Chapter Thirteen

MANAGING COMMAND PROCEDURES

Introduction

Although previous lessons have covered several OpenVMS features that can be used in writing command procedures, we have not really discussed their design.

Writing simple command procedures can be a straightforward task. However, the building of more complicated command procedures requires the execution of more complicated tasks and the use of some of the sophisticated OpenVMS features previously discussed. Two features of these tasks are:

- A need to test or make decisions about values that exist, and then execute a specific set of tasks based on your conclusions. For example:
  - IF something is true
  - THEN execute task A
  - ELSE execute task B

- A need to handle errors made by the user of the procedure or caused by missing files or information. For example:
  - IF the input file is missing
  - THEN
    - Ask the user to supply it or another file, or
    - GOTO an error handling section of the command procedure that will stop its execution in an orderly manner.

These two features allow you to write command procedures that are easy and efficient to use and prevent corrupted data caused by command procedure failures.

This lesson introduces the topics of ***execution flow*** and ***error handling***.

Objectives

Upon completion of this lesson, students will be able to write and debug complex command procedures. In order to accomplish these tasks a programmer should be able to:

- Design and format command procedures
- Control the flow of command execution
- Handle error conditions
- Use the SuRPAS command procedure template (fal$com:template.com)

Chapter Terms

The topics discussed in this lesson assume that the student is familiar with the following terms:

Conditional branch	An instruction that causes the command procedure to execute certain commands only when a condition is met.
Iteration	Repetition of a group of instructions, usually until a particular condition is met.
Nesting	The placement of a group of statements and/or instructions inside another group of statements and/or instructions.
Recursion	When a routine or procedure invokes itself, usually until a particular condition is met.
Subroutine	A routine designed to be used by other routines to accomplish a specific task.

Command Procedure Guidelines

The concepts for developing command procedures in OpenVMS are similar to developing high-level language programs and routines. This section covers concepts and steps for designing, creating, executing, and testing command procedures:

1. Designing the Command Procedure

There are a number of decisions to make when designing a command procedure. They include:

- The tasks the procedure will perform
- The results the procedure should produce
- The variables your command procedure will use and how will they be loaded
- The conditional actions the command procedure requires and how to test them
- How you will exit from the command procedure

2. Creating the Command Procedure

Creating the command procedure will be a straightforward activity if your design is complete. The command procedure design will feed the creation process and aid in the structuring of the procedure. Just as there are a number of levels or phases to your design, there should be a number of levels or phases to your creation activity. The use of program stubs as part of your creation will aid in organizing how and when you build the pieces of the command procedure. When building a command procedure, use a text editor that you are comfortable with and specify the file type as .COM.

- Prompt for interactive input using the INQUIRE command.
- Determine conditional actions; test them using IF and THEN statements.
- Write program stubs as placeholders for commands and write error messages.
- Decide where you need to exit or quit the command procedure, using EXIT and STOP commands.
- Add comments to the command procedure as you create the task code. Design your comments so that they provide insight to those sections where the coding technique is not obvious.

3. Execute and Test the Command Procedure

Execute and test your command procedure in an organized manner, stepping through the commands and checking the flow of the procedure.

- To begin execution, use the @ sign followed by the name of the command procedure.
- Use the SET VERIFY command to:
  - Display each line of the procedure as it executes
  - Help you locate errors if they occur
- Use the SET PREFIX command to time-stamp a procedure log-file by prefixing each command line with its time of execution.
- Use the SHOW SYMBOL command to determine how the symbols in the procedure are defined.
- Test all possible paths of execution.

4. Modify and Retest the Command Procedure if Necessary

Repeat steps 2 and 3 as necessary; continue testing the program logic by entering valid and invalid commands and by entering commands for program stubs. Finish testing the program logic by exiting from the procedure using the EXIT command.

Add cleanup tasks; begin a cleanup subroutine with a label such as CLEAN_UP that deletes or purges files, restores default settings, closes all open files, and so on.

Use the DCL command SET NOVERIFY after the procedure has been tested and perfected.

5. Add Comments for Readability and Maintainability

It is difficult to add too many comments to a command procedure. Comments should be added both during the building of the command procedure (step 2) and again after testing is complete (step 5).

- Use an exclamation point (!) to indicate a comment line.
- Describe the procedure in detail.
- Describe any parameters that are passed to the procedure.
- Put lengthy comments either at the beginning of the procedure for performance enhancement or at the end (after the EXIT).

A Formatted Command Procedure

The following example illustrates the conventional format for coding a command procedure. Notice the placement of the labels (directly after the dollar sign) and how the comments are formatted. All SuRPAS command procedures must be built using the SuRPAS command procedure template. Please see the section that discusses the template (FAL$COM:TEMPLATE.COM), later in this chapter, to better understand the exact formatting of command procedures.

- The Modification History section is a part of the command procedure template. Always place any modification notes relevant to the code after this section. Do not fill it in manually. TEMPLATE.COM formats this for you and the Code Management System (CMS) supplies the modification history information for you when the code is checked in.
- Indent at least one space for commands such as a SHOW command.
- Labels for loops should begin in the second column.

```
$!   ===============================================
$!  D A I L Y . C O M
$!   ===============================================
$!
$    SET NOVERFIY
$!
$!   -------------MODIFICATION HISTORY---------------
$!    WHO    DATE    RELEASE   SEQ#     FTK#   NOTE#
$!   ===   ======== ======= =====   ======= =====
$!   -----------------------------------------------
$    SET NOON
$!
$!   Initialization Section
$      SET VERIFY
$      SHOW TIME
$!
$!   Main Section - Capture information on the system
$      SHOW USERS/FULL
$      SHOW PROCESS/ALL
$      DIR/GRAND/SIZE=ALL [...]
$!
$!   Find files added and/or changed
$       DIRECTORY/MODIFIED/SINCE=YESTERDAY [...]
$!
$! Cleanup Section
$CLEANUP:
$!
$  EXIT
```

Example 13-1 - A Formatted Command Procedure

Controlling the Flow of Execution

The use of conditional and loop commands when writing a command procedure allows you to control the flow of execution. Using a conditional command, the value of a symbol is tested. The results of this test determine what happens next.

You can write code that will test for the existence of parameters on a command procedure call. If the parameters exist, they will be used; if not, the command procedure will query the user for the information.

These tests are accomplished using DCL commands that control execution flow. In addition, you can write code that allows you to build nested, iterative, and recursive loops.

The following table summarizes conditional commands:

Command	Function
IF	Tests the value of an expression and executes the specified command or commands of the result if the test is true.
GOTO	Transfers control to a different part of the procedure that is identified by a label.
@	Creates a new command level by executing another command procedure.
CALL	Transfers control to a labeled subroutine in the procedure.
GOSUB	Transfers control to a labeled subroutine in the procedure while maintaining local symbol values.

Table 13-1 - Commands used to Control Execution Flow

The IF Command

The IF command tests the value of an expression:

- If the condition is true, the command(s) following the THEN statement are performed.
- If the condition is false,
  - the next DCL command in sequence is performed, or
  - an optional ELSE statement can be performed.

Three Formats for the IF Command

1. The single command line IF-THEN Command

```
$ IF conditional-expression THEN command
```

2. The multiple command line IF-THEN command

```
$ IF conditional-expression
$ THEN [command]
$ [command]
      •
      •
      •
$ ENDIF
```

3. The IF-THEN-ELSE command

```
$ IF conditional-expression
$ THEN [command]
$ [command]
      •
      •
      •
$ ELSE [command]
$ [command]
      •
      •
      •
$ ENDIF
```

The ***conditional-expression*** can consist of one or more numeric constants, string literals, symbolic names, or lexical functions; these are separated by logical, arithmetic, or string operators.

Evaluation of Expressions in IF Commands

Expressions are automatically evaluated during the execution of the command.

- Character strings beginning with alphabetic characters that are not enclosed in quotation marks are assumed to be symbol names or lexical functions.
  - The command language interpreter tries to replace these strings with their current values.
  - The command interpreter does not execute an IF command when it contains an undefined symbol. Instead, it issues a warning message and executes the next command in the procedure.

- Symbol substitution in IF commands is not iterative. Each symbol is replaced only once.

- If you want iterative substitution, precede a symbol name with an apostrophe or an ampersand.

```
$ INQUIRE NCOPY “How many copies do you want?”
$
$ IF
$   NCOPY .LT. 5
$ THEN
$   WRITE SYS$OUTPUT “Copies are printing on Office Printer”
$ ELSE
$   WRITE SYS$OUTPUT “Copies are printing on the Lab Printer”
$ ENDIF
```

Example 13-2 - Evaluating Conditional Expressions

The GOTO Command

The GOTO command is used to transfer control to a line that is **not** the next line in the command procedure. The format of the GOTO command is:

```
$ GOTO label
```

In the GOTO command, control is transferred to a location specified by a ***label*** pointer. The destination label is at another location in the command procedure, and is followed by a colon. (The label attached to the GOTO command is NOT followed by a colon.)

The destination label, a string of up to 255 characters with no embedded blanks, must be the first item on a command line.

When the command interpreter encounters a label, it is entered into a label table.

- The table is allocated from space available in the local symbol table.
- If the command interpreter encounters a label that already exists in the table, the new definition replaces the existing one.
- If you duplicate labels, control is always transferred to the label most recently read by DCL.

If the specified label does not exist in the current command procedure, the procedure cannot continue and is forced to exit.

The following illustrates the GOTO statement and its screen output. Note that the SHOW USERS command is never executed in this example.

```
$ GOTO LUNCH   ! Unconditional transfer of control.
$ SHOW USERS   ! We never execute this statement.
$
$ LUNCH:
$    SHOW DEFAULT
$    SHOW TIME
```

Example 13-3 - The GOTO Statement

```
$ GOTO
   FAL$DISK_USER:[LEVINEJ]
   23-JAN-2002 23:29:17.23
$
```

Example 13-3a - Output from the GOTO Statement

The @ Command

The @ command allows a user to start another command level from within a command procedure or from an interactive command line. The @ command has the following syntax:

```
$ @command-procedure
```

The ***command-procedure*** is executed within the process context of the interactive or batch environment in which it is started. All global symbols apply. Local symbols are only available within the procedure which created them and any nested procedures.

The CALL Command

The CALL command transfers control to a labeled subroutine within a command procedure. It creates a new procedure level as does the @ command. The major structural difference between the CALL and @ commands is that the physical location of the block of statements executed using the CALL command is within the command procedure itself, while the block of statements executed by the @ command is a separate command procedure file.

The CALLing of a labeled subroutine allows you to nest a level of statement execution within another block of executing statements. In the OpenVMS operating system, the following can be nested to a maximum of 32 levels: command procedures (using @), subroutines (using CALL), or a combination of the two. When nesting procedures and subroutines, the local symbols defined within these levels of execution are visible to any commands at an inner nesting level.

The syntax of the CALL command is:

```
$ CALL label [p1 p2 ... p8]
```

The ***label*** designated in the CALL command indicates the location of the beginning of the subroutine. The ***p1*** through ***p8*** parameters are data that can be passed to the subroutine. Parameters are separated by spaces, not by commas.

When you transfer control to the subroutine using the CALL command:

- The symbols P1 through P8 are assigned the values of the assigned arguments.
- Execution then proceeds until an EXIT command is encountered.
- At this point, control is transferred to the command line immediately following the CALL command.

Keep in mind that the amount of space available for ***labels*** is limited. The command interpreter stores labels in the local symbol table. A command procedure that contains many labels and uses many symbols may cause the command interpreter to run out of table space, resulting in an error message.

SUBROUTINE and ENDSUBROUTINE Commands

OpenVMS DCL supports two additional statements that can be used in conjunction with the CALL command. The SUBROUTINE and ENDSUBROUTINE commands can further define the physical boundaries (the beginning and ending points) of an internally structured subroutine. Subroutines built with the SUBROUTINE and ENDSUBROUTINE command pair can only be executed by using the CALL command.

- The SUBROUTINE and ENDSUBROUTINE commands define the beginning and end of the subroutine.
- The SUBROUTINE command must be the first executable statement in a subroutine.
- The ENDSUBROUTINE command functions as an EXIT command from a nested procedure call if no EXIT command is specified in the procedure.
- The SUBROUTINE and ENDSUBROUTINE commands cannot be abbreviated to fewer than four characters.
- If the SUBROUTINE command is encountered in the normal execution flow of the procedure without having been invoked by a CALL command, all commands following SUBROUTINE are skipped until the corresponding ENDSUBROUTINE command is encountered.
- If a subroutine does not end with ENDSUBROUTINE, a subsequent CALL command may not be able to find label destinations.

The EXIT command and ENDSUBROUTINE command can both be used to end subroutines that are invoked by the CALL command. The differences between these two commands are summarized below:

- The EXIT command can return a status to the DCL prompt. It is stored in the symbol $STATUS.
- The ENDSUBROUTINE command is just a return call with no status and can only be used in conjunction with the SUBROUTINE command.

This procedure execution example (a code fragment) uses the SUBROUTINE and ENDSUBROUTINE commands. The subroutine is called and passed the two numbers 1 and 3. When subroutine execution is completed, the result is displayed on the screen, using SYS$OUTPUT.

```
$ CALL FIND_SUM 1 3
$!
$ WRITE SYS$OUTPUT SUM
$!
$DONE:
$ EXIT
$!
$  FIND_SUM:SUBROUTINE
$!
$ SUM == P1 + P2
$ EXIT
$  ENDSUBROUTINE
```

Example 13-4 - The CALL Statement

Can you explain why the result is 13 and not the expected answer of 4?

The GOSUB Command

The GOSUB command transfers control to a labeled subroutine in a command procedure without creating a new procedure level. We usually refer to GOSUB subroutines as “local” subroutines. Any labels and local symbols defined at the current command procedure level are available to a subroutine invoked with the GOSUB command.

The syntax for the GOSUB command is:

```
$ GOSUB label
```

The ***label*** indicates the location in the command procedure where the local subroutine begins execution. As with the CALL command, it is important to keep in mind that the amount of space available for labels is limited. The command interpreter stores labels in the local symbol table. A command procedure that contains many labels and uses many symbols may cause the command interpreter to run out of table space, resulting in an error message.

The GOSUB command can be nested up to a maximum of 16 levels deep within a procedure level. A GOSUB subroutine is terminated using the RETURN command. When the RETURN command is executed, control is transferred to the command immediately following the GOSUB command that invoked the local subroutine. The RETURN command can be executed with an optional attached status value.

Although the use of global symbols is the same for the CALL and GOSUB commands, there are some differences regarding local symbols:

- Local symbols created by a CALL procedure are not available after the procedure exits.
- Local symbols created in a GOSUB routine are available after the RETURN is executed.

The following four examples demonstrate uses of subroutines within a command procedure. Subtle differences among them affect the results of their calculations. Build these four command procedures at your workstation. Is the correct answer 55 or 10? Which one(s) result in the correct answer? **Create these simple command procedures in EVE and determine the correct result for each.**

```
$!  A.COM
$!
$ A = 5
$ B = 5
$!
$ CALL ADD_SUM A B
$ SHOW SYMBOL C
$ EXIT
$!
$  ADD_SUM:SUBROUTINE
$ C == P1 + P2
$ EXIT
$  ENDSUBROUTINE
```

Example 13-5 - Using CALL, Try #1

```
$!  B.COM
$!
$ A = 5
$ B = 5
$!
$ GOSUB ADD_SUM
$ SHOW SYMBOL C
$ EXIT
$!
$  ADD_SUM:
$ C = A + B
$ RETURN
```

Example 13-6 - Using GOSUB

```
$!  C.COM
$!
$ A = 5
$ B = 5
$!
$ CALL ADD_SUM 'A' 'B'
$ SHOW SYMBOL C
$ EXIT
$!
$  ADD_SUM:SUBROUTINE
$ C == P1 + P2
$ EXIT
$  ENDSUBROUTINE
```

Example 13-7 - Using CALL, Try #2

```
$!  D.COM
$!
$ A = 5
$ B = 5
$!
$ CALL ADD_SUM 'A' 'B'
$ SHOW SYMBOL C
$ EXIT
$!
$  ADD_SUM:SUBROUTINE
$ C == 0 + P1 + P2
$ EXIT
$  ENDSUBROUTINE
```

Example 13-8 - Using CALL, Try #3

Nested, Iterative, and Recursive Procedures

You can combine all of the DCL control commands to produce conditional execution of your command procedures. Furthermore, you can use the techniques of iteration, nesting, and recursion to write complex command procedures.

Iteration is the repetition of a group of instructions. To write iterative procedures, use a looping structure.

Be careful not to create an infinite loop!
If you accidentally create an infinite loop, press the Ctrl/Y key to stop the procedure. Once the procedure stops, enter the SET VERIFY command and re-execute the procedure to try to find the cause of the problem.

```
$ A = 5
$ COUNT = 1
$ LOOP:
$ COUNT = COUNT + 1
$ A = A + 1
$ IF COUNT .LT. 5 THEN GOTO LOOP
$ SHOW SYMBOL A
```

Example 13-9 - Using Iteration

Nesting is the placement of a group of statements or instructions within another group of statements or instructions. If your application can be separated into several parts, it is often helpful to write a command procedure to implement each of the parts. You can then nest the command procedures within the main procedure. This nesting technique can decrease the complexity of your main procedure, making it easier for you to debug and maintain your application.

When you nest command procedures, you must be aware of the restrictions that apply. Some of these restrictions are characteristics that are changed when new procedures are invoked.

```
$ B = 0
$  OUTER_LOOP:
$    A = 0
$    B = B + 1
$    INNER_LOOP:
$         A = A + 1
$         SHOW SYMBOL A
$         SHOW SYMBOL B
$         IF A .LT. 3 THEN GOTO INNER_LOOP
$    IF B .LT. 5 THEN GOTO OUTER_LOOP
$ EXIT
```

Example 13-10 - Using Nesting

Recursion involves a routine or procedure that calls itself. In certain situations you may decide to have your procedure call itself recursively, instead of calling another procedure or transferring control to different parts of the procedure.The same considerations that apply to nesting procedures also apply to recursive procedures.

```
$!  RECUR.COM
$!
$ IF P1 .EQ. " " THEN GOTO LOBBY
$ IF P1 .EQ. 1 THEN GOTO PENTHOUSE
$ IF P1 .EQ. 2 THEN GOTO ATTIC
$ EXIT
$!
$ LOBBY:
$ WRITE SYS$OUTPUT "I am in the lobby"
$ @RECUR 1
$ EXIT
$!
$ PENTHOUSE:
$ WRITE SYS$OUTPUT "I am in the penthouse"
$ @RECUR 2
$ EXIT
$!
$ ATTIC:
$ WRITE SYS$OUTPUT "I am in the attic"
$ EXIT
```

Example 13-11 - Using Recursion

Error Handling

It is important to be able to detect the errors that occur in your command procedures. There are conventional ways to handle error conditions or ***Control-Y interrupts*** when a command procedure is executed. A Control-Y interrupt occurs when the user presses the $\boxed{\text{Ctrl/Y}}$ key combination.
There are two basic ways to handle errors:

Reactive

- Accept the system default action
- Specify your own action using the value of $STATUS

Preventive

- Override the default error handling using one of the following:
  - ON WARNING (catches WARNING, ERROR, and SEVERE errors)
  - ON ERROR (catches ERROR and SEVERE errors)
  - ON SEVERE_ERROR (catches SEVERE errors)
  - SET NOON
  - Specify your own action using $STATUS
- Use qualifiers in the file I/O statements to capture frequently occurring errors:
  - /END
  - /ERROR
- Handle interrupts using one of the following:
  - ON CONTROL_Y
  - SET NO CONTROL=Y

Status Check

During execution of a command procedure, the command interpreter checks the condition code returned from each command or program that executes. The condition code can be generated by the system or supplied by the user in the EXIT or RETURN command.

- The system places condition codes in the global symbol $STATUS.
- The severity of the condition code is represented in the three low-order bits of $STATUS. If the low-order bit is set, the condition code represents a level of success. If the low-order bit is clear, the condition code represents a level of failure.
- This severity level is also represented by the global symbol $SEVERITY.
- Values for $SEVERITY, their meanings and default actions are as follows:

0 = Warning (W)	---	Continue processing
1 = Success (S)	---	Continue processing
2 = Error (E)	---	Exit
3 = Information (I)	---	Continue processing
4 = Severe or fatal error (F)	---	Exit

```
$!   Continue if status is S or I.  Exit on status of E or F
$!   If warning, increase previous defined counter, issue message
$!   and continue.
$!
$ RUN TESTPROG.EXE
$ IF $SEVERITY .EQ. 0 THEN GOSUB WARNING_MSG
$ GOTO REST_OF_PROC
$!
$ WARNING_MSG:
$ WARN_COUNTER = WARN_COUNTER + 1
$ WRITE SYS$OUTPUT " ' 'WARN_COUNTER' warnings(s) have occurred"
$ RETURN
    •
    •
    •
```

Example 13-12 - Checking the Severity Status

The ON Command

The ON command specifies an action to be taken when an error condition is encountered in a command procedure. The specified action is performed only if the command interpreter is enabled for error checking or Control-Y interrupts. The syntax of the ON command is:

```
$ ON condition THEN command
```

The ON command temporarily overrides existing error handling. After one execution of the command, default error handling is reset. If you wish your ON command to remain in effect, you must reissue it after handling the error. The ***default condition*** is:

```
$ ON ERROR THEN EXIT
```

The command interpreter continues when a warning occurs, and executes an EXIT command when an error or severe error occurs.

Exception to the default ON action:
if you use a GOTO to a nonexistent label, the procedure issues a warning message and exits.

The ON command must be issued at each command level requiring level handling; it is valid only at the current level.

The specified ***command*** is executed when errors equal to or greater than the specified level of error occur. The action specified by the ON command applies only within the command procedure in which the ON command is executed.

The ON command also provides a way to define an action to be taken for a Control-Y interrupt that occurs during execution of a command procedure. The default Ctrl/Y action is to prompt for command input at the Ctrl/Y command level.

The example on the next page illustrates error handling in a command procedure. The command procedure begins by saving the initial setting of the SET MESSAGE command and disabling the message field reporting for the duration of the command procedure. The saved MESSAGE setting will be restored prior to exiting the command procedure.

```
$!  CHOICES.COM
$! This command procedure presents the user with a menu of
$! options.  The user can translate a logical name, find out
$! whether a device exists on the system, or exit from the
$!  command procedure.
$!
$   SAVE_MSG = F$ENVIRONMENT("MESSAGE")
$   SET MESSAGE/NOFACILITY/NOSEVERITY/NOIDENTIFICATION/NOTEXT
$!
$   DISPLAY_MENU:
$   TYPE SYS$INPUT

          Your options are:

          TRANSLATE     translate a logical name

          CHECK_DEV     See if a device exists on the system

          END           Exit from the procedure

$   INQUIRE CHOICE "Enter your option "
$!
$   ON WARNING THEN GOTO ERR_MSG
$   GOTO 'CHOICE'
$!
$   TRANSLATE:
$     ON ERROR THEN GOTO END
$     @FAL$DISK_USER:[9RB]TRANSLATE_LNM.COM
$     GOTO DISPLAY_MENU
$!
$   CHECK_DEV:
$     ON ERROR THEN GOTO END
$     @FAL$DISK_USER:[9RB]CHECK_DEVICE.COM
$     GOTO DISPLAY_MENU
$!
$   ERR_MSG:
$     WRITE SYS$OUTPUT " "
$     WRITE SYS$OUTPUT "Invalid option - Please try again!"
$     GOTO DISPLAY_MENU
$!
$   END:
$     SET MESSAGE 'SAVE_MESSAGE'
$     EXIT
```

Example 13-13 - Using Error Handling

The SET NOON Command

The SET NOON (pronounced *set no on*) command disables any error checking performed by the system (turns off the ON command). The command interpreter continues to place the status code value in $STATUS and the severity level in $SEVERITY, but does not perform any action based on these values.

The syntax for the SET NOON command is:

```
$ SET [NO]ON
```

The command procedure continues to execute no matter how many errors are returned. You cannot use the ON command `(ON condition THEN command)` to handle errors when SET NOON is in effect. If you use SET NOON in a command procedure that executes another procedure, the default (SET ON) is established while the second procedure executes.

There are a number of reasons for using the SET NOON command. They include:

- Deleting scratch files as a part of a cleanup operation. If the system finds no temporary files, you don't want that to be regarded as an error.
- Keeping unexpected conditions from aborting system startup during the OpenVMS system startup procedure.

As with the ON command, the SET NOON command is valid at the current command level only. Error handling is restored after an EXIT or SET ON command.

ON CONTROL_Y Interrupts

You can use the ON command to handle Ctrl/Y interrupts. Typically, when a user presses the Ctrl/Y key combination the command procedure will exit. Sometimes this is okay, sometimes it isn't. This is especially true if files are open because they won't be closed. The ON CONTROL_Y THEN command allows the procedure designer to define what happens when a user presses the Ctrl/Y key combination. The syntax of this form of the ON command is:

```
$ ON CONTROL_Y THEN command
```

Unlike the ON command, the ON CONTROL_Y command remains in effect until:

- It is changed with another CONTROL_Y command
- It is overruled by the `SET NOCONTROL=Y` command
- The user exits the command level.

In addition, the `SET NOON` does not disable the ON CONTROL- command.

Once the ON CONTROL_Y command has been issued any pressing of the Ctrl/Y key will cause the command to be executed rather than the default; the command procedure is aborted.

The ON CONTROL_Y is effective only for the current level of command procedure. If a lower level command procedure receives a Ctrl/Y and doesn't handle it, and you have a handler at a higher level, you exit from the lower level procedure back to the higher one and pick up at the next command.

To turn off the Ctrl/Y handler completely, issue the command:

```
$ ON CONTROL_Y THEN CONTINUE
```

The SET NOCONTROL=Y Command

The SET NOCONTROL=Y command disables the Ctrl/Y key combination at all command levels. The syntax for the command is:

```
$  SET  [NO]CONTROL=Y
```

Use the SET NOCONTROL=Y command with caution because it remains in effect after you exit the command level at which you entered the command.
You must set it back manually.

This will not affect you at terminal level, but it does affect all levels of your command procedure. Remember if the Ctrl/Y key is disabled and you get into an endless loop you will have no way to get out! The INTERRUPT message will still be displayed, but no interruption will take place. Try to use this command only in portions of your command procedure where you don't want to exit.

SET NOCONTROL=Y also disables the Ctrl/C key combination for all commands and programs that do not have special action routines responding to Ctrl/C.

Handling File I/O Errors

Use the /ERROR qualifier with the OPEN, READ/END, WRITE, and CLOSE commands to handle errors encountered while working with files. By using the /ERROR qualifier control is passed to the specified label when an error occurs and the system error message is not displayed. This allows you to manage error and end of file situations within the command procedure.

In the example which follows the use of the /ERROR and /END qualifiers allow the command procedure to deal with file errors.

```
$   START:
$         OPEN/READ/ERROR=NO_FILE  CALLS  [9RB]:CALLS.LOG
$!
$   READ_LOOP:
$   READ/END=DONE  CALLS  DATA_LINE
$   WRITE  SYS$OUTPUT  DATA_LINE
$   GOTO  READ_LOOP
$!
$   NO_FILE:
$         WRITE  SYS$OUTPUT  “Call  log  doesn’t  exist.”
$         EXIT
$!
$   DONE:
$         CLOSE  CALLS
$         EXIT
```

Example 13-14 - Handling File I/O Errors

The Command Procedure Template

The SuRPAS command procedure template, TEMPLATE.COM, is located in the FAL$COM directory. This template is used for all command procedures built for SuRPAS.

When creating a new SuRPAS command procedure copy the template and rename it the name of your new command procedure. An example of the DCL COPY command to achieve this is found below:

```
   $ COPY FAL$COM:TEMPLATE.COM NEW_PROCEDURE.COM
```

where ***NEW_PROCEDURE.COM*** is the name of the command procedure that you are about to create. When you open this new file with the AMS editor you will find the template shown in Example 13-15, on the next page.

When using this template, remember that the modification history is filled in by the Code Management System (CMS) when the command procedure is checked in (see Chapter 22 for a more detailed discussion of CMS). If you need to include a more in-depth discussion of a specific modification, add a set of comments after the Modification History section.

The TIMESTAMP commands in the template execute the SuRPAS time stamp command procedure. This procedure places time stamps in the job log when your command procedure runs. This assists the night duty personnel when they are isolating the location of a problem (see Chapter 24 for a discussion of the nightly process and troubleshooting).

```
$ ! ============================================================
$ ! xxxxxxxx.COM
$ ! ============================================================
$ ! brief purpose of this .COM
$ !
$ !
$ ! NIGHT DUTY:
$ !
$ !
$ SET NOVERIFY
$ !
$ !      ------------------MODIFICATION HISTORY---------------
$ !      WHO    DATE    RELEASE  SEQ#    FTK#  NOTE#
$ !      ===  ========  ======= =====  ======= =====
$ !      ------------------------------------------------ <EOM>
$ !
$ !     $ TIMESTAMP event,image,tagline
$ !
$ !            where:  event = event name running
$ !                    image = program name or dcl verb
$ !                  tagline = 20 chars free form comments
$ !
$ !             eg; $ TIMESTAMP JOBDAILY,JXMOVBAT,MOVE BATCHES
$ !
$ SET NOON
$ ON ERROR THEN GOTO ERRTRP
$ !
$ TIMESTAMP xxxxxxxx,,*** BEGIN EVENT ***
$ !
$ ! ============================================================
$ !  explanation of next step
$ !

$ ! ============================================================
$ !  Print reports automatically
$ !
$      TIMESTAMP xxxxxxxx,JXAUTPRI,...BEG AUTPRI
$ !
$      RUN FAL$PRGLIB:JXAUTPRI.EXE
$ !
$     TIMESTAMP xxxxxxxx.JXAUTPRI,...END AUTPRI
$ !
$ ! ============================================================
$ ENDTRP:
$ !
$ !  This is normal exit block
$ !
$       TIMESTAMP xxxxxxxx,,*** END EVENT ***
$ !
$       EXIT
$ !
```

Example 13-15 - TEMPLATE.COM Command Procedure Template

Summary

Concepts

Command Procedure Guidelines

Easily read and maintained command procedures follow an organized development cycle that includes:

- Design
- Formatting
- Commenting
- Testing

Controlling the Flow of Execution

Control the flow of command procedures by using:

- IF
- GOTO
- CALL
- GOSUB

The techniques of iteration, nesting, and recursion enable you to write complex command procedures.

Error Handling

The ON command allows you to determine the action the system takes when an error equal to or greater than a specified severity level occurs.

The SET NOON command tells the system not to respond to error conditions. The command procedure must detect and handle error conditions.

A command procedure can trap or disable Ctrl/Y interrupts using the ON CONTROL_Y or SET NOCONTROL=Y.