MESSAGE CATALOGS AND NATIVE LANGUAGE SUPPORT

Glenn Cole
Consultant
Fairfax, Virginia USA

Summary

One of the least-used features of the HP3000 is the Message Catalog facility and the more-recent Native Language Support. The message catalog is intended to be used in applications where there is a non-trivial number of (error) messages to the user. The classic application of this is MPE.

The other intended area of use is where the same application program is used by different people who understand different languages. In this case, only the messages need to be changed — modifying and recompiling the application is not necessary.

The message facility is easy to use, has minimal overhead, and may save substantial data stack area. This paper highlights a case history where use of this facility was virtually required, and then details the relatively painless way in which it was implemented. Afterwards, a comparison is given between the original Message Catalog facility of MPE and the similar feature within Native Language Support.

Introduction

The Message Catalog facility provides a means of programmatically accessing messages contained within a specially-formatted file by using standard MPE file system intrinsics. Parameter substitution is allowed, and messages may be routed directly to the list device, retrieved for use by the application program, or both. This facility is provided by Hewlett-Packard as part of the Fundamental Operating System (FOS), and thus is available under all versions of MPE on all HP3000 computer systems at no extra cost.

The Case: (Part 1 - The Problem)

At the time (late 1983), I was employed by a major Moving & Storage company as a Programmer/Analyst, working on a team developing an on-line Dispatch System. It was designed so that when a truck driver called his location in to a dispatcher, the dispatcher could see where the truck was going, update its current location, and pass along any messages to the driver. The Dispatch System also kept a history of each shipment that could be brought up at any time, along with any comments recorded by another dispatcher during handling of the shipment.

The Dispatch System used VIEW and IMAGE extensively, and was written in COBOL II running on a Series 64 under the Q-MIT release of MPE IV. It was a menu driven system with dynamic subprograms. Most of these subprograms used VIEW screens as well. Some of these subprograms could call other subprograms. The "longest path" was about 4 levels deep, not counting the main menu. Most of the subprograms in this path had many messages to retain — the average was about 40 messages of 80 characters each. These were kept in WORKING-STORAGE along with the other data requirements of the subprogram. Each time a subprogram was called, its WORKING-STORAGE SECTION — that is, the data area of the program — was added to the data stack. The end result was that when this "longest path" was used, it resulted in a STACK OVERFLOW and the program aborted.

This was not good news. The target date for implementation was approaching rapidly. Not only did we need a solution, but we needed one that could be implemented fairly quickly, that would not compromise the design set forth by the analysts, and that would work.

The Case: (Part 2 - A Solution)

The stage was set. I knew of the existence of message catalogs by virtue of reading the System Intrinsics Manual. However, I had never used this facility, nor had I ever seen an application that did. None of the other programmers knew anything about message catalogs either.

I happen to like the documentation provided for the HP3000. Sometimes it is difficult to find the desired information. Sometimes the information, once found, is neither entirely accurate nor complete. (Fortunately, on the occasion when it is incorrect, it usually says "This cannot be done" when in fact it can, instead of the other way around.)

In this case, the documentation seemed quite clear, and it appeared that a solution was at hand. First, I set up a small test program to verify that I understood both the concepts and the mechanics of Message Catalogs.

The concepts are rather simple and straightforward. A message catalog (Figure 1) is created initially as a standard MPE flat file, and consists of from 1 to 62 "sets," each containing up to 32767 messages. Sets are indicated by "$SET n" beginning in column 1, where "n" is the set number (1 - 62). The sets should be in ascending sequence by set number, but the numbers need

```
$SET 1  ***  SYSTEM MESSAGES  ***
$       ^The above comment begins HERE (2nd space)
$This comment line is INVALID (no space after "$")
1 LDEV #! IN USE BY FILE SYSTEM
05 IS "!" ON LDEV #! (Y/N)?
$  Note: (1) Numbers need not be CONSECUTIVE
$        (2) Leading zeros are allowed
$        (3) Comments may be inserted between msgs
9 ANOTHER MESSAGE (NOCIERROR)
$SET 3  ***  Non-Consecutive Set Number
10 This message is          &
continued.
$ The above prints "This message iscontinued." (Why?)
```

Figure 1 - Sample Message Catalog

not be consecutive. (For example, a message catalog with only 3 sets — numbered 2, 7, and 15 — is perfectly valid.) The remainder of the line following the space after the set number may be used as a free-form comment (e.g., "$SET 1 *** System Messages"). Other comments may be included by entering a "dollar-sign space" ($) beginning in column 1, with the remainder of the line (columns 3 thru 72) used for the comment.

Each message in the file is uniquely identified by its set number (described above) and by its message number (1 - 32767). Thus, message number 100 in set 1 may be completely unrelated to message number 100 in set 2. Messages are entered under the appropriate $SET heading with the message number beginning in column 1, followed by a single space, followed by the text of the message. Note that the message numbers within a set must be in ascending sequence, though they need not be consecutive. (Please review Figure 1.)

There are three (3) special-purpose characters that may be included in the message text. These include the continuation characters "ampersand" (&) and "percent" (%), and the parameter substitution character "exclamation mark" (!). When a continuation character is entered as the last non-blank character on a line, it indicates that the next line is to be included as part of the current message. It is important to note that all blanks immediately preceding the continuation character are IGNORED. Thus, there is some latitude as to where the continuation character may be placed. More importantly, this means that the following line probably should begin with a space.

Now, recall that the application program may elect to have the message routed directly to the user's terminal. This is where the two continuation characters differ. If the "percent" symbol (%) is used, then a carriage return and line feed are printed before continuing with the next line. The "ampersand" (&), however, indicates that the continuation line should be printed WITHOUT first printing a carriage return and line feed. In this case, the continuation line will be printed adjacent to the current line. These symbols correspond roughly to the SPACE (%40) and %320 carriage control characters.

Each message may contain up to five (5) exclamation marks (!) for parameter substitution.

Each passed parameter is inserted in the message where the corresponding exclamation mark occurs, with the first parameter replacing the first exclamation mark, the second parameter replacing the second exclamation mark, and so on, until all parameters are included. (Please review Figure 1 again. See — a picture really IS worth a thousand words!)

The mechanics of using a message catalog are almost as simple as understanding the concepts given above. Before an application can use a message catalog, it (the catalog) must first be prepared for use (Figure 2). Remember, the messages are entered with EDITOR (for example) and kept as one would keep FORTRAN or SPL source code, that is, as a numbered file with 80-byte fixed-length records.

```
:EDITOR
<<Banner line displayed here>>
/ADD
    1    $SET 1  *** System Messages ***
    2    1 System Msg #1
    3    //
...
/KEEP MSGFILE
/EXIT
END OF PROGRAM
:
:FILE INPUT=MSGFILE
:RUN MAKECAT.PUB.SYS
** VALID MESSAGE CATALOG
END OF PROGRAM
:
:SAVE CATALOG
:PURGE MSGFILE
:RENAME CATALOG,MSGFILE
:
```

This CREATES the initial message file.

This CONVERTS the file to a form understandable by the message system.

This SAVES the converted catalog as a permanent file, which may still be used by EDITOR.

Figure 2 - Preparing the Message Catalog for Use

The next step in preparing this catalog for use involves running the system utility program MAKECAT to install a "directory" on the file as a single user label. This utility reads the data from formal file designator INPUT and builds a temporary file called CATALOG. Any existing temporary file named CATALOG is renamed to CATnn, where "nn" is the first available number. (Note that the first file is renamed to CAT1, not CAT01, and so on.) A short (perhaps cryptic) error message is displayed for each line believed to be in error. If this happens, then the CATALOG file is not built.

Return to the EDITOR, text the file, fix the error, and try again.

Once validated, the message "** VALID MESSAGE CATALOG" is displayed and the CATALOG file is built. At this point, you may :SAVE CATALOG and then :RENAME CATALOG,myfile. (Of course, if a file named CATALOG already exists, you may have to use the :RENAME CATALOG,myfile,TEMP command and then :SAVE myfile.) Once this has been accomplished, the message catalog is ready for use. (Please review Figure 2.) Note that you can still modify the message catalog by returning to the EDITOR, texting the file, making the desired changes, keeping the file either under the same name or under some other name, and running MAKECAT again.

In order for the application program to access the message catalog, the file must first be opened. This is done with the standard FOPEN intrinsic. The important things here are the FOPTIONS and AOPTIONS parameters. FOPTIONS (file options) should include Old, Permanent, and ASCII (%5). AOPTIONS (access options) should include Multi-Record and Nobuf (%420).

The intrinsic GENMESSAGE is the key to the whole operation. A (brief) description of this intrinsic is given in Figure 3. Recall that the target application that I was concerned with used VIEW screens. This meant that the message could not be sent directly to the terminal; the application had to use a buffer and then show the message using the appropriate VIEW calls.

The small test program that I used to verify I understood all of this worked fine. The questions now were: How can this be integrated into the Dispatch System? Should each subprogram call GENMESSAGE explicitly? Will the savings in data stack area be sufficient to allow the program to run without the STACK OVERFLOW?

If the subprograms called GENMESSAGE explicitly, then they would have to know the file number (returned by FOPEN) of the message catalog. This means that the routine performing the FOPEN would have to pass the file number to the other subprograms, which meant modifying the parameter list both in the calling routine and in the called routine. (The other alternative — each subprogram FOPENing the message catalog when beginning, and FCLOSEing it when leaving — could not be considered seriously because of the tremendous overhead the FOPENs and FCLOSEs would cause.) This seemed to have too much chance of programmer error.

There was still another alternative, one that appealed to me from the start, and which seemed to involve the least program modification. Recall that when a DYNAMIC subprogram is called, it gets an "original" copy of its WORKING-STORAGE section. That is, the variables in WORKING-STORAGE are re-initialized to the values defined when the subprogram was compiled. However, for a subprogram that is NOT declared as DYNAMIC, the variables in WORKING-STORAGE are NOT re-initialized. For example, let us imagine a non-DYNAMIC subprogram with a variable named COUNT described initially with a value of zero (0). Let us also imagine that during the course of this subprogram, COUNT is incremented by one (1). Now, for the first execution of this subprogram, COUNT will be zero (0) upon entry, and one (1) upon exit. However, the second time through, since COUNT is not re-initialized to zero, COUNT will be one (1, the previous value) upon entry and two (2) upon exit.

My idea, then, was to have a single subprogram that would do both the FOPEN and the GENMESSAGE calls. The subprogram would define the file number initially with a value of zero (0). This value would be checked and the subprogram execute the FOPEN only when the file number was zero, i.e., on the very first call. A successful FOPEN would change the file number to a non-zero value, so subsequent calls would not execute the FOPEN. (Note, however, that this approach meant that the message catalog would be closed implicitly by the file system when the application terminated, instead of being closed explicitly by the application.)

```
    I                          IV        IV        IV
msglen := GENMESSAGE (filenum, setnum, msgnum,

                          BA        IV
                         buff,     buffsize,

                           LV        LV        LV
                         parmask, parm1,...,parm5,

                          IV         I         0-V
                         msgdest, errnum);


buff     : A byte array to which the message is returned
buffsize: Length, in bytes, of "buff"; passed TO intrinsic

parmask:  16-bit Logical mask describing parms 1 thru 5
     Bit  (0:1) = 1.  Ignore rest of word & parms 1 thru 5
                = 0.  Rest of word, in 3-bit groups, defines
                      parm types for parms 1 thru 5
     Bits (1:3) = (Parm 1 type)
                  0 - byte address of string
                      (string terminated by ASCII "null")
                  1 - integer
                  2 - word address of "double" identifier
                  3 - ignore this parm
     Bits (4:3) = (Parm 2 type; same values as Parm 1)
     Bits (7:3) = (Parm 3 type; same values as Parm 1)
     (etc.)

msgdest :  destination of assembled message;
           0=$STDLIST;  >2=File number of destination file
```

Figure 3 - The GENMESSAGE Intrinsic

The other subprograms did not need the full flexibility available with GENMESSAGE, so adding a few limitations to keep things simple would not be critical. For example, no routine in the Dispatch System needed to substitute more than three (3) parameters in a message, and these values were never more than eight (8) bytes long. Thus, it seemed feasible to code the details of the

message retrieval into a single subprogram. As an added benefit, this meant that the technique could be modified at some point in the future without having to modify the other subprograms. The remainder of this paper refers to this user-written message retrieval subprogram as GETMSG. (Appendix A shows the FORTRAN source code for GETMSG. The actual implementation was written in COBOL, but FORTRAN is more compact. Either language will suffice.)

Replacing the hard-coded messages in the subprograms with calls to GETMSG was not very difficult. Fortunately, it was not very time-consuming either. Since the immediate worry was the "longest path" within the Dispatch System, these subprograms were modified first. After all, if this "solution" did not work, then it was not a solution at all, and the effort expended thus far was merely educational at best.

The subprograms had to know what Set Number and Message Numbers to use, since these were the main input parameters for GETMSG. These were assigned somewhat arbitrarily, based on the number of messages used by a given subprogram. (I think all four subprograms were assigned Set Number 1, and the Message Numbers assigned were 1-100, 101-200, etc. This gave plenty of room for future growth.)

The subprograms had been coded with the messages grouped together in WORKING-STORAGE. When each subprogram was text into the editor, this range of lines was kept in a separate file. (This auxiliary file was modified later so that it could be included as part of the message file. This meant that the actual messages did not have to be re-entered.) In the subprogram, the variable names assigned to the messages remained the same. However, each description was changed from PIC X(40) to PIC S9(4) COMP, and a VALUE equal to the Message Number was assigned. The Message Numbers were assigned in sequence, with the first message getting the lowest Message Number assigned to the subprogram. (Please see Figure 4 for a "before" and "after" look at this area.) Fields were also added to hold the Set Number, the generic Message Number, the retrieved Message, and three (3) 8-byte parameters for substitution (whether needed or not).

The PROCEDURE DIVISION was then searched for all moves to the VIEW window area. Each of these MOVE statements was replaced by two (2) other statements: A MOVE of the desired Message Number to a holding area, and a PERFORM of the utility routine (as yet unwritten) that would retrieve the desired message. (The Set Number did not have to be moved, since it remained constant throughout the subprogram.) Note that if parameter substitution was required for the desired message, the necessary value(s) would also have to be moved prior to the PERFORM.

The utility routine PERFORMed was quite simple, but I felt it was worthy of its own paragraph. It consisted only of a CALL to GETMSG, followed by a MOVE of the returned message to the VIEW window area. Note that if an error occurred in GETMSG (e.g., a missing Set Number/Message Number combination), this was indicated by the returned message.

The data area saved was tremendous, and the "longest path" worked, so this was indeed a solution. Perhaps just as significant is the fact that it took just two (2) days to research and implement it. There was very little new coding involved, and the technique was relatively simple and straight-forward.

```
01  PROGRAM-MESSAGE-AREAS.
    05  PM-NO-SUCH-ENTRY          PIC X(40)  VALUE
        "** No such entry; please try again.     ".
    05  PM-DUPLICATE-TRAILER      PIC X(40)  VALUE
        "** That Trailer already exists!         ".
     :
        Working-Storage before GETMSG
```

```
01  PROGRAM-MESSAGE-AREAS.
    05  PM-NO-SUCH-ENTRY      PIC S9(4) COMP VALUE 101.
    05  PM-DUPLICATE-TRAILER  PIC S9(4) COMP VALUE 102.
     :

01  GETMSG-PARMS.
    05  GP-SET-NUMBER         PIC S9(4) COMP.
    05  GP-MSG-NUMBER         PIC S9(4) COMP.
    05  GP-PARM-1             PIC X(08).
    05  GP-PARM-2             PIC X(08).
    05  GP-PARM-3             PIC X(08).
    05  GP-MESSAGE            PIC X(80).
 :
        Working-Storage after GETMSG
```

Figure 4 - Working-Storage before and after **GETMSG**.

Another Alternative: Native Language Support

In the time since GETMSG was implemented, Hewlett-Packard has introduced Native Language Support (NLS). As with the standard MPE message catalog facility, this is included as part of the Fundamental Operating System; however, I believe you must be running under some version of MPE V. Native Language Support is an excellent idea, and it embodies far more than an improved version of message catalogs (now called the "application message facility" under NLS). However, only the application message facility is discussed below; most of NLS goes beyond the scope of this paper. (The manual is very readable; I strongly encourage you to read through it.)

A chart comparing MPE message catalogs with NLS is shown in Figure 5. Note that the two are very compatible. In fact, the GETMSG routine described earlier can be re-written to use NLS without modifying any other subprogram!

	MPE	NLS
Limits:		
Set #'s:	1 - 62	1 - 32766
Message #'s:	1 - 32767	1 - 32767
# User Labels:	1	0
Data Compression?	NO	YES (see text)
Physical Characteristics:	REC=40,16,F,BINARY; CODE=0	REC=128,1,F,BINARY; CODE=MGCAT
Formatting Program:	MAKECAT.PUB.SYS	GENCAT.PUB.SYS
Intrinsics Used:	FOPEN, FCLOSE, GENMESSAGE	CATOPEN, CATCLOSE, CATREAD
Order of Parameter Substitution:	Determined by calling sequence.	May be forced by message. If not, then determined by calling sequence.

Figure 5 - MPE Message Catalog vs. NLS

There are a few things worth noting that are not mentioned in the chart. First, the only data compression used by NLS is based on the blank space between the last non-blank character of each line and the logical end-of-record. (Remember, the Editor's line number is actually in the last eight (8) bytes of each record. The logical end-of-record mentioned above refers to the position just before this.) Thus, the worst place for the continuation character (in terms of data compression) is in the last position of the record. The amount of disc space consumed by this file compared to that of a standard MPE message catalog appears to be directly proportional to the average message length. Thus, a file whose average message takes up one-half of the record consumes about one-half of the disc space of the equivalent MPE message catalog. (As an example, the system message catalog takes up about 40% less space when formatted for NLS.)

Regarding physical disc I/O's: a brief study of the File Close records in the system log file shows that while the number of RECORDS processed remains the same whether or not NLS is used, the number of BLOCKS processed is reduced anywhere from three (3) to over ten (10) times. That is, NLS may require less than one-tenth of the disc I/O's of an equivalent MPE message catalog.

Note also that the blocks are smaller under NLS — 128 x 1 = 128 words for NLS versus 40 x 16 = 640 words for MPE.

Conclusion

In conclusion, I would like to encourage you to look into message catalogs. They are easy to use, easy to customize, and in some cases, they may prove critical to the success of an application. I would also like to encourage you to look into Native Language Support. The reference manual is very readable. Among other things not mentioned in this paper, it explains how a single application can easily reference different message catalogs geared toward different languages. Hewlett-Packard has put considerable time, thought, and effort into this, and has made these tools available to us at no extra cost.

Biography

Glenn Cole has a B.S. in Mathematics from James Madison University, Harrisonburg, VA. He has been using the HP 3000 since 1978, and is now an HP 3000 consultant in northern Virginia (USA).

Appendix A. FORTRAN source for GETMSG.

```
      SUBROUTINE GETMSG (ISET, IMSG, CPARM1, CPARM2, CPARM3, CMSG)
C
      CHARACTER*08 CPARM1, CPARM2, CPARM3
      CHARACTER*80 CMSG
C
      INTEGER        IMSGLEN
      CHARACTER*28 CCATNAME
      CHARACTER*10 CPARML (3)
      LOGICAL        LPARM1, LPARM2, LPARM3
      EQUIVALENCE  (LPARM1, CPARML (1))
      EQUIVALENCE  (LPARM2, CPARML (2))
      EQUIVALENCE  (LPARM3, CPARML (3))
C
      DATA IFILE/0/, IMAXLEN /80/, CCATNAME /"CATFILE.PUB.PROD "/
C
      SYSTEM INTRINSIC FOPEN, GENMESSAGE
C
C     ---------------------------------------------------
C     Open message catalog if not open already.
C
      IF (IFILE .NE. 0) GO TO 10
          IFILE = FOPEN (CCATNAME, %5L, %420L)
          IF (.CC.) 5, 9, 5
5             CMSG = "** Message catalog not open.  Call M.I.S."
              GO TO 99
9         CONTINUE
10    CONTINUE
C
C     ---------------------------------------------------
C     Init parms (ASCII NUL (%0) at end of each string).
C
      CPARML (1) = CPARM1
      CPARML (2) = CPARM2
      CPARML (3) = CPARM3
      DO 20 I = 1, 3
          CPARML (I) [9:2] = "  "
          CPARML (I) [INDEX(CPARML(I)," "):1] = %0C
20        CONTINUE
C
C     ---------------------------------------------------
C     Actual message retrieval; %0 indicates all parms are STRINGS
C
      IMSGLEN = GENMESSAGE (IFILE, ISET, IMSG, CMSG, IMAXLEN,
     +                      %0, LPARM1, LPARM2, LPARM3,,,,IERROR)
C
      IF (IERROR .EQ. 0) GO TO 99
C
```

```
            CMSG = "** Message retrieval failed.  Call M.I.S."
C
C           ------------------------------------------------
C           Wrap up
C
99          RETURN
            EXIT
```