Using the Message System [3]

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

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

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

3.1 Planning a conversion

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

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

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

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

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

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

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

3.2 Building a message text file

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

Use the following steps to build the message text file:

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

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

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

   For example, if your code contains the following message:

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

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

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

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

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

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

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

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

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

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

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

   set NLSPATH to the following value:

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

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

3.3 Modifying the program source

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

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

   The include line appears in the program as follows:

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

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

   ```
   nl_catd  mcfd;
   ```

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

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

   The following assumptions were made when designing the routine:

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

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

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

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

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

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

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

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

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

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

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

        return(0);
}
```

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

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

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

3.4 Integrating message system files

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

3.4.1 Integrating messages into the PL

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

3.4.2 Building and installing the catalogs

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

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

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

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

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

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

   ```
   CATERRFLAGS  += -s
   ```

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

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

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

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

The message catalog must always be installed unconditionally.

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

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

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

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

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

3.5 Maintaining message system catalogs

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

3.5.1 Deleting a message from a release

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

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

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

3.5.2 Adding and changing messages

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

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