Chapter Eighteen

CONDITIONALS AND LOOPS

Introduction

The example code in Chapters Fifteen through Seventeen demonstrate sequential execution of program statements. In each example and code snippet the statements are executed in sequence from the top to the bottom as they are written in the source code file or printed on a page.

In the analysis of problems to be solved and in the construction of Fortran programs, two other program structures are needed. These are the selection structure (often referred to as the conditional or branch) and the repetition structure (often referred to as the loop).

With the introduction of these structures, our programs are going to become more complex, increasing the likelihood that mistakes will be made. To help in debugging this code we will complete our debugger discussion by learning about breakpoints and watchpoints.

Objectives

To implement non-sequential activities in Fortran, a programmer should be able to:

- Design code that implements conditionals and branching
- Design code that implements iterative and conditional loops
- Debug complex code using breakpoints and watchpoints

Transfer Structures

Overview

It is frequently necessary to deviate from the normal execution sequence within a program and explicitly transfer control to another section of code where execution will continue. This is accomplished with a number of explicit transfer statements.

This section discusses:

- The GOTO statement
- Statement labels
- The CONTINUE statement
- The computed GOTO statement

The GOTO Statement

The GOTO statement transfers control to a label located at another specific point in the program code. This statement is normally avoided as it is considered poor programming technique. Exceptions include when the use of the GOTO will reduce complexity and increase readability. The statement has the following syntax:

```
    GOTO label
```

This directs the sequence of execution of statements to be interrupted; the next statement to be executed is located at the line specified by the ***label*** referenced in the GOTO statement. The labeled statement may appear anywhere in the program unit.

The GOTO statement may appear in any of the conditional or loop structures discussed in this chapter. Typically, one of the assumptions that is made when using a conditional or loop structure is that (after one of the alternative statement sequences is completed) the execution sequence will *always* continue with the statement following the end of the structure. With the use of a GOTO statement, however, we can no longer make this assumption. Therefore, extra care is required in order to maintain program reliability, and extensive commenting should be used to explain the alternative paths of execution that may be taken depending on the results of each conditional test.

Fortran imposes a restriction which should be noted. In a conditional structure, Fortran prohibits explicit transfer of control into any of the alternative blocks of the conditional. Branches into loop structures are also forbidden. However, a jump from within a conditional structure to the END IF statement is permitted, but not permitted by the SuRPAS coding standards.

Statement Labels

A statement label creates a location to which a transfer of execution can branch. The label consists of one or more numbers written at the left of the statement (usually in the first five columns of the statement line). A label must be on a statement line containing a legal Fortran executable statement. A statement should not be given a label unless it is referenced by an explicit control statement (e.g., GOTO) or is a FORMAT statement referenced by an I/O statement. Labels on non-executable statements are unnecessary and should be avoided.

The use of labels in SuRPAS programs must follow the SuRPAS Fortran coding standards. The standards specific to Fortran labels can be found in Chapter Fifteen. Note that these standards require all labels to be attached either to a CONTINUE statement or to a FORMAT statement.

The following are examples of valid Fortran statements with labels, but that do not follow the SuRPAS standard. In the next section, the CONTINUE statement is used to create Fortran labels that follow the SuRPAS standard.

```
4    Mfund = Mfund + 1
10   READ *, A, B
375  IF (ICount .GT. JCount)   ITotal = .TRUE.
```

The CONTINUE Statement

The CONTINUE statement designates a point in the program where a label is needed to terminate a branch or GOTO. PFPC coding standards do not allow the attachment of a label to an executable statement. The CONTINUE statement allows the branch to occur while meeting these standards. Code is more easily modified when labels are attached to CONTINUE statements as opposed to assignment statements (the order and location of which may change).

The CONTINUE statement has no execution value. It is normally used to terminate loops or blocks of repetition code, and to receive control from a branch. CONTINUE statements are usually written with a label.

These statements demonstrate the use of the CONTINUE statement and the correct way to create the examples of labels found in the previous section:

```
4    CONTINUE
     Mfund = Mfund + 1
10   CONTINUE
     READ *, A, B
375  CONTINUE
     IF (ICount .GT. JCount)   ITotal = .TRUE.
```

The Computed GOTO Statement

The computed GOTO statement allows a programmer to select one of a number of possible branches (labels) based on a supplied integer expression. The syntax of the computed GOTO statement is as follows:

```
GOTO(label_1,label_2,...label_n)
integer_expression
```

The *labels* provided in the computed GOTO statement are located elsewhere in the program unit, and the result of the *integer_expression* dictates which of the labels will receive the transfer of control. Execution of this statement results in a branch to the statement labeled *label*$_1$ if the value of *integer_expression* is 1, to *label*$_2$ if the value of *integer_expression* is 2, etc.

If the value of the integer expression is less than 1 or greater than the count of the last label supplied, none of the branches is selected and control proceeds to the statement immediately following the computed GOTO.

The following computed GOTO example shows the transfer effect based on the value of the integer expression `Rate_Code + 1`:

```
GOTO (30, 45, 60, 10) Rate_Code + 1
```

In this example, the GOTO results in a branch to label 30 if the resulting value of the expression is 1, to label 45 if the resulting value is 2, to label 60 if the resulting value is 3, and to label 10 if the resulting value is 4.

Conditional Structures

Overview

Conditionals and branches permit us to select and execute specific blocks of code while skipping others.

This section discusses:

- Logical IF statements
- Logical expressions
- IF ... THEN ... ELSE statements

Logical IF Statement

The logical IF statement is used only when a single statement needs to be executed (or skipped) as the result of a true conditional test. In this case, the executable statement is placed on the same line as the IF statement, immediately following the conditional test. The syntax of the Logical IF statement is as follows:

```
IF (logical_expression) executable_statement
```

When the *executable_statement* (rather than the keyword THEN) follows the *logical_expression*, no END IF statement is needed. It is important to note that the only effect of the logical expression in this statement is to determine whether or not the *executable_statement* will be executed. The statement following the IF statement will be executed in either case (unless the *executable_statement* is a GOTO statement).

Logical Expressions

The most important component of a conditional structure is the logical expression. In its simplest form, a logical expression is a relational expression, consisting of a pair of expressions separated by a relational operator (refer to Chapter 16 - Handling Data in Fortran). The format of a simple logical expression is:

$$\texttt{Expression}_1 \ \ \texttt{Relational\_operator} \ \ \texttt{Expression}_2$$

Each of the expressions may be any valid expression of type integer, real, or double precision. The logical expression may also consist of a single expression that has a logical result (true or false).

The following are some examples of valid logical expressions:

```
Temperature .LT. 32
.TRUE.
ICount .GT. 1
ABS(Rate) + 3 .LE. 1.0E-6
```

The Block IF Statement

A common form of the conditional structure is the block IF structure. This structure specifies that a block of code will be executed if a certain logical expression is true. The block IF structure has the following syntax:

```
IF (logical_expression) THEN
     statement
     statement
     ...
ENDIF
```

If the *logical_expression* is true, the program executes the *statements* in the block between the IF and ENDIF statements. If the *logical_expression* is false, then the program skips all of the *statements* in the block between the IF and ENDIF statements and executes the statement immediately following the ENDIF.

The `IF (logical_expression) THEN` is a Fortran statement that must be written on a single line, and the statements to be executed must occupy separate lines below it. An ENDIF statement must follow on a separate line. There can not be a statement label on the line containing the ENDIF statement. For readability, the block of code between the IF and the ENDIF statements is indented by two or three columns (spaces).

Always indent the body of a block IF structure by a TAB or three spaces to improve the readability of the code.

The IF...THEN...ELSE Statement

In the block IF structure, a block of code is executed if the controlling logical expression is true. If the controlling expression is false, all of the statements in the block structure are skipped.

There are times when it may be desirable to execute one set of statements if a condition is true and a different set of statements if the condition is false. An ELSE clause can be added to the block IF structure to accomplish this differentiation. The following is the syntax of the IF...THEN...ELSE statement:

```
IF (logical_expression) THEN
     statement
     statement
     ...
ELSE
     statement
     statement
     ...
ENDIF
```

If the *logical_expression* is true, the program executes the statements in the block between the IF statement and ELSE clause and skips the statements between the ELSE clause and ENDIF statement. Once these statements have been executed, the program will jump to the statement immediately following the ENDIF.

If the *logical_expression* is false, the program executes the statements in the block between the ELSE clause and the ENDIF statements, having skipped the set of statements between the IF statement and the ELSE clause. The next statement to be executed is the one immediately following the ENDIF statement.

The ELSE and ELSEIF Clauses

There is another expansion of the block IF structure that allows a number of conditions to be tested and different blocks of code to be executed based on the results of each test. This capability is supported by both the ELSE and ELSEIF clauses. The syntax of the expanded IF...THEN...ELSE structure is:

```
IF (logical_expression1) THEN
     statement
     statement
     ...
ELSEIF (logical_expression2) THEN
     statement
     statement
     ...
ELSE
     statement
     statement
     ...
ENDIF
```

If *logical_expression*$_1$ is true, the program executes the statements in the block between the IF statement and the ELSEIF clause and skips the statements between the ELSEIF clause and the ENDIF statement. The next statement to be executed is the one immediately following the ENDIF statement.

If *logical_expression*$_1$ is false, the program tests *logical_expression*$_2$. If *logical_expression*$_2$ is true, the program executes the statements in the block between the ELSEIF and the ELSE clauses and skips the sets of statements between the IF statement and the ELSEIF clause and between the ELSE clause and the ENDIF statement. The next statement to be executed is the one immediately following the ENDIF statement.

If *logical_expression*$_1$ and *logical_expression*$_2$ are false, the program executes the statements in the block between the ELSE clause and the ENDIF statement and skips the first two sets of statements between the IF statement and the ELSE clause. The next statement to be executed is the one immediately following the ENDIF statement.

Each ELSE and ELSEIF clause must occupy a line by itself. There can be no labels on a line containing an ELSE or an ELSEIF clause. A block IF structure can have a number of ELSEIF clauses, but only one ENDIF statement.

The Count-based Loop - Controlled Repetition

Overview

It is difficult to do any serious programming without encountering groups of statements that are to be executed repeatedly. A program which reads a single set of data values, performs a calculation, and prints a single set of results is a rare occurrence; more commonly programs consist of sequences of statements which are executed repeatedly.

For many applications, the number of repetitions of a group of statements is known in advance. Often the repetition is associated with a linear array and the calculation is performed once for each array element. These count-based repetitions are supported by the Fortran DO loop.

This section discusses:

- DO loops
- Implied DO loops
- Nested DO loops

The DO Loop

The DO loop consists of a DO statement containing the necessary information to control a repetition, followed by the group of statements to be executed repeatedly. The loop ends with a labeled CONTINUE statement or an ENDDO statement. When writing SuRPAS program code, the DO...ENDDO form of the DO loop is preferred to the DO...CONTINUE form.

The DO statement which initiates the loop, contains an optional terminating label and an optional iteration expression. The optional terminating label is only included when using the CONTINUE form of the DO loop. The optional iteration expression can be omitted when creating a DO...ENDDO loop. The following is the syntax of the DO loop that terminates with a labeled CONTINUE:

```
     DO terminating_label_nnn iteration_expression
          statement
          statement
          ...
nnn  CONTINUE
```

The ***terminating_label_nnn*** following the DO, identifies the labeled CONTINUE statement that terminates the loop (***nnn***). The ***iteration_expression*** contains an index, an initial value, a limiting value, and an optional increment. The syntax of the *iteration_expression* is:

```
     index = initial_value, limit_value, [increment]
```

The ***index*** is a variable that increments each time through the loop. The initial value of the *index* is dictated by the ***initial_value*** in the expression and iterates until it reaches the ***limit_value***. Usually, the index, initial value, and limit value are of type integer (although they are permitted to be of type real or double precision). The *initial_value* can be smaller than the *limit_value* and the optional ***increment*** can be a negative number. This allows the user to create a loop that runs “backward”, or decrements.

The steps executed in a DO loop are described below:

1. Set the index to its initial value.
2. Compare the index to the limit value. If the index exceeds the limit, terminate the loop.
3. Execute the block of statements.
4. Increment the index by the increment value (defaults to 1) and go to step #2.

The DO...ENDDO loop syntax is:

```
     DO
          statement
          conditional_statement EXIT
          statement
          ...
          conditional_statement CYCLE
          ...
     ENDDO
```

Note that in the DO...ENDDO form of the DO loop there is no iteration expression to control the loop activity. Fortran on Alpha processors supports the use of the EXIT and CYCLE statements for loop control. The EXIT statement terminates execution of the loop while the CYCLE statement (discussed later in this chapter) terminates execution of the current iteration of the loop. *The DO...ENDDO form should be used when writing SuRPAS program code, rather than the DO...CONTINUE form.*

Conditional Exit from a Loop using the GO TO Statement

The GOTO statement is an acceptable method for conditional exit from a DO loop, especially from an inner structure within a nested DO loop. The results of a conditional test may determine that there is no purpose to remaining in a nested portion of a DO loop or the entire loop structure. In this situation, the ability to branch outside of the loop structure is achieved using the **GOTO ... *label*** sequence of statements.

Implied DO Loops

Certain Fortran statements may contain an array with a list of elements that are addressed consecutively. These statements can support iteration through the array elements and have a notation analogous to the iteration expression of a DO statement. This notation has the following syntax:

```
variable (index), index = initial_value,
                          limit_value, [increment]
```

The *variable* consists of a sequence of items of any form that is permitted in the list. The *index, initial_value,* and *increment* are subject to the same rules that apply to DO loops. If the *increment* is omitted, the default value is 1.

Nested Iteration

Some common repetitive tasks consist of single loops that are totally nested or embedded inside other loops.

An implied DO loop may be included in a statement inside a DO loop or in a list or array variable within an outer implied DO loop.

Conditional Loops

Overview

Modern high-level programming languages have structures that permit conditional looping (the repetition of a block of statements for an indefinite number of times), as long as a particular condition remains true.

This section discusses the DO WHILE loop and the CYCLE statement.

The DO WHILE Loop

Fortran loop structures permit the execution of a sequence of statements more than once. The DO WHILE loop is a **conditional loop**. The difference between the iterative loop (discussed in earlier sections) and the conditional loop is the way the loop is controlled. The code in a conditional loop is repeated an indefinite number of times until a user-specified condition is satisfied.

The DO WHILE loop is a block of statements that is repeated indefinitely as long as a particular condition is satisfied. The syntax of the DO WHILE loop in Compaq Alpha Fortran is as follows:

```
[name:]DO WHILE (logical_expression)
    statement
    statement
    ...
ENDDO
```

The block of statements between the DO and the ENDDO is repeated indefinitely until the *logical_expression* becomes true. At that point, execution control is transferred to the statement immediately following the ENDDO statement. The optional **name** construct allows the programmer to assign unique names to DO loops. When building nested DO loops the *name* construct can be used along with the CYCLE statement to identify which level of current loop iteration is being terminated. *The name construct is not supported in SuRPAS coding.*

The CYCLE Statement

The CYCLE statement interrupts the current execution cycle of the specifically named DO structure; if none is named, then the CYCLE statement interrupts the innermost DO structure. The syntax of the CYCLE statement is as follows:

```
CYCLE [name]
```

When a CYCLE statement is executed, the following occurs:

1. The current execution cycle of the named or innermost DO structure is terminated. If a DO structure name is specified, the CYCLE statement must be within the range of that DO structure.
2. The iteration count (if any) is decremented by 1.
3. The DO variable (if any) is incremented by the value of the increment parameter (if any).
4. A new iteration cycle of the DO structure begins.

Any executable statements following the CYCLE statement, including labeled termination statements, are not executed. A CYCLE statement can be labeled, but it cannot be used to terminate a DO structure.

The following is an example of the use of the CYCLE statement:

```
DO I = 1, 10
   A(I) = C + D(I)
   IF (D(I) < 0) CYCLE
   A(I) = 0
ENDDO
```

Miscellaneous Debugging Features

Overview

The OpenVMS symbolic debugger allows the programmer to watch variables and code, and set conditions under which that code will be suspended. At any time the programmer can examine any portion of his source code to remind him of the executable flow or to get specific line numbers.

This section discusses:

- Viewing your source code
- Setting watchpoints
- Setting tracepoints

Suspending Execution When Data Values Change

You can direct the debugger to watch a specified entity and notify you if its value is modified. The debugger suspends program execution when the specified entity is modified, allowing you to enter debugger commands to determine why and how the entity (for example, a variable) was modified.

Watchpoints

A watchpoint is a location monitored by the debugger so that it can inform you when your program has changed the location's contents.

- When you debug a program, you can set a watchpoint on a variable
- Watchpoints are monitored continuously. Therefore, you can determine whether program locations are being modified during program execution.

The commands used to control watchpoints are summarized as follows:

SET WATCH	Defines the locations to be monitored.
SHOW WATCH	Displays the locations currently being monitored.
CANCEL WATCH	Disables monitoring of the specified locations

Setting Watchpoints

Use the SET WATCH command to establish watchpoints. When the specified variable is modified, the debugger suspends program execution, displays the address of the instruction and the old and new contents of the variable, and prompts for a command. The syntax of the command is:

```
SET WATCH[/qualifier] address-expression [, ...]
  [WHEN (conditional-expression)]
  [DO (command[;p...])]
```

The ***address-expression*** parameter specifies the location(s) at which a watchpoint is to be set. Some qualifiers can be used instead of an address.

The ***conditional-expression*** parameter is a language expression that is evaluated when the watchpoint is triggered. If the expression does not evaluate to .TRUE., the watchpoint is not activated.

The ***command*** parameter represents any debugger command(s) that you want the debugger to execute as part of the DO command sequence when the watchpoint is triggered.

The SET WATCH command accepts optional WHEN and DO clauses. The example below uses the following source code to demonstrate the WATCH command:

```
1:     PROGRAM  MAIN
2:     J = 0
3:     I = J
4:     CALL  SUB1  (I)
5:     J = 2
6:     END
```

```
DBG>  SET  WATCH  J;  GO
watch  of  MAIN\J  at  MAIN\%LINE  5
5:        J = 2
    old  value:  0
    new  value:  2
break  at  MAIN\%LINE  6
6:        END
DBG>
```

Example 18-2 - Using the SET WATCH Command

Displaying Watchpoints

Use the SHOW WATCH command to display currently active watchpoints.

```
DBG>  SHOW  WATCH
watchpoint  of  MAIN\J
DBG>
```

Example 18-3 - Using the SHOW WATCH Command

Cancelling Watchpoints

Use the CANCEL WATCH command to cancel a currently established watchpoint. For example, enter the following command to cancel the watchpoint at location J:

```
DBG>  CANCEL  WATCH  J
```

SET WATCH Command Qualifiers

The following table describes the SET WATCH command qualifiers that affect watchpoint action.

/AFTER:*n*	Takes watch action after the ***n***th time the designated watchpoint is encountered.
/INTO	Monitors a nonstatic variable by tracing instructions not only within the defining routine but also within a routine that is called from the defining routine (and any other such nested calls.)
/OVER	Monitors a nonstatic variable by tracing instructions only within the defining routine, not within a routine that is called by the defining routine.
/[NO]SOURCE	Controls whether the source line for the current location is displayed at the watchpoint.

Monitoring Program Execution

The OpenVMS Debugger allows you to monitor the sequence in which your program instructions are executed, without interrupting program execution.

Tracepoints

Use tracepoints to monitor the execution of a program.

- A tracepoint is similar to a breakpoint since it suspends program execution and displays the address at the point of suspension.
- After a tracepoint is reached, program execution resumes immediately.
- The debugger prompt is not displayed when a tracepoint is reached.

The following is a summary of the commands to control tracepoints:

SET TRACE	Defines where execution is momentarily suspended
SHOW TRACE	Displays all currently set tracepoints
CANCEL TRACE	Removes currently set tracepoints

Setting Tracepoints

Use the SET TRACE command to establish tracepoints in your program. The syntax of the command is:

```
SET TRACE[/qualifier] address-expression [, ...]
  [WHEN (conditional-expression)]
  [DO (command[;p...])]
```

The ***address-expression*** parameter specifies the location(s) at which a tracepoint is to be set. With high-level languages such as FORTRAN, this is typically a line number, a routine name, or a label.

The ***conditional-expression*** parameter is a language expression that is evaluated when the tracepoint is about to be activated. If the expression does not evaluate to .TRUE., the tracepoint is not activated.

The ***command*** parameter represents any debugger command(s) that you want the debugger to execute as part of the DO command sequence when the trace action is taken.

Like the SET BREAK and SET WATCH commands, the SET TRACE accepts optional WHEN and DO clauses. After entering the SET TRACE command, use the GO command to start or resume program execution. The program executes as usual and the debugger displays tracepoint messages when the tracepoints are encountered.

The following are examples of the SET TRACE command:

`DBG> SET TRACE SUB1`	Sets a tracepoint at the entry point to subroutine SUB1.
`DBG> SET TRACE/LINE`	The trace starts at each new line.
`DBG> SET TRACE/CALL`	Traces every CALL instruction.

The example below uses the following source code to demonstrate the SET TRACE command:

```
1:     PROGRAM  MAIN
2:     J = 0
3:     I = J
4:     CALL  SUB1  (I)
5:     J = 2
6:     END
```

```
DBG>  SET  TRACE/LINE
DBG>  GO
trace  on  lines  at  MAIN\%LINE  2
    2:        J = 0
trace  on  lines  at  MAIN\%LINE  3
    3:        I = J
trace  on  lines  at  MAIN\%LINE  4
    4:        CALL  SUB1  (I)
trace  on  lines  at  routine  SUB1
trace  on  lines  at  SUB1+6
trace  on  lines  at  MAIN\%LINE  5
    5:        J = 2
trace  on  lines  at  MAIN\%LINE  6
    6:        END
%DEBUG-I-EXITSTATUS,  is  '%SYSTEM-S-NORMAL,  normal...
DBG>
```

Example 18-4 - Using the SET TRACE Command

Displaying Tracepoints

Use the SHOW TRACE command to display currently established tracepoints.

```
DBG> SHOW TRACE
tracepoint at routine SUB1
tracepoint on lines
DBG>
```

Example 18-5 - Using the SHOW TRACE Command

Cancelling Tracepoints

Use the CANCEL TRACE command to cancel a currently established tracepoint. For example, enter the following command to cancel the tracepoint at location SUB1:

```
DBG> CANCEL TRACE SUB1
```

SET TRACE Command Qualifiers

The following table describes the SET TRACE command qualifiers that affect tracepoint action.

/AFTER:*n*	Takes trace action after the ***n***th time the designated tracepoint is encountered.
/BRANCH	Traces every branch instruction during program execution.
/CALL	Traces every call and return instruction during program execution, including the return instruction.
/INTO	Traces the specified points within the called routines as well as within the routine in which execution is currently suspended.
/LINE	Traces the beginning of each source line encountered during program execution.
/OVER	Traces the specified points only within the routine in which execution is currently suspended.
/RETURN	Sets a tracepoint for the return instruction from an indicated routine. This qualifier can be applied to any routine.

Displaying Source Code

The debugger allows you to display source lines from any region in your program, in languages that support source code display. A module must be set before you can display its source code.

The debugger can display source lines only if you have specified the /DEBUG qualifier on both the compile command and the LINK command. The display of source lines is independent of program execution.

If a program has been optimized by the compiler, the code that is executing as you debug may not match the source code.

Methods of Displaying Source Code

Three debugger commands provide source code display capability. These commands are:

TYPE	To display source code by line number
EXAMINE/SOURCE	To display source code based on the program location
SEARCH	To display source code based on a search string.

TYPE Command

The TYPE command displays source code at a specific line or range of lines. The following are examples of the TYPE command.

`DBG> TYPE 15`	Displays a single line (line 15)
`DBG> TYPE 1:9`	Displays a range of lines (lines 1 thru 9)

`DBG> SET MODULE SUBPROG1`
`DBG> TYPE SUBPROG1\1:9`
	Display a range of lines in another module

EXAMINE/SOURCE Command

The EXAMINE/SOURCE command displays source code at a specific instruction or program location. The following are examples of the EXAMINE/SOURCE command.

`DBG> EXAMINE/SOURCE .PC`	Displays the source line to be executed next (PC refers to the program counter)

`DBG> SET MODULE SUBPROG2`
`DBG> EXAMINE/SOURCE SUBPROG2`
	Displays the first line of a subroutine

`DBG> EXAMINE/SOURCE .1\PC`	When in a subroutine, displays the source line to be executed next when control returns to the calling program (one level up the call stack)

SEARCH Command

The SEARCH command displays source code at a specific line or range of lines based on the results of a supplied source code search. The following are examples of the SEARCH command.

`DBG> SEARCH/ALL 1: "CALL"`	Displays all occurrences of "CALL", beginning at line 1 of the program.

`DBG> SET MODULE SUBPROG1`
`DBG> SEARCH SUBPROG1 "CALL"`
	Displays the first occurrence of "CALL" in a subroutine.

Fortran Code Listing - TEMP.LIS

```
TEMP                                                          20-APR-2002 22:
                                                              20-APR-2002 21:

          PROGRAM Temp
C
C         /*****************************************/
C         /* This program displays how cold it is */
C         /* outside, based on the temperature    */
C         /* entered by the user                  */
C         /*****************************************/
C
          INTEGER*4  Temperature ! temperature entered
          INTEGER*4  Centigrade  ! temperature in centigrade

          PRINT *, ‘ Enter the current temperature (in fahrenheit): ‘
          READ (*,2) Temperature
2         FORMAT (I4)

          Centigrade = INT ((5 * (Temperature - 32)) / 9)
          IF (Centigrade) 10, 20, 30

10        PRINT *, ‘ The temperature is below freezing...it is ‘
          PRINT 15, Centigrade, Temperature
15        FORMAT (I4, ‘ Centigrade, and ‘, I4, ‘ Fahrenheit’)
          GO TO 99

20        PRINT *, ‘ The temperature is freezing...it is ‘
          PRINT 15, Centigrade, Temperature
          GO TO 99

30        IF (CENTIGRADE .GE. 100) GO TO 40
          PRINT *, ‘ The temperature is above freezing...it is ‘
          PRINT 15, Centigrade, Temperature
          GO TO 99

40        PRINT *, ‘ The temperature is BOILING...it is ‘
          PRINT 15, Centigrade, Temperature

99        CONTINUE
          END     ! end of PROGRAM TEMP.FOR
```

Example 18-2 - Program Listing for TEMP.FOR named TEMP.LIS

SuRPAS Fortran Coding Standards for Conditionals and Loops

Overview

The SuRPAS Fortran coding standards that are addressed in this chapter include:

- Lining up IF-ENDIF, DO-ENDDO
- IF tests
- DO loops

Lining Up IF-ENDIF, DO-ENDDO

IF-ENDIF and DO-ENDDO pairs should be aligned in the same column. Do NOT indent the closing statement (ENDIF or ENDDO) to align it with the associated code block. This makes maintenance much harder, especially if the block is nested inside another. If the block spans more than half a page, a comment after the END IF is recommended, as in the following example:

```
     ENDIF     ! IF (MASACT .EQ. ‘ ‘)
```

Don’t hesitate to clean up code where the ENDIF’s and ENDDO’s are misaligned. Use your judgement, of course - if there are dozens of lines to fix and you’re under time pressure, it is acceptable to postpone this revision until a more convenient time.

IF-Tests

If your code must test a number of mutually-exclusive conditions, it is not acceptable to write separate tests because every case will be checked even after code gets a “hit”. Always code such tests as IF-ELSE IF-ENDIF blocks. For example, write:

```
     IF (Stat .EQ. ‘A’) THEN
          I = 1
     ELSEIF (Stat .EQ. ‘B’) THEN     ! only checked if
                                     ! STAT not ‘A’
          I = 2
     ELSEIF ...                      ! only checked if
          (etc.)                     ! STAT not ‘A’ or B’
     ENDIF
```

The following example of separate testing is NOT acceptable:

```
     IF (Stat .EQ. ‘A’) I = 1
     IF (Stat .EQ. ‘B’) I = 2
```

When ordering the tests, place the most-likely condition first so the subsequent checks will be made less frequently. If the code takes no action in a particular case, use a stand-alone CONTINUE statement as a “placeholder.” ***Be sure to comment that fact clearly!***

DO Loops

In general, DO loops should be written using the DO-ENDDO style. Loops that reference a numbered CONTINUE statement as their termination can be used if the statement is the target of a GOTO somewhere in the loop body. However, it may be preferable to always use the DO - ENDDO form and control flow by one of the following methods:

- Placing a target CONTINUE statement right before the ENDDO
- Using the Fortran CYCLE verb rather than a GOTO in the loop body to indicate that the intervening code is to be skipped

DO Loops (continued)
The EXIT verb may be used to break out of a loop rather than writing a GOTO whose target is a CONTINUE immediately after the ENDDO. The name EXIT is somewhat misleading, so its use should be clearly commented to indicate that the loop is being exited, not the entire module. The SA should determine whether the use of the EXIT is acceptable, or if a GOTO is easier to read.

Remember that Fortran checks the loop counter *before* the loop runs; therefore, it is not necessary to write an explicit IF test to branch around a loop if the counter is 0 or negative. However, clear commenting is still recommended if your code relies on the possibility that a loop may not execute.

It may also be possible to avoid a GOTO and/or an EXIT by appropriate construction of a DO-WHILE loop. A typical task in SuRPAS is to read a file using FALIOS until after the routine returns a nonzero status. To cast this in the form of a DO-WHILE loop:

- Make an initial read.
- Begin the DO - WHILE (IOS .EQ. 0...) processing.
- At the bottom of the DO - WHILE, make the NEXT read.

```
C    Initial read of ACRLLOAD.DAT
     CALL FALIOS (DAT_LUN, LUNTBL, ‘NEX’, ‘RRL’, ‘NLK’,
+         ‘ ‘, ‘ ‘, IOS, M6300C.IOB)
     IF (IOS .NE. 0) THEN
        CALL MFSERR (IOS)
     ENDIF

C    Loop for remaining record
     DO WHILE (IOS .EQ. 0)

C    (loop body to process results of FALIOS call ...)

C    Read next record at the bottom of the loop

     CALL FALIOS (DAT_LUN, LUNTBL, ‘NEX’, ‘RRL’, ‘NLK’,
+         ‘ ‘, ‘ ‘, IOS, M6300C.IOB)
     ENDDO        ! loop for ACRLLOAD.DAT
```

The condition for a DO-WHILE loop is tested at the start of the loop, and Fortran does not provide an equivalent DO-UNTIL construct whose condition is tested at the end of the loop. The above example uses two calls to FALIOS, where the first one loads the buffer and sets the status. If you have a true UNTIL loop which must be executed at least once regardless of starting conditions, you may have no choice but to simulate an UNTIL by use of GOTO’s.

Commands

Transfer Structures

GOTO *label*
Transfers control to a label at another location in the program.

CONTINUE
Receives execution control from a branch or loop. A NoOp (no operation).

GOTO (*label$_1$, label$_2$,...label$_n$*) *integer_expression*
Transfers control to a label, based on the value of the integer expression.
The GOTO statement is not supported when an IF structure can be used in its place.

Conditional Structures - preferred to GOTO statements

IF (*logical_expression*) *executable_statement*
Executes a supplied statement, if the logical expression is true.

IF...THEN...ENDIF
IF statement executes a block of code, if the logical expression is true.

IF...THEN...ELSE
Executes one block of code if logical expression is true, or another block if the logical expression is false.

Loop Structures

DO *terminating_label iteration_expression*
Iterating loop executes block of code a number of times based on index.

***Variable* (*index*), *index = initial_value, limit_value*, [*increment*]**
Implied DO loop.

DO..ENDDO
Iterating loop terminated by an EXIT or CYCLE statement

Conditional Loops

DO WHILE (*logical_expression*) ... ENDDO
Conditional DO loop.