Cray® Message System Programmer's Guide

SG–2121 6.5

Document Number 007–3548–002

New Features

Cray® Message System Programmer's Guide SG–2121 6.5

This revision of the Cray Message System Programmer's Guide supports the IRIX 6.5 software release. It incorporates the following changes:

- The name of the message system has been changed from "UNICOS Message System" to "Cray Message System" in order to reflect its use with the IRIX operating system.
- Printing formatted messages in manuals is no longer supported, and text on this subject has been removed.
- On IRIX systems, the default `NLSPATH`, found in `nl_types.h`, differs from the default UNICOS and UNICOS/mk `NLSPATH`. This may cause problems for packages that do not set the `NLSPATH` using modules and assume the default `NLSPATH`.
- On IRIX systems, assuming the system language is C, if the user has not set `NLSPATH`, and neither `LANG` nor `LC_MESSAGES` is set, and the catalog name `lib.cat` is passed to the `catopen`(3) utility, `catopen`(3) tries to open a catalog using the paths defined by the `DEF_NLSPATH` macro in file `nl_types.h`.
- On IRIX systems, most message and explanation catalogs are located in the `/usr/lib/locale/C/LC_MESSAGES` directory.
- On IRIX systems, message macros for formatting explanation text are defined in the `/usr/share/lib/tmac/tmac.sg` file.
- On IRIX systems, other language, territory, and code set designations are defined and supported in six localized desktops and over 30 basic locales.
- On IRIX systems, catalogs for other languages are installed in the `/usr/lib/locale/LANG/LC_MESSAGES` directory tree.
- On IRIX systems, users control the language in which they receive messages by setting the `LANG` environment variable or the `LC_MESSAGES` locale category.

Record of Revision

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.
8.3	January 1995 Revision to reflect message system features added in the UNICOS 8.3 release for X/Open compliance. This revision is only distributed online through Docview.
9.0	July 1995 Reprint with revision to reflect message system features added since the last reprint.
6.5	May 1998 Revision to support the IRIX operating system 6.5 release.

Contents

Page

Page

Figures

Tables

About This Guide

This publication documents the Cray message system, which runs on the UNICOS, UNICOS/mk, and IRIX operating systems. This publication discusses the Cray 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 Cray message system.

Related Publications

The following documents contain additional information that may be helpful:

- 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 UNICOS system, and for information about using the message system to retrieve message explanations, see *General UNICOS System Administration*, publication SG–2301.
- 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 UNICOS/mk system, and for information about using the message system to retrieve message explanations, see *UNICOS/mk General Administration*, publication SG–2601.

Ordering Publications

Silicon Graphics maintains publications information on the IRIX operating system and related products at the following URL:

`http://techpubs.sgi.com/library`

The preceding website contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

Cray Research maintains publications information on the UNICOS and UNICOS/mk operating systems and related products at the following URL:

`http://www.cray.com/swpubs`

Also, the *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a Cray Research or Silicon Graphics document, 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.

Silicon Graphics employees may send their orders via electronic mail (using a UNIX system) to `orderdsk`.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
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 or directive line.
...	Ellipses indicate that a preceding element can be repeated.

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

Term	Definition
UNICOS systems	All configurations of Cray parallel vector processing (PVP) systems that support the Cray message system
UNICOS/mk systems	All configurations of Cray massively parallel processing (MPP) systems that support the Cray message system
IRIX systems	All configurations of CHALLENGE, CHALLANGE 10000, CRAY, CRAY Origin2000, Indigo2 10000, Indigo R4000, Indigo2, Indy, O2, Octane, Onyx, Onyx Extreme, Onyx2, Origin 200, Origin 2000, POWER CHALLENGE, POWER CHALLANGE 10000, POWER Indigo2, and POWER Onyx systems that support this release

The default shell in the UNICOS and UNICOS/mk operating systems, 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 Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Cray UNICOS Version 10.0 is an X/Open Base 95 branded product.

Reader Comments

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

- Send us electronic mail at the following address:

  `techpubs@sgi.com`

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, 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 Customer Service Call 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.

We value your comments and will respond to them promptly.

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.

Silicon Graphics/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 Cray 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 Cray 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 or UNICOS/mk environment or in a Silicon Graphics IRIX 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 Cray message system.

Warning: Sites using the Cray ML-Safe configuration of the UNICOS or UNICOS/mk operating systems or the Trusted IRIX operating system can use the information and procedures outlined in the following sections 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, which contain the text of the messages issued at run time

- *explanation catalogs*, located in the same directory as the message catalogs, which 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 chapter 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 31, provides a sample procedure for converting an existing piece of software to use the message system.

Appendix A, page 41, is an appendix that lists guidelines for writing effective messages and usable message explanations.

Message System Design [2]

The Cray 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 chapter 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 31. That chapter 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))
- Catalog search path utility (`whichcat`(1)) (UNICOS and UNICOS/mk systems only)
- 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.

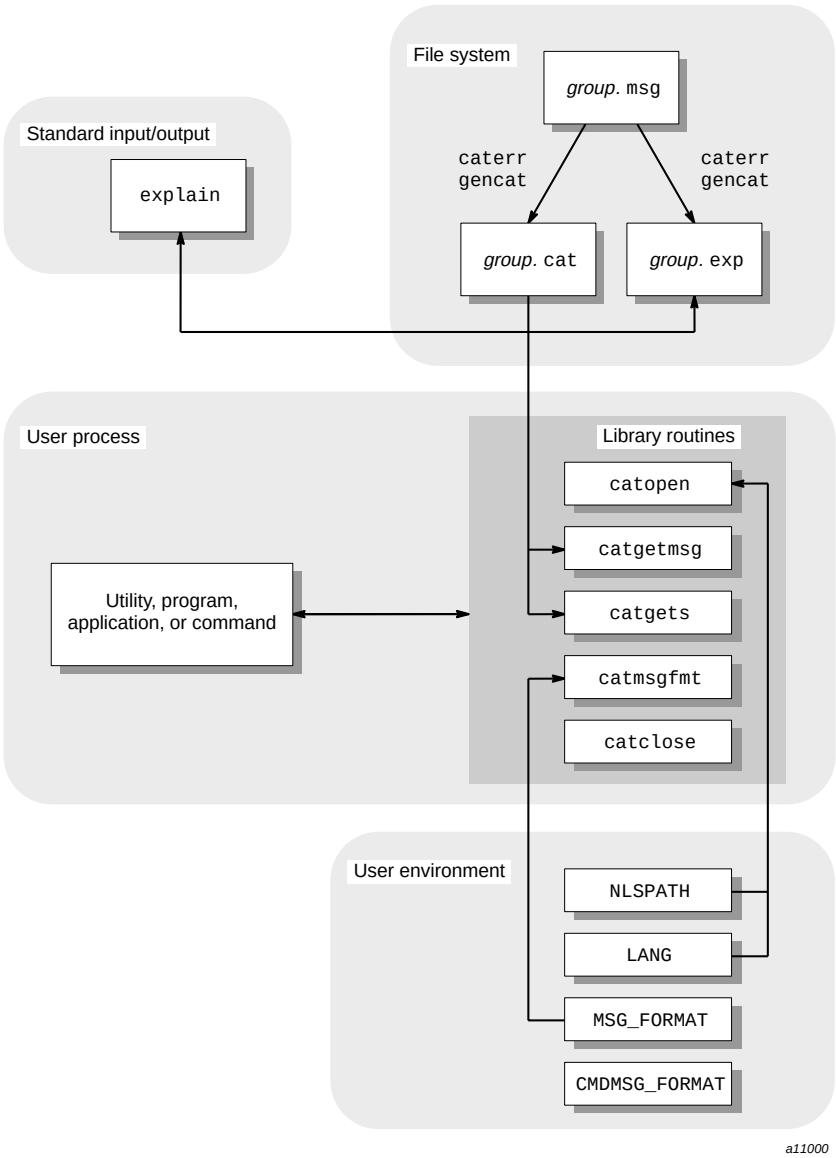

Figure 1. Message System Overview

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 (on UNICOS and UNICOS/mk systems) or a `make`(1) or `smake`(1) makefile (on IRIX systems). On UNICOS and UNICOS/mk systems, catalogs are installed in the `/lib` or `/usr/lib` directory trees. On IRIX systems, catalogs are installed in the `/usr/lib/locale/`*LANG*`/LC_MESSAGES` directory tree. (*LANG* represents either the `LANG` environment variable or the `LC_MESSAGES` category, both of which the `catopen`(3C) utility uses to locate the correct message catalog.)

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. On UNICOS and UNICOS/mk systems, 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 category.

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 sections 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.

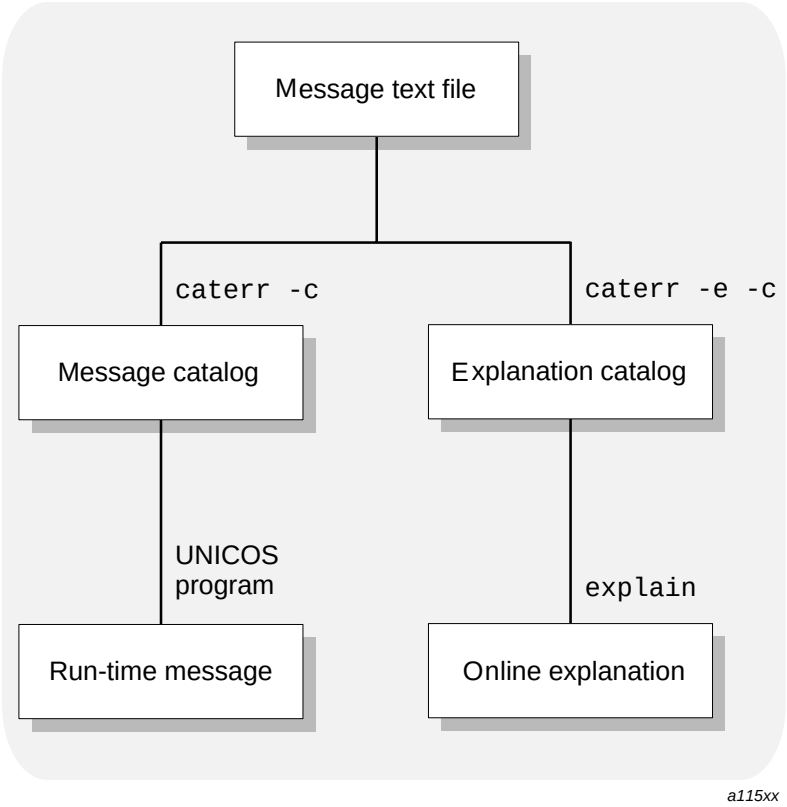

Figure 2. Processing the Message Text File

The message text file should be named as follows:

group`.msg`

This name is required to satisfy rules for catalog names and implicit rules in `nmake`(1) (UNICOS and UNICOS/mk systems). *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 supplied by Cray Research 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 sections 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 sections 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 20.)

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) (UNICOS and UNICOS/mk systems) or `printf`(3S) (IRIX systems) format codes. For example, format codes such as `%s`, `%d`, and `%f`) could be used 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 34.

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 that exceeds one line in the message text file:

```
$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 back slash (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 25.

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 Silicon Graphics/Cray Research software development (as opposed to on site) are formatted using the `nroff` message macros contained in the `tmac.sg` file (see the `msg`(7D) man page or the `msg`(5) reference page). 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 format the explanations.

The following sections 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 on UNICOS and UNICOS/mk systems and described on the `msg`(7D) man page. On IRIX systems, message macros are defined in the `/usr/share/lib/tmac/tmac.sg` file and described on the `msg`(5) reference page.) The message macros are collections of `nroff`(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
```

2.2.2.2 Unformatted Explanation Text

Sites that elect not to format 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 11, 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 12 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 20.) To produce a message catalog and an explanation catalog from one message text file, you must run `caterr` twice.

The following sections 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 Cray 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 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*]]

On UNICOS and UNICOS/mk systems, the string En is the designation for the American English language. On IRIX systems, the string en_US is the designation for the American English language.

On UNICOS and UNICOS/mk systems, other language, territory, and code set designations (if any) are defined and supported locally. IRIX systems support six localized desktops and over 30 basic locales produced by Silicon Graphics.

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

2.3.1.2 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`.

On UNICOS and UNICOS/mk systems, 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`, the `explain` utility would search for the explanation catalog named `/usr/tmp/lib.exp`. On IRIX systems, the `explain` utility will properly display explanations even if the `NLSPATH` definition does not append the `.cat` suffix.

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.

(UNICOS and UNICOS/mk systems only)

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

(IRIX systems only)

```
/usr/lib/locale/%L/LC_MESSAGES/%N:\
/usr/lib/locale/%L/Xopen/LC_MESSAGES/%N:\
/usr/lib/locale/%L/LC_MESSAGES/%N.cat:\
/usr/lib/locale/C/LC_MESSAGES/$N:\
/usr/lib/locale/C/LC_MESSAGES/%N.cat
```

Most message and explanation catalogs are located on disk in the `/usr/lib/nls/En` directory (for UNICOS and UNICOS/mk systems) or in the `/usr/lib/locale/C/LC_MESSAGES` directory (for IRIX systems). 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 (UNICOS and UNICOS/mk systems). 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. On IRIX systems, if the `LANG` or `LC_MESSAGES` language designation variable is set to an unsupported value, the catalog from `C` locale will be used, and the message will be displayed in American English.

On UNICOS and UNICOS/mk systems, 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. There is no parallel utility on IRIX systems.

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 for UNICOS and UNICOS/mk `nmake`(1) implicit rules. (`nmake`(1) is not available on IRIX systems.)

The catopen(3) function references the NLSPATH environment variable when determining the message catalog to open. For example, on UNICOS and UNICOS/mk systems, if the user has not set NLSPATH, and neither LANG nor LC_MESSAGES is set, and the catalog name lib is passed to catopen, catopen tries to open the catalog /usr/lib/nls//lib.cat. If this catalog does not exist, catopen tries the next catalog, in this order:

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

On IRIX systems, assuming the system language is C, if the user has not set NLSPATH, and neither LANG nor LC_MESSAGES is set, and the catalog name lib.cat is passed to the catopen utility, catopen tries to open a catalog using the paths defined by the DEF_NLSPATH macro in file nl_types.h, in this order:

```
/usr/lib/locale/C/LC_MESSAGES/lib.cat
/usr/lib/locale/C/Xopen/LC_MESSAGES/lib.cat
/usr/lib/locale/C/LC_MESSAGES/lib.cat.cat
/usr/lib/locale/C/LC_MESSAGES/lib.cat
/usr/lib/locale/C/LC_MESSAGES/lib.cat.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 23.

The explain(1) user utility references the NLSPATH environment variable when determining what 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 on UNICOS and UNICOS/mk systems:

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

On IRIX systems, if `NL_CAT_LOCALE` is passed to `catopen`(3C) and `LC_MESSAGES` equals `C`, `explain`(1) searches for the following catalogs in succession:

```
/usr/lib/locale/C/LC_MESSAGES/lib.exp
/usr/lib/locale/C/Xopen/LC_MESSAGES/lib.exp
/usr/lib/locale/C/LC_MESSAGES/lib.exp.cat
```

Otherwise, `explain`(1) searches for the following catalogs in succession:

```
/usr/lib/locale/%L/LC_MESSAGES/lib.exp
/usr/lib/locale/%L/Xopen/LC_MESSAGES/lib.exp
/usr/lib/locale/%L/LC_MESSAGES/lib.exp.cat
/usr/lib/locale/C/LC_MESSAGES/lib.exp
/usr/lib/locale/C/LC_MESSAGES/lib.exp.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 `caterr` calls. 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 on UNICOS and UNICOS/mk systems, use the `catopen`(3), `catclose`(3), `catgetmsg`(3), and `catgets`(3) library functions. To access the message catalog from your program on IRIX systems, use the `catopen`(3C), `catclose`(3C), `catgetmsg`(3C), and `catgets`(3C) 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`.

On UNICOS and UNICOS/mk systems, 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 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`.

On IRIX systems, the performance difference is minimal between `catgets` and `catgetmsg` on relatively short messages, which are the most common. `catgets` may be slightly faster than `catgetmsg` because it does not need to copy the message into a supplied message buffer.

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 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 fewer than two arguments. (UNICOS and UNICOS/mk systems)
NL_ERR_ARGNULL	The *name* argument to catopen is NULL. (UNICOS and UNICOS/mk systems)
NL_ERR_HEADER	catopen was not able to validate the message catalog file header as a valid message catalog file. (UNICOS and UNICOS/mk systems)
NL_ERR_MALLOC	catopen was not able to allocate memory (using malloc(3)) for internal structures.
NL_ERR_MAP	The requested catalog file could not be memory mapped (see the mmap(2) man page). (IRIX systems)
NL_ERR_MAXOPEN	The maximum number of per process open message catalogs had been exceeded. See NL_MAX_OPENED in the nl_types.h file. (IRIX systems)
NL_ERR_VERSION	catopen found an invalid version number in the message catalog file header. (UNICOS and UNICOS/mk systems)

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:

- (UNICOS and UNICOS/mk systems) catopen was not able to successfully open (using open(2)) any of the message catalog files specified in the NLSPATH environment variable search path.
- (UNICOS and UNICOS/mk systems) catopen was not able 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.
- (IRIX systems) catopen was not able to successfully open (using open(2)) any of the message catalog files specified in the NLSPATH environment variable search path.
- (IRIX systems) catopen was not able to successfully read from (using read(2)) the message catalog file header and set directory.

- (IRIX systems) `catopen` could not obtain file status information (using `fstat`(2)) for the message catalog.

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, suppose you have the following message, which contains replaceable parameters:

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

The message 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) on UNICOS and UNICOS/mk systems or `printf`(3S) on IRIX systems).

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 sections 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.

On UNICOS and UNICOS/mk systems, 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.

On IRIX systems there is no `sys` catalog. To retrieve system error messages, use the `strerror`(3c) function.

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 23, 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 Cray message system requires changes to the way messages have traditionally been coded, tested, and documented in most organizations. This chapter 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 section 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:

- Where are the error messages located? Are they contained within one error-processing routine or are they dispersed throughout the program?
- 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.
- 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. It is recommended 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 many of the group codes in use for the UNICOS, UNICOS/mk, and IRIX operating systems.

- 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 UNICOS `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, UNICOS/mk, and IRIX 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, suppose 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 41. 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, suppose 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 in UNICOS and UNICOS/mk Systems

If your code is part of the UNICOS or UNICOS/mk system, your completed code and message catalogs must be integrated and built into the UNICOS or UNICOS/mk release. The following sections 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. (USM also works for the UNICOS/mk system.) 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 `nmake` makefile:

   ```
   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 sections 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.

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 sections give guidelines for writing messages and for writing message explanations.

> **Note:** These guidelines are intended for Silicon Graphics/Cray Research software developers who are writing messages for code included in UNICOS, UNICOS/mk, and IRIX 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 Silicon Graphics trademark and style conventions when using industry terms. Consult a writer or editor if you have questions regarding 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 Silicon Graphics/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 sections 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.

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 the catalog from the output of the `caterr`(1) command. The `-c` option of `caterr` calls `gencat`(1) and generates a message or explanation catalog from a message text file in a single step.

explanation catalog

A binary file, produced by the `gencat`(1) command, that contains the text of UNICOS, UNICOS/mk, or IRIX error message explanations. The user accesses and displays these explanations by using the `explain`(1) command. For more information, see also the `explain`(1) man page.

group code

The name given to the catalog of messages for a product; it 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 that begin 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.

Index

N

O

P

R

S