a nonrecoverable catalog and its
volumes, you must do a synchronized volume
restore of all volumes owned by the catalog.
If you have incorrectly specified the CATALOG
parameter, correct the parameter. If CATALOG
dname was specified, correct the associated
DLBL catalog name. Rerun the command.
```

As with the messages themselves, make the explanations as clear and specific as possible. Try to create a course of action for users that leads to the resolution of the problem.

Message Section Example [B]

This appendix contains a complete example of a correctly formatted message system section. This example is a portion of the documentation for the `dm` group code for the UNICOS 6.1 release.

Glossary

catalog

The binary form of the message or explanation file. There are two kinds of catalogs: message catalogs and explanation catalogs. The `gencat`(1) command produces a catalog from the output of the `caterr`(1) command. The `-c` option of `caterr` calls `gencat` and generates a message or explanation catalog from a message text file in one step.

explanation catalog

A binary file, produced by the `gencat`(1) command, that contains the text of error message explanations. These explanations are accessed and displayed by the user with the `explain`(1) command.

group code

The name given to the catalog of messages for a product. The group code is a shorthand way to refer to the software products that share one message file. The group code should consist of 2 to 6 alphanumeric characters; a maximum of 10 characters are allowed. The group codes `Local`, `local`, and `LOCAL` and all group codes beginning with `Z` (uppercase only) are reserved for site use.

message catalog

A binary file produced by the `gencat`(1) command that contains the text of error messages as they are called from the software at run time.

message text file

The file that contains the source form of the messages and explanations. A message text file can contain messages, formatted and unformatted explanations, and comments.