OpenVMS Technical Journal V9

Simplifying Maintenance with DCL

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

Overview

An important element of effective software application maintenance and a crucial component of job retention is expedient problem resolution. Key to resolving a problem is effective identification of its cause. A couple of decades(!) ago I developed a standardized wrapper for OpenVMS DCL procedures that proved an effective assistant in locating processing problems. I have successfully deployed this code at many sites and continue to use it to this day. I offer it here as a means to ease the burden for those of you maintaining batch and interactive DCL processes in production environments.

Introduction

A key component to effective software maintenance is quick problem identification. OpenVMS users are fortunate in this regard in that DCL provides a nice set of commands to facilitate error capture and reporting. This document presents a DCL procedure template that demonstrates how to take advantage of those commands. You will also get a look at my basic coding philosophy, which is:

- *KISS* (Keep It Simple Sweetie ;)
- Be consistent
- Use comments!

DCL Procedure Template Overview

The template is designed to be applied to new and existing procedures. The template is very simple and does not limit procedure functionality. It enforces structured programming techniques via a single point of entry and exit. It also enforces implementation of return status values and a standard error handling routine.

Figure 1 presents the template in its entirety. Subsequent sections discuss the template components in detail.

Figure 1: DCL Procedure Template

```
$ IF F$MODE() .EQS. "BATCH" THEN SET VERIFY    !Always the first line!
$!---------------------------------------
$! proc.COM
$!     procedure description
$!     Created:  id, mmm-yyyy
$!     Modified:  id, mmm-yyyy
$!     -
$!-------------------------------------------------------------------------------
$!     STANDARD PROCEDURE INITIALIZATION SECTION
$!
$ _BATCH = 0
$ IF F$MODE() .EQS. "BATCH" THEN _BATCH = 1
$ _BELL[0,8] = %X7
$ _STATUS = 1                           !Return Status
$ SAY := WRITE SYS$COMMAND
$ ASK := READ SYS$COMMAND -
       /END_OF_FILE=CANCEL_PROCEDURE -
       /PROMPT="
$ ON ERROR THEN GOTO ERROR_TRAP         !Standard abort trapping.
$ ON CONTROL_Y THEN GOTO ERROR_TRAP
$ IF _BATCH THEN SAY = "!"              !If a batch process, convert info
$!                                       messages to comments.
$!-------------------------------------------------------------------------------
$!     PROCEDURE BODY
       .
       .
       .
$!-------------------------------------------------------------------------------
$!     STANDARD PROCEDURE TERMINATION CODE
$!
$ END_PROCEDURE:
$!     *** Procedure specific termination code goes here. ***
$!
$ EXIT '_STATUS'
$!
$ ERROR_TRAP:
$!     *** Procedure specific error handling code goes here. ***
$!
$ IF .NOT. _BATCH
$   THEN
$ CONT_PROMPT:
$      SAY _BELL, "Procedure: ", F$ENVIRONMENT("PROCEDURE")
$      ASK "Procedure aborted by a <Ctrl^Y> or error.  Enter 0 to continue: " _QST
$      IF _QST .NES. "0" THEN GOTO CONT_PROMPT
$  ENDIF
$!
$!-------------------------------------------------------------------------------
$ CANCEL_PROCEDURE:
$!     STANDARD ABORT HANDLING CODE (Don't mess with this)
$!
$!     If this is the primary procedure (depth=0) and it is a batch procedure,
$!     have it exit with an error status (2) if it has terminated abnormally.
$!
$ IF _BATCH .AND. F$ENVIRONMENT("DEPTH") .LE. 0
$   THEN
$      _STATUS = 2
$   ELSE
$      _STATUS = 3
$  ENDIF
$ GOTO END_PROCEDURE
```

Detailed Template Walk-Through

The following sections provide a detailed walk-through of the DCL procedure template shown in Figure 1. The template components are explained, as is the reasoning behind them.

Any template is only effective if it is associated with a set of rules. What follows are the DCL coding policies I enforce when using this template. They might seem restrictive at first glance, but they are actually quite simple and easy to live with. They provide a fringe benefit in that they encourage structured coding practices.

Standardized Procedure Development Rules

- No EXIT statements are permitted within the body of the procedure. Only one EXIT statement appears in any procedure and its location is pre-set in the termination code.
- All procedure termination runs through the END_PROCEDURE section.
- If the procedure is terminated abnormally for any reason, it must run through the ERROR_TRAP section.
- If the procedure is cancelled for any reason, it must run through the CANCEL_PROCEDURE section.
- Avoid using the following commands and lexical function:
  - `SET NOVERIFY` or `F$VERIFY(0)`

    I require that batch jobs create log files. You will find that government accountability mandates do too. Turning off verification prevents data from being written to the log files.
  - `ON` *`condition`* `THEN CONTINUE`

    If a procedure generates an error, it is required to report the problem. Continuing from an error condition without a notification event is not allowed.
  - `SET NOON`

    Do not use the SET NOON command to disable error trapping within a procedure. Its sole purpose is to reset error trapping protocols in special circumstances. Any time the SET NOON command is used, it should be immediately followed by these two lines of code:

    ```
    $ ON ERROR THEN GOTO ERROR_TRAP          !Standard abort trapping.
    $ ON CONTROL_Y THEN GOTO ERROR_TRAP
    ```

    These commands preserve standard error trapping functionality in the procedure.
- Use the ON *condition* commands with care. When special traps are created in procedures, ensure that they use the standard error routine to terminate the procedure. When the need for a special trap has passed, insert the following instruction:

  ```
  $ SET NOON
  $ ON ERROR THEN GOTO ERROR_TRAP             !Standard abort trapping.
  $ ON CONTROL_Y THEN GOTO ERROR_TRAP
  ```
- An important aspect of the template's design is to capture and control procedure aborts. This is particularly important when using nested procedures (a procedure called by another procedure). To preserve exit control, all nested procedure calls should be immediately followed by this command line:

  ```
  $ IF $STATUS .EQ. 3 THEN GOTO CANCEL_PROCEDURE
  ```

  Example:

  ```
  $ @PAYROLL:SINKORSWIM.COM
  $ IF $STATUS .EQ. 3 THEN GOTO CANCEL_PROCEDURE
  ```

  The `IF $STATUS` command line allows nested procedures that have generated an error to exit in a controlled fashion, preserving the error control capability.

Procedure Initialization

Now we will take a look at what I call the *procedure initialization section*. This is a short section of DCL code (a *snippet* in modern parlance) that goes at the beginning of each command procedure. It establishes a standard set of symbols and error control functions.

Figure 2: Procedure Initialization Section

```
$ IF F$MODE() .EQS. "BATCH" THEN SET VERIFY    !Always the first line!
$!---------------------------------------
$! proc.COM
$!     procedure description
$!     Created:   id, mmm-yyyy
$!     Modified:  id, mmm-yyyy
$!     -
$!-------------------------------------------------------------------------------
$!     STANDARD PROCEDURE INITIALIZATION SECTION
$!
$ _BATCH = 0
$ IF F$MODE() .EQS. "BATCH" THEN _BATCH = 1
$ _BELL[0,8] = %X7
$ _STATUS = 1                              !Return Status
$ SAY := WRITE SYS$COMMAND
$ ASK := READ SYS$COMMAND -
       /END_OF_FILE=CANCEL_PROCEDURE -
       /PROMPT="
$ ON ERROR THEN GOTO ERROR_TRAP            !Standard abort trapping.
$ ON CONTROL_Y THEN GOTO ERROR_TRAP
$ IF _BATCH THEN SAY = "!"                 !If a batch process, convert info
$!                                          messages to comments.
$!-------------------------------------------------------------------------------
$!     PROCEDURE BODY
```

- $ IF F$MODE() .EQS. "BATCH" THEN SET VERIFY

  In our coding polices, we state that batch procedures must always log the commands they execute. The F$MODE() lexical function on the template's first line ensures that the procedure will turn on verification if it is running in a batch process. If the procedure is being run interactively, it will preserve the verification mode in effect when it is executed. This line must always be the first line in the procedure.

  Why the insistence on having this as the first line? If some miscreant has disabled logging in a batch procedure that calls this one, ensuring that the mode is set to VERIFY in the first line guaranties that you will see the comment lines that follow in the resultant log file. These very nicely inform you what procedure is running which immensely simplifies process diagnostics.

- Introductory Comments

  Next comes a set of introductory comments. These include the procedure name, a brief description of the procedure's purpose, and creation and modification information. At Migration Specialties, we are firm believers in the judicious use of comments in code. In fact, if you are employing coders that do not use comments, you should fire them and hire us. This isn't just a shameless marketing plug, it's sound business practice. Well-commented code speeds problem assessment and resolution.

- Standard Symbols

  Now it's time to define our standard symbols.

```
$ _BATCH = 0
$ IF F$MODE() .EQS. "BATCH" THEN _BATCH = 1
$ _BELL[0,8]    = %X7
$ _STATUS = 1                          !Return Status
$ SAY := WRITE SYS$COMMAND
$ ASK := READ SYS$COMMAND -
   /END_OF_FILE=CANCEL_PROCEDURE -
   /PROMPT="
```

  - **_BATCH**: Signifies the calling mode. 0 = Interactive; 1 = Batch.
  - **_BELL**: Symbolic representation of the control character (^G) that sounds a beep on a VT compatible terminal.
  - **_STATUS**: The _STATUS symbol is used to permit exiting of nested procedures in a controlled fashion when an error is encountered.
  - **SAY**: A symbolic representation of the command WRITE SYS$COMMAND. In other words, a shortcut. It is more common to see this shortcut defined as WRITE SYS$OUTPUT. I use SYS$COMMAND so output will appear on the display even if the procedure is executed with output redirected via the /OUTPUT qualifier.
  - **ASK**: A shortcut for an interactive READ command. Here SYS$COMMAND is used instead of SYS$INPUT to avoid problems with calls from nested procedures and batch procedures.

    Note the use of the /END_OF_FILE qualifier. Its usage tells the procedure to branch to the CANCEL_PROCEDURE label if a user enters a <Ctrl^Z> at an interactive prompt. <Ctrl^Z> is a common way to exit utilities in VMS, so using it in your procedures preserves operational continuity. Your users won't notice, and that's a good thing.

    Usage of both the SAY and ASK symbolic commands is demonstrated in the Error Trapping section later on in this document.

  So what's with the use of the underscore (_) as a symbol prefix? This is a technique I developed to avoid conflicts with existing symbols on client systems. For example, a client might be using a symbol called STATUS, and I do not want to confuse their STATUS symbol with mine, so I prefixed mine with an underscore; i.e., _STATUS.

  All right, so why didn't I prefix SAY and ASK with an underscore? It has been my experience that these are commonly used command symbols so making mine unique was not necessary.

  Ah ha! You have already seen room for improvement, haven't you? These commands could be placed in their own command procedure, defined as global instead of local symbols. Then all you would need to do is call this "setup" procedure to acquire all your standard symbol definitions. You are absolutely right and that is what I normally do. I have declared these symbols inline for clarity in this article. Creating a standard setup configuration for command procedures is gist for another article.[1]

- ON ERROR & ON CONTROL_Y

  These commands are the heart of the error and abort control functionality the template provides.

```
$ ON ERROR THEN GOTO ERROR_TRAP            !Standard abort trapping.
$ ON CONTROL_Y THEN GOTO ERROR_TRAP
```

  Both ON ERROR and ON CONTROL_Y hand off process errors and user aborts (like entry of a <Ctrl^C>) to the label ERROR_TRAP. This turns control of the procedure over to the error trap routine, which ensures execution of problem notification and controlled exit routines. More details on these functions appear in the Error Trapping section.

- Changing Output to Comments

[1] If you think I'm getting rich by grinding out multiple articles for the HP Technical Journal, check out the pay scale for authors. We do it because we love OpenVMS and enjoy sharing information.

```
$ IF _BATCH THEN SAY = "!"          !If a batch process, convert info
$!                                   messages to comments.
```

This is a little trick I use to allow a procedure to be run interchangeably as an interactive or batch process. When run interactively, the text in the SAY lines is displayed on the user's terminal. When run as a batch job, the SAY lines appear in the log file as comments. Not using this technique does no harm, but the batch log files will be messier, which doesn't aid problem diagnosis. Remember this old adage that I just made up: *cleanliness leads to clarity*.

That's it for the *procedure initialization section*. Once tailored to meet your specific requirements, the procedure initialization code should be identical for all production procedures.

Procedure Body

Between the *procedure initialization* and *procedure termination sections* lies the *procedure body*. Within the bounds of the coding rules listed at the beginning of this document, anything goes in the *procedure body*. Don't forget to comment your code; your job might depend upon it.

Also, don't forget to include the following line after any nested procedure calls in the procedure body. The reason for this will be explained in the Canceling a Procedure section.

```
$ IF $STATUS .EQ. 3 THEN GOTO CANCEL_PROCEDURE
```

Procedure Termination Section

Even shorter than the *procedure initialization section*, the *procedure termination section* serves as the common, *and only*, exit point for the procedure.

Figure 3: Procedure Termination Section

```
$!-------------------------------------------------------------------------------
$!     STANDARD PROCEDURE TERMINATION CODE (Modify with care)
$!
$ END_PROCEDURE:
$!
$!     *** Any procedure specific termination code goes here. ***
$!
$      EXIT '_STATUS'
```

This section of the template is where all procedure termination takes place. This is the only place in the procedure where an EXIT statement is allowed. Any code in the body of the procedure that exits normally should do so by executing a GOTO END_PROCEDURE command.

Any special instructions that need to be executed prior to procedure termination should also appear in this section. For example, deletion of temporary work files could take place here as part of the procedure clean-up process. The code in this section is executed every time the procedure terminates, regardless of whether the termination is normal or abnormal, so be careful with any code added to this section. Abnormal terminations are discussed in the Error Trap and Canceling a Procedure sections.

But wait! If you had a general set of wind down commands you wanted to execute, wouldn't this be a great place to put the call to such a procedure? Absolutely! This is why standardized code is so useful.

Note that the _STATUS value is passed to the calling procedure with the EXIT command. The _STATUS value will be read by the calling procedure as the standard system symbol $STATUS. This is how we let the calling procedure know the termination state of nested procedure. Possible return status values are:

1. Success
2. Error
3. Abnormal termination

More on _STATUS code setting and usage appears in the Canceling a Procedure section.

Error Trapping

Thus far, the template has primarily provided a standardized initialization and coding schema. Now we are getting into the sections that handle things when problems occur.

Figure 4: Procedure Error Trapping Section

```
$ ERROR_TRAP:
$!     *** Procedure specific error handling code goes here. ***
$!
$ IF .NOT. _BATCH
$   THEN
$ CONT_PROMPT:
$      SAY _BELL, "Procedure: ", F$ENVIRONMENT("PROCEDURE")
$      ASK "Procedure aborted by a <Ctrl^Y> or error.  Enter 0 to continue: " _QST
$      IF _QST .NES. "0" THEN GOTO CONT_PROMPT
$  ENDIF
```

The ERROR_TRAP section provides a standard mechanism for capturing and reporting errors. This is where procedure control branches to when an unexpected processing error occurs or a user deliberately aborts a procedure. This is the target of the ON ERROR and ON CONTROL_Y commands in the initialization section.

Any special recovery code that is not implemented in the body of the procedure can be placed immediately following the ERROR_TRAP label. The recommended coding standard is to implement specialized error trapping within the procedure body and then initiate termination by executing the GOTO ERROR_TRAP command.

Using this template, interactive processes encountering an error display an error message on the user's screen and pause for user confirmation before terminating. The purpose of this is to preserve the original error message on the screen, allowing users to see and report the message. Forcing entry of a zero to continue prevents errors from being missed because the user pressed the <Return> key multiple times.

This template does not demonstrate error logging and reporting for batch jobs. Adding this functionality is as simple as adding an ELSE clause to the IF .NOT. _BATCH statement. You will find a more comprehensive example in the article *Using OpenVMS to Meet a Sarbanes-Oxley Mandate*, available at this link:

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

Canceling a Procedure

We have arrived at the final section of the template. The CANCEL_PROCEDURE label is where execution control is passed if:

- A <Ctrl^Z> is used to terminate an ASK statement.
- A nested procedure exits abnormally ($STATUS = 3).
- The process stream needs to be cancelled immediately; i.e., a decision process in the body of the procedure executed a GOTO CANCEL_PROCEDURE statement.
- An error or abort is processed via the error trap section.

Figure 5: Procedure Cancellation Section

```
$ CANCEL_PROCEDURE:
$!     STANDARD ABORT HANDLING CODE (Don't mess with this)
$!
$!     If this is the primary procedure (depth=0) and it is a batch procedure,
$!     have it exit with an error status (2) if it has terminated abnormally.
$!
$ IF _BATCH .AND. F$ENVIRONMENT("DEPTH") .LE. 0
$   THEN
$      _STATUS = 2
$   ELSE
$      _STATUS = 3
$  ENDIF
$ GOTO END_PROCEDURE
```

CAUTION: The next few paragraphs reference the template defined symbol _STATUS and the standard system return status symbol $STATUS. Pay attention to the symbol names, as distinguishing between the two symbols is important.

The procedure cancellation section's job is to ensure that nested procedures exit in a controlled fashion. This is where using the IF $STATUS check immediately after nested procedure calls comes into play:

```
$ IF $STATUS .EQ. 3 THEN GOTO CANCEL_PROCEDURE
```

If a nested procedure encounters an error or abort, it reports it, then exits with the standard system return status symbol $STATUS set to 3 via the _STATUS declaration in the EXIT statement ($ `EXIT '_STATUS'`). Setting $STATUS to 3 achieves two objectives:

1. Setting $STATUS to an odd numeric value (3) allows the nested procedure to exit without generating another error. Setting $STATUS to 2 or another even value would generate an error in the calling procedure, resulting in another trip through the calling procedure ERROR_TRAP section. We want to avoid this, both for efficiency and to prevent further aggravating users. I know it can be hard, but we are here to serve and protect the users.
2. The IF $STATUS check immediately following the nested procedure call queries the return status. If a 3 is returned, it signifies an abnormal termination initiated by the nested procedure. Control is immediately passed to the CANCEL_PROCEDURE label in the calling procedure.

If this is an interactive procedure, that's the end of the decision process. The procedure stack will unwind, passing a return status of 3 up the line until the command line or calling menu is reached.

In the case of a batch process, an additional check is needed. We want the batch process to exit with an error status so that the job information is retained via the /RETAIN=ERROR function of the queue manager. (You do have RETAIN=ERROR set on all your batch queues, right?!) Consequently, as the stack of nested batch procedures unwinds, each procedure in the stack uses the lexical function F$ENVIRONMENT("DEPTH") to determine if it was the first procedure called. When the first procedure is reached, _STATUS is set to 2 to force an error condition via the system symbol $STATUS when the primary procedure terminates.

Well, you might ask, why not just blow up batch processes at the point the error occurred? You could. That is how most non-standardized procedures work. However, doing so prevents any automated clean-up from occurring, which adds to the time needed to recover from a problem, detracting from our goals of quick, efficient problem resolution. It also precludes using a standard template for both batch and interactive procedures, which by now you have come to realize is a really valuable tool.

Conclusion

That's it! Hard to believe such simple concepts took so many pages to explain. It just goes to show the power of a few well-placed DCL commands. I have found the use of the techniques described here to be extremely helpful in facilitating quick problem identification and resolution. I hope you find them equally useful.

Your comments and feedback are welcome. Those attached to large denomination bills will get a quicker response.

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

For real world examples of DCL procedure templates, check out these two articles:

- 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