Simplification thru Symbols

Author: Mr. Bruce Claremont, Software Migration & OpenVMS Consultant

Overview

DCL procedure development and maintenance can be simplified through judicious use of DCL symbol assignments. This article provides an outline of how to use symbols to improve procedures and the user interface.

Introduction

Many coders develop code with little thought to those that will inherit it or the users that must interface with it. I develop code with two core objectives always in mind:

- Ease of maintenance

- User friendly

Achieving these objectives with DCL is easy with a little advanced preparation. Part of that advanced preparation is using a consistent set of symbols to represent common functions, which is the subject of this article.

Defining Common Attributes

In addition to storing values, DCL symbols provide a nice way to define shortcuts for commonly used commands. Over time I developed a set of global symbols that define commands, values, and VT escape sequences that I commonly employ in DCL procedures. I collected them into a single procedure I named SET_ATTRIBUTES.COM. I invoke SET_ATTRIBUTES at the beginning of each application procedure I write and I use it in most of my home-grown system maintenance utilities.

SET_ATTRIBUTES.COM Procedure

The SET_ATTRIBUTES procedure appears in its entirety in Figure 1. Subsequent sections discuss the procedure components in detail.

Figure 1: SET_ATTRIBUTES.COM Procedure

```
$! SET_ATTRIBUTES.COM
$! Procedure defines standard symbols and escape sequences.
$!
$ IF F$TRNLNM("MSI$ATTRIBUTES") .EQS. ""
$    THEN
$!
$!      Determine processing mode.
$!
$       _BATCH          == 0
$       IF F$MODE() .EQS. "BATCH" THEN _BATCH == 1
$       _INTERACTIVE    == 0
$       IF F$MODE() .EQS. "INTERACTIVE" THEN _INTERACTIVE == 1
$!
$!      Define VT terminal control codes (escape sequences).
$!
$       _BELL[0,8]    == %X7
$       _ESC[0,8]     == %X1B
$!
$       _BLINK        == "''_ESC'[5m"
$       _BOLD         == "''_ESC'[1m"
$       _CANCEL       == "''_ESC'[m"
$       _CEOL         == "''_ESC'[K"                !Clear to end-of-line.
$       _CEOS         == "''_ESC'[J"                !Clear to end-of-screen.
$       _CLEAR        == "''_ESC'[2J''_ESC'[1;1f"   !Clear entire screen
$!                                                  !and home cursor.
$       _EOS          == "''_ESC'[24;1f"            !Go to end of screen.
$       _REVERSE      == "''_ESC'[7m"
$       _UNDERLINE    == "''_ESC'[4m"
$!
$!      Get process information.
$!
$       _PID   == F$GETJPI("","PID")
$       _NODE  == F$GETSYI("NODENAME")
$       _USERNAME == F$EDIT(F$GETJPI("", "USERNAME"), "COLLAPSE")
$!
$!      Define common DCL command symbols.
$!
$       RESET*_TERM :== SET TERMINAL /LINE_EDIT /NUMERIC_KEYPAD
$!
$       IF _INTERACTIVE
$         THEN
$               SAY :== WRITE SYS$COMMAND
$               ASK :== READ SYS$COMMAND /END_OF_FILE=CANCEL_PROCEDURE /PROMPT="
$         ELSE
$               SAY == "!"
$               ASK == "!"
$        ENDIF
$!
$       ASSIGN /NOLOG 1 MSI$ATTRIBUTES
$   ENDIF
$!
$ END_PROCEDURE:
$ EXIT
```

What's With the Underscores?
The underscores you see prefixing many of my symbols are not a necessity. They are a convention I use to avoid conflicts with existing symbols on client systems. For example, a client might be using a symbol called ESC, and I do not want to confuse their ESC symbol with mine, so I prefixed mine with an underscore; i.e., _ESC.

Saving Cycles

The first item in the procedure is a check for the existence of the process logical name MSI$ATTRIBUTES.

```
$ IF F$TRNLNM("MSI$ATTRIBUTES") .EQS. ""
$   THEN
```

The reason for this goes back to the dawn of the computer age when saving processor cycles was important. The symbols created by this procedure generally only need to be defined once per user session. To ensure the symbols get defined, I place a call to this procedure at the beginning of all user procedures. The MSI$ATTRIBUTES check determines if the SET_ATTRIBUTES procedure has already been run. If so, there is no need to run it again. Thus, processor cycles are saved, something which is still beneficial today. You will find the following ASSIGN statement at the end of the procedure that creates the MSI$ATTRIBUTES process logical.

```
$      ASSIGN /NOLOG 1 MSI$ATTRIBUTES
```

Now I can already hear all the VMS cognoscenti groaning that placing the SET_ATTRIBUTES call in SYS$MANAGER:SYLOGIN.COM or the user's LOGIN.COM procedure would eliminate the need to call it from each application procedure and they are right. Placing the procedure call at the beginning of each application procedure was a packaging decision. It allows me to install applications on multiple systems at multiple sites and not have to concern myself with modifying system or user login processes.

Process Mode

```
$!     Determine processing mode.
$!
$      _BATCH          == 0
$      IF F$MODE() .EQS. "BATCH" THEN _BATCH == 1
$      _INTERACTIVE    == 0
$      IF F$MODE() .EQS. "INTERACTIVE" THEN _INTERACTIVE == 1
```

Knowing the process mode is often useful and it is information that only needs to be captured once per session. Knowing whether a process is running interactively or in batch allows intelligence to be built into procedures to determine how they respond to errors and exceptions. For example, a procedure running interactively that encounters an error could check the _INTERACTIVE symbol and upon verifying interactive mode, issue a message to the user terminal and pause for a response. The same procedure running in batch would check the _INTERACTIVE symbol, find it was not running interactively, issue an error message via e-mail, and abort with an error status.

Binary Overlays

```
$      _BELL[0,8]     == %X7
$      _ESC[0,8]      == %X1B
```

Binary overlays provide a handy method to place non-printable characters in a symbol. In my case, two non-printable characters I often use are a hex 7, which sounds the terminal bell, and a hex 1B, or the Escape character, which is used to prefix VT control codes (more on control codes a bit further on). The above example demonstrates how to define the bell and escape characters as DCL symbols, making them easy to access and understand in a DCL procedure.

Terminal Control Codes (Escape Sequences)

When using DCL procedures to interact with users, it is often advantageous to format and highlight text, generating a more attractive, friendly user interface. Doing so makes the user's job easier, a benefit which may have a direct bearing on your continued employment.

VT terminals came equipped with a standard set of control codes that managed data display characteristics. Any VT terminal emulator worth using supports these control codes. The control codes were often called Escape Sequences because they were generally prefixed with the Escape character.

Using the previously defined _ESC symbol, it is possible to create symbols that represent common VT escape sequences. That's what this next section of DCL code accomplishes.

```
$      _BLINK       == "''_ESC'[5m"
$      _BOLD        == "''_ESC'[1m"
$      _CANCEL      == "''_ESC'[m"
$      _CEOL        == "''_ESC'[K"              !Clear to end-of-line.
$      _CEOS        == "''_ESC'[J"              !Clear to end-of-screen.
$      _CLEAR       == "''_ESC'[2J''_ESC'[1;1f" !Clear entire screen
$!                                              !and home cursor.
$      _EOS         == "''_ESC'[24;1f"          !Go to end of screen.
$      _REVERSE     == "''_ESC'[7m"
$      _UNDERLINE   == "''_ESC'[4m"
```

Process Information

I like having static process information on hand. I often incorporate it into things like temporary file names and informational messages. By creating symbols containing the data in SET_ATTRIBUTES, the lexical functions need only be run once, saving more of those precious processor cycles. The _PID, _NODE, and _USERNAME symbol definitions provide examples of this type of information capture.

```
$      _PID  == F$GETJPI("","PID")
$      _NODE == F$GETSYI("NODENAME")
$      _USERNAME == F$EDIT(F$GETJPI("", "USERNAME"), "COLLAPSE")
```

DCL Command Symbols

A favorite use for DCL symbols is to create short cuts for long DCL commands. Take a look at the following examples:

```
$!     Define common DCL command symbols.
$!
$      RESET :== SET TERMINAL /LINE_EDIT /NUMERIC_KEYPAD
$!
$      IF _INTERACTIVE
$        THEN
$          ASK :== READ SYS$COMMAND /END_OF_FILE=CANCEL_PROCEDURE /PROMPT="
$          SAY :== WRITE SYS$COMMAND
$        ELSE
$            ASK == "!"
$            SAY == "!"
$       ENDIF
```

The RESET symbol provides a quick way to issue a long SET TERMINAL command. The ASK and SAY symbols provide short, descriptive commands that make both writing and reading DCL procedures easier.

Note the use of the _INTERACTIVE symbol. Whether the process is interactive determines how the ASK and SAY symbols are defined. The commands represented by ASK and SAY are useful in an interactive procedure, but not in a batch procedure. Hence, in a batch procedure the symbols serve to define the lines they reside in as comments.

Putting it Together

Everything discussed thus far is basic DCL. Put it together and you have a mechanism to simplify DCL code development and maintenance. The following procedure provides an example of how the symbols defined in SET_ATTRIBUTES can be used. The procedure checks for the existence of a user-supplied file, prompting for a file name if one is not provided. The symbols created in SET_ATTRIBUTES help simplify and clarify the DCL code in this procedure.

```
$! CHECK_FILE.COM
$!     Check file specified in P1 to ensure that it is valid.
$!     If P1 is null, prompt user for a file name.
$!
$!     On Entry:
$!          P1 - File name            (optional)
$!          P2 - Prompt string        (optional)
$!          P3 - Default extension    (optional)
```

```
$!      On Return:
$!          FILE$ - Validated file designation.
$!
$       ON ERROR THEN GOTO ERROR_TRAP      !Error trap.
$       ON CONTROL_Y THEN GOTO ERROR_TRAP !User abort trap.
$       STATUS = 1                          !Default exit status value.
$       @SYS$UTILITIES:SET_ATTRIBUTES.COM !Set terminal & process attributes.
$!
$       FILE$ == ""
$       PROMPT = "File> "
$!
$ START:
$       IF P1 .EQS. ""
$         THEN
$               IF .NOT. _BATCH
$                 THEN
$                       SAY ""
$                       IF P2 .NES. "" THEN PROMPT = P2 + "> "
$                       ASK "''PROMPT'" P1
$                 ENDIF
$               GOTO START    !Check file name.
$         ENDIF
$!
$!      Apply default extension if specified and the specified file name
$!      doesn't already have one.  If a file designation ends in a period,
$!      treat that as a valid extension and do not apply the default extension.
$!
$       IF P3 .NES. ""
$         THEN
$               IF F$EXTRACT(F$LENGTH(P1) - 1, 1, P1) .NES. "."
$                 THEN
$                       IF F$LOCATE(".", P3) .EQ. F$LENGTH(P3) THEN -
$                         P3 = "." + P3
$                       EXT = F$PARSE(P1,,, "TYPE")
$                       IF EXT .EQS. "." THEN P1 = P1 + P3
$                 ENDIF
$         ENDIF
$!
$!      Ensure file exists.
$!
$       IF F$SEARCH(P1) .EQS. ""
$         THEN
$               SAY _BELL, "ERROR - File ", _BOLD, P1, _CANCEL, " not found."
$               P1 = ""
$               GOTO START
$         ENDIF
$       FILE$ == F$PARSE(P1,,, "DEVICE") + F$PARSE(P1,,, "DIRECTORY") -
$               + F$PARSE(P1,,, "NAME") + F$PARSE(P1,,, "TYPE")
$!
$ END_PROCEDURE:
$       EXIT 'STATUS' + 0*F$VERIFY(VERIFY)
$!
$ ERROR_TRAP:
$       SAY _BELL
$       SAY "An error or <Ctrl^Y> has aborted this procedure."
$ CHECK_POINT:
$       IF F$MODE().NES."BATCH"
$         THEN
$               ASK "''_BELL'Enter 0 to exit the procedure: " CHK
$               IF CHK .NES. "0" THEN GOTO CHECK_POINT
$         ENDIF
$ CANCEL_PROCEDURE:
$       STATUS = 3
$       SET TERMINAL /LINE_EDITING
$       GOTO END_PROCEDURE
```

Conclusion

Using DCL symbols wisely provides the means to simplify your DCL procedures while enhancing their functionality. A procedure like SET_ATTRIBUTES also provides a single point of reference to make global changes to commonly used symbols, which is yet another maintenance benefit. As usual, a little forethought can go a long way towards improving the end product.

Thanks for sticking it out to the end and reading the entire article. Your comments and feedback are welcome.

About the author: Mr. Bruce Claremont has been working with OpenVMS since 1983. Mr. Claremont has extensive programming, project management, and system management experience. He also likes motorcycles. He founded Migration Specialties in 1992 and continues to deliver OpenVMS and application migration services along with VAX, Alpha, PDP-11, HP1000, and Data General hardware emulation ports. You can reach Bruce at +1 719-784-9196. More information about Migration Specialties products and services can be found at *www.MigrationSpecialties.com*.

For more information

More on DCL development:

- *Simplifying Maintenance with DCL*,

  http://h71000.www7.hp.com/openvms/journal/v9/simplifying_maintenance_with_dcl.html

Real world DCL examples:

- *ODS-2/ISO-9660 CD Creation*

  http://www.migrationspecialties.com/pdf/ODS-ISO.pdf

- *Using OpenVMS to Meet a Sarbanes-Oxley Mandate*

  http://www.migrationspecialties.com/pdf/Using%20OpenVMS%20to%20Meet%20a%20Sarbanes-Oxley%20Mandate2.pdf

VT Terminal Information (with thanks to Ian Miller for the links)

- http://www.vt100.net/

- http://www.cs.utk.edu/~shuford/terminal/dec.html