Chapter 6

Control Statements

This chapter contains the following subsections:

- "CALL"
- "CONTINUE"
- "DO"
- "DO WHILE"
- "ELSE"
- "ELSE IF"
- "END"
- "END DO"
- "END IF"
- "GO TO (Unconditional)"
- "GO TO (Computed)"
- "GO TO (Symbolic Name)"
- "IF (Arithmetic)"
- "IF (Branch Logical)"
- "IF (Test Conditional)"
- "PAUSE"
- "RETURN"
- "STOP"

Control statements affect the normal sequence of execution in a program unit. They are described in alphabetic order in this chapter and summarized below.

CALL	References a subroutine program in a calling program unit.
CONTINUE	Has no operational function; usually serves as the terminal statement of a**DO** loop.
DO	Specifies a controlled loop, called a **DO** loop, and establishes the control variable, indexing parameters, and range of the loop.
DO WHILE	Specifies a **DO** loop based on a test for true of a logical expression.
ELSE	Used in conjunction with the block **IF** or **ELSE IF** statements.
ELSE IF	Used optionally with the block **IF** statement.
END	Indicates the end of a program unit.

END DO Defines the end of an indexed **DO** loop or a **DO WHILE** loop.

END IF Has no operational function; serves as a point of reference like a **CONTINUE** statement in a **DO** loop.

GO TO (Unconditional)
Transfers program control to the statement identified by the statement label.

GO TO (Computed)
Transfers control to one of several statements specified, depending on the value of an integer expression.

GO TO (Symbolic name)
Used in conjunction with an **ASSIGN** statement to transfer control to the statement whose label was last assigned to a variable by an assign statement.

IF (Arithmetic) Allows conditional branching.

IF (Branch logical)
Allows conditional statement execution.

IF (Test Conditional)
Allows conditional execution of blocks of code. The block **IF** can contain **ELSE IF** statements for further conditional execution control. The block **IF** ends with the **END IF**.

PAUSE Suspends an executing program.

RETURN Returns control to the referencing program unit. It can appear only in a function or subroutine program.

STOP Terminates an executing program.

CALL

The **CALL** statement references a subroutine subprogram in a calling program unit.

Syntax

`CALL` *sub*`[( [`*a*`[,`*a*`]...] )]`

where

sub is the symbolic name of the subroutine.

a is an actual argument, an expression, array name, array elements, record elements, record arrays, record array elements, Hollerith constants, or an alternate return specifier of the form **s*, where *s* is a statement label, or **&***s*, where *s* is a statement label.

Method of Operation

A **CALL** statement evaluates the actual arguments, association of the actual arguments with the corresponding dummy arguments, and execution of the statements in the subroutine. Return of control from the referenced subroutine completes the execution of the **CALL** statement.

Rules for Use

- The actual arguments *a* form an argument list and must agree in order, number, and type with the corresponding dummy arguments in the referenced subroutine.
- A subroutine that has been defined without an argument can be referenced by a **CALL** statement of the following forms:

  `CALL` *sub*
  `CALL` *sub*`()`
- If a dummy procedure name is specified as a dummy argument in the referenced subroutine, then the actual argument must be an external procedure name, a dummy procedure name, or one of the allowed specific intrinsic names. An intrinsic name or an external procedure name used as an actual argument must appear in an **INTRINSIC** or **EXTERNAL** statement, respectively.
- If an asterisk is specified as a dummy argument, an alternate return specifier must be supplied in the corresponding position in the argument list of the **CALL** statement.
- If a Hollerith constant is used as an actual argument in a **CALL** statement, the corresponding dummy argument must not be a dummy array and must be of arithmetic or logical data type. This rule is an exception to the first rule above.
- A subroutine can call itself directly or indirectly (recursion).

Note: Recursion is an extension to Fortran 77. Fortran 77 does not allow a subroutine to reference itself.

Example

In the following example, the main routine calls **PAGEREAD**, passing the parameters **LWORDCOUNT** , **PAGE**, and **NSWITCH**. After execution of **PAGEREAD**, control returns to the main program, which stops.

```
   program MakeIndex
   character*50 page
   dimension page (100)
   nswitch = 0
111       lwordcount = inwords1*2
*
   call pageread (lwordcount,page,nswitch)
   stop
*
   subroutine pageread (lwordcount,page,nswitch)
   character*50 page
   dimension page (100)
```

```
   icount = 100
        .
        .
        .
   end
*
```

CONTINUE

The **CONTINUE** statement has no operational function. It usually serves as the terminal statement of a **DO** loop.

Syntax

```
CONTINUE
```

Method of Operation

When a **CONTINUE** statement that closes a **DO** loop is reached, control transfer depends on the control variable in the **DO** loop. In this case, control will either go back to the start of the **DO** loop, or flow through to the statement following the **CONTINUE** statement. (See Item 3, "Loop Control Processing," in "DO" for full information about control of **DO** loops.)

Example

In the following example, the **DO** loop is executed 100 times, and then the program branches to statement 50 (not shown).

```
    iwordcount = 100
    do 25, i= 1,lwordcount
    read (2, 20, end=45) word
 20 format (A50)
 25 Continue
 *
    goto 50
```

DO

The **DO** statement specifies a controlled loop, called a **DO** loop, and establishes the control variable, indexing parameters, and range of the loop.

Syntax

DO [*s*] [,]*i* = *e1*, *e2* [, *e3*]

where

s is a statement label of the last executable statement in the range of the**DO** loop. This statement is called the terminal statement of the **DO** loop. *s* can be omitted. If *s* is omitted, the loop must be terminated with an **END DO** statement. On completion of the loop, execution resumes with the first statement following the **END DO** statement.

i — is a name of an integer, real, or double–precision variable, called the **DO** variable.

e1 — is an integer, real, or double–precision expression that represents the initial value given to the **DO** variable.

e2 — is an integer, real, or double–precision expression that represents the limit value for the **DO** variable.

e3 — is an integer, real, or double–precision expression that represents the increment value for the **DO** variable.

Method of Operation

The range of a **DO** loop consists of all executable statements following the statement, up to and including the terminal statement of the **DO** loop. In a **DO** loop, the executable statements that appear in the **DO** loop range are executed a number of times as determined by the control parameters specified in the **DO** statement.

The execution of a **DO** loop involves the following steps:

1. **Activating the DO loop**. The **DO** loop is activated when the **DO** statement is executed. The initial parameter *m1*, the terminal parameter *m2*, and the incremental parameter *m3* are established by evaluating the expressions *e1*, *e2*, and *e3*, respectively. The expressions are converted to the type of the **DO** variable when the data types are not the same. The **DO** variable becomes defined with the value of the initial parameter *m1*. The increment *m3* cannot have a value of zero and defaults to the value 1 if *e3* is omitted.

2. **Computing the iteration count.** The iteration count is established from the following expression:

   ```
   MAX( INT( (m2 - m1 + m3)/m3), 0)
   ```

   The iteration count is zero in the following cases:

   ```
   m1 > m2     and  m3 > 0
   m1 << m2  and  m3 = 0
   ```

   If the initial value (*m1*) of the **DO** exceeds the limit value (*m2*), as in

   ```
   DO 10 I = 2,1
   ```

   the **DO** loop will not be executed unless the **-onetrip** compiler option is in effect. This option causes the body of a loop thus initialized to be executed once.

   Ordinarily, on encountering a **D** –loop whose upper or lower limit has already been reached, the compiler–generated code skips the loop. This action conforms with Fortran standards.

   To make Fortran 77 compatible with Fortran 66, the compiler allows you to generate code that performs a loop at least once, regardless of whether the upper or lower limit has already been reached. This is accomplished by specifying the **-onetrip** option as described in Chapter 1 of the *Fortran 77 Programmer's Guide*. This option is included for older programs written under the assumption that all loops would be performed at least once.

3. **Loop control processing.** This step determines if further execution of the range of the **DO** loop is required. Loop processing begins by testing the iteration count. If the iteration count is positive, the first statement in the range of the **DO** loop is executed. Normal execution proceeds until the terminal statement is processed. This constitutes one iteration of the loop. Incrementing is then required, unless execution of the terminal statement results in a transfer of control.

   If the iteration count is zero, the **DO** loop becomes inactive. Execution continues with the first executable statement following the terminal statement of the **DO** loop. If several **DO** loops share the same terminal statement, incremental processing is continued for the immediately containing **DO** loop.

4. **Incremental processing.** The value of the **DO** variable is incremented by the value of the incremental parameter *m3*. The iteration count is then decreased by one, and execution continues with loop control processing as described above.

   A **DO** loop is either active or inactive. A **DO** loop is initially activated when its **DO** statement is executed. Once active, a **DO** loop becomes inactive when one of the following occurs:

   - The iteration count is zero.
   - A **RETURN** statement within the **DO** loop range is executed.
   - Control is transferred to a statement outside the range of the **DO** loop but in the same program unit as the **DO** loop.
   - A **STOP** statement is executed or the program is abnormally terminated.

   Reference to a subprogram from within the range of the **DO** loop does not make the **DO** loop inactive except when control is returned to a statement outside the range of the **DO** loop.

   When a **DO** loop becomes inactive, the **DO** variable of the **DO** loop retains its last defined value.

Rules for Use

- You can nest **DO** loops but do not overlap them.
- If a **DO** statement appears within an **IF** block, **ELSE IF** block, or **ELSE** block, the range of the **DO** loop must be contained within that block.
- If a block **IF** statement appears within the range of a **DO** loop, the corresponding **END IF** statement must appear within the range of the **DO** loop.
- The same statement can serve as the terminal statement in two or more nested **DO** loops.

Restrictions

- Do not use the following statements for the statement labeled s in the **DO** loop:

Unconditional **GO TO**	**END IF**
Assigned **GO TO**	**RETURN**
Arithmetic **IF**	**STOP**
Block **IF**	**END**
ELSE IF	Another **DO** statement
ELSE	

- If the statement labeled s is a logical **IF** statement, it can contain any executable statement in its statement body, except the following:

DO statement	**END IF**
Block**IF**	**END**
ELSE IF	Another logical **IF** statement
ELSE	

- Except by the incremental process covered above, the **DO** variable must not be redefined during the execution of the range of the **DO** loop.
- A program must not transfer control into the range of a **DO** loop from outside the **DO** loop. When this happens, the result is indeterminate.
- When the **DO** variable is a floating−point variable, especially if the loop increment value *e3* cannot be represented exactly in floating−point form, the number of times the loop executes could be off by one due to floating−point arithmetic error.

Example

```
   DO 10, i = 1, 10
          D
          D
          D
 10       CONTINUE
          E
```

In the above example, the statements noted with a **D** following the **DO** statement are executed sequentially ten times, then execution resumes at the statement **E** following **CONTINUE**.

DO WHILE

The **DO WHILE** statement specifies a controlled loop, called a **DO** loop, based on a test for true of a logical expression.

Syntax

`DO [s[,]] WHILE (e)`

where

s is a statement label of the last executable statement in the range of the **DO** loop. This statement is called the terminal statement of the **DO** loop.

e is a logical expression.

If s is omitted, the loop must be terminated with an **END DO** statement.

Method of Operation

The **DO WHILE** statement tests the specified expression before each iteration (including the first iteration) of the statements within the loop. When the logical expression e is found to be true, execution resumes at the statement specified by the label *s*. If *e* is omitted, execution resumes at the first statement following the **END DO** statement.

ELSE

Use the **ELSE** statement in conjunction with the block **IF** or **ELSE IF** statements.

Syntax

```
ELSE
```

Method of Operation

An **ELSE** block is the code that is executed when an **ELSE** statement is reached. An **ELSE** block begins after the **ELSE** statement and ends before the **END IF** statement at the same **IF** level as the **ELSE** statement. (For details about the term **IF** level, refer to "IF (Test Conditional)".) As well as containing simple, executable statements, an **ELSE** block can be empty (contain no statements) or can contain embedded block IF statements. Do not confuse the **ELSE** block with the **ELSE** statement.

An **ELSE** statement is executed when the logical expressions in the corresponding block **IF** and **ELSE IF** statements evaluate to false. An **ELSE** statement does not evaluate a logical expression; the **ELSE** block is always executed if the **ELSE** statement is reached. After the last statement in the **ELSE** block is executed (and provided it does not transfer control), control flows to the **END IF** statement that closes that whole **IF** level.

Rules for Use

- Do not specify **ELSE IF** or **ELSE** statements inside an **ELSE** block at the same **IF** level.
- The **IF** level of the **ELSE** statement must be greater than zero; that is, there must be a preceding corresponding block **IF** statement.

Restrictions

- Enter an **ELSE** block only by executing the **ELSE** statement. Do not transfer control into the **ELSE** block from the outside.
- If an **ELSE** statement has a statement label, the label cannot be referenced by any statement.

Example

The following example shows an **ELSE** block.

```
IF (R) THEN
    A = 0
ELSE IF (Q) THEN
    A = 1
ELSE
    A = -1
END IF
```

ELSE IF

The **ELSE IF** statement is used optionally with the **IF** block statement.

Syntax

`ELSE IF (`*e*`) THEN`

where *e* is a logical expression.

Method of Operation

Two terms need to be defined to explain the **ELSE IF** statement: **ELSE IF** block (defined below) and **IF** level (defined in "IF (Branch Logical)").

An **ELSE IF** block is the code that is executed when the logical expression of an **ELSE IF** statement is true. An **ELSE IF** block begins after the **ELSE IF** statement and ends before the next **ELSE IF**, **ELSE**, or **END IF** statement at the same **IF** level as the **ELSE IF** statement. As well as containing simple, executable statements, an **ELSE IF** block can be empty (contain no statements) or can contain embedded block **IF** statements. Do not confuse the **ELSE IF** block with the **ELSE IF** statement.

When an **ELSE IF** statement is reached, the logical expression *e* is evaluated. If *e* is true, execution continues with the first statement in the **ELSE IF** block. If the **ELSE IF** block is empty, control is passed to the next **END IF** statement that has the same **IF** level as the **ELSE IF** statement. If *e* is false, program control is transferred to the next **ELSE IF**, **ELSE**, or **END IF** statement that has the same **IF** level as the **ELSE IF** statement.

After the last statement of the **ELSE IF** block is executed (and provided it does not transfer control), control is automatically transferred to the next **END IF** statement at the same **IF** level as the **ELSE IF** statement.

Rule for Use

The **IF** level of the **ELSE IF** statement must be greater than zero (there must be a preceding corresponding block **IF** statement).

Restrictions

- Do not transfer control into an **ELSE I** block from outside the **ELSE IF** block.
- No statement can reference the statement label (if any) of an **ELSE IF** statement. The only way to reach an **ELSE IF** statement is through its **IF** block statement.

Example

The following example shows an **ELSE IF** block.

```
IF(R) THEN
    A = 0
ELSE IF (Q) THEN
    A = 1
END IF
```

END

The **END** statement designates the end of a program unit.

Syntax

```
END
```

Method of Operation

An **END** statement in a main program has the same effect as a **STOP** statement: it terminates an executing program.

An **END** statement in a function or subroutine subprogram has the effect of a **RETURN** statement: it returns control to the referencing program unit.

Rules for Use

- An **END** statement cannot be the last statement in every program unit.
- Do not continue an **END** statement.

END DO

The **END DO** statement defines the end of a indexed **DO** loop or a **DO WHILE** loop.

Syntax

```
END DO
```

END IF

The **END IF** statement has no operational function. It serves as a point of reference like a **CONTINUE** statement in a **DO** loop.

Syntax

```
END IF
```

Rules for Use

- Every block **IF** statement requires an **END IF** statement to close that **IF** level. (**IF** level is described in "IF (Test Conditional)").
- The **IF** level of an **END IF** statement must be greater than zero (there must be a preceding corresponding **IF** block statement).

Example

See the example given with the description of the **ELSE** statement in "ELSE".

GO TO (Unconditional)

The unconditional **GO TO** statement transfers program control to the statement identified by the statement label.

Syntax

GO TO *s*

where *s* is a statement label of an executable statement appearing in the same program unit as the unconditional **GO TO**.

Example

The following statement transfers program control to statement 358 and normal sequential execution continues from there.

```
GO TO 358
```

GO TO (Computed)

The computed **GO TO** statement transfers control to one of several statements specified, depending on the value of an integer expression.

Syntax

GO TO (*s*[,*s*]...)[,]*i*

where

s	is a statement number of an executable statement appearing in the same program unit as the computed **GO TO**.

i is an integer.

A noninteger expression can also be used for *i*. Non−integer expressions are converted to integers (fractional portions are discarded) before being used to index the list of statement labels.

Method of Operation

A computed **GO TO** statement evaluates the integer expression and then transfers program control to the specified statement

In the computed **GO TO** statement with the following form

```
GO TO (s1, s2, ... ,sn),i
```

if $i<1$ or $i>n$, the program control continues with the next statement following the computed **GO TO** statement; otherwise, program control is passed to the statement labeled *si*. Thus, if the value of the integer expression is 1, control of the program is transferred to the statement numbered *s1* in the list; if the value of the expression is 2, control is passed to the statement numbered *s2* in the list, and so on.

Rule for Use

The same statement label can appear more than once in the same computed **GO TO** statement.

Example

In the following example, the fifth list item is chosen because **KVAL** + 1 = 5. Program control is transferred to the statement labeled 350.

```
KVAL = 4
GO TO(100,200,300,300,350,9000)KVAL + 1
```

GO TO (Symbolic Name)

Use the symbolic **GO TO** statement in conjunction with an **ASSIGN** statement to transfer control to the statement whose label was last assigned to a variable by an **ASSIGN** statement.

Syntax

```
GO TO i [[,] (s [,s]...)]
```

where *i* is an integer variable name and *s* is a statement label of an executable statement appearing in the same program unit as the assigned **GO TO** statement.

Method of Operation

The variable *i* is defined with a statement label using the **ASSIGN** statement in the same program unit as the assigned **GO TO** statement. When an assigned **GO TO** is executed, control is passed to the statement identified by that statement label. Normal execution then proceeds from that point.

Rules for Use

- The same statement label can appear more than once in the same assigned **GO TO** statement.
- If a list in parentheses is present, the statement label assigned to *i* must be one of those in the list.

Example

```
GO TO KJUMP,(100,500,72530)
```

The value of **KJUMP** must be one of the statement label values: 100, 500, or 72530.

IF (Arithmetic)

The arithmetic **IF** statement allows conditional branching.

Syntax

IF (*e*) *s1*, *s2*, *s3*

where

e	is an arithmetic expression of type integer, real, or double–precision but not complex.
s1, *s2*, *s3*	are numbers of executable statements in the same program unit as the arithmetic **IF** statement.

Method of Operation

In the execution of an arithmetic **IF** statement, the value of the arithmetic expression *e* is evaluated. Control is then transferred to the statement numbered *s1*, *s2*, or *s3* if the value of the expression is less than zero, equal to zero, or greater than zero, respectively. Normal program execution proceeds from that point.

Rules for Use

You can use the same statement number more than once in the same arithmetic **IF** statement.

Example

Consider the following statement:

```
IF (A + B*(.5))500,1000,1500
```

If the expression **A + B*(.5)** is

- less than zero, control jumps to statement 500
- equal to zero, control jumps to statement 1000
- greater than zero, control jumps to statement 1500

IF (Branch Logical)

The branch logical **IF** statement allows conditional statement execution.

Syntax

```
IF (e) st
```

where

e is a logical expression.

st is any executable statement except **DO**, block **IF**, **ELSE IF**, **ELSE**, **END IF**, **END**, or another logical **IF** statement.

Method of Operation

A logical **IF** statement causes a Boolean evaluation of the logical expression. If the value of the logical expression is true, statement *st* is executed. If the value of the expression is false, execution continues with the next sequential statement following the logical **IF** statement.

Note that a function reference in the expression is allowed but might affect entities in the statement*st*.

Example

The following examples show branch logical **IF** statements.

```
IF(A .LE. B) A = 0.0
IF (M .LT. TOC) GOTO 1000
IF (J) CALL OUTSIDE(B,Z,F)
```

IF (Test Conditional)

The test conditional **IF** statement allows the conditional execution of blocks of code. The block **IF** can contain **ELSE** and **ELSE IF** statements for further conditional execution control. The block **IF** ends with the **END IF** statement.

Syntax

```
IF (e) THEN
```

where *e* is a logical expression.

Method of Operation

An **IF** block is the code that is executed when the logical expression of a block **IF** statement evaluates to true. An **IF** block begins after the block **IF** statement and ends before the **ELSE IF**, **ELSE**, or **END IF** statement that corresponds to the block **IF** statement. As well as containing simple, executable statements, an **IF** block can be empty (contain no statements) or can contain embedded block **IF** statements. Do not confuse the term **IF**–block with block**IF**.

Block **IF** statements and **ELSE IF** statements can be embedded, which can make figuring out which statements are in which conditional blocks confusing. The **IF** level of a statement determines which statements belong to which **IF−THEN−ELSE** block. Fortunately, the **IF** level of a statement can be found systematically. The **IF** level of a statement is

(*n1* − *n2*)

where (starting the count at the beginning of the program unit): *n1* is the number of block **IF** statements up to and including *s*, and *n2* is the number of **END IF** statements up to but *not* including *s.*

The **IF** level of every block **IF**, **ELSE IF**, **ELSE**, and **END IF** statement must be positive because those statements must be part of a block **IF** statement. The **IF** level of the **END** statement of the program unit must be zero because all block **IF** statements must be properly closed. The **IF** level of all other statements must either be zero (if they are outside all **IF** blocks) or positive (if they are inside an **I** −block).

When a block **IF** statement is reached, the logical expression *e* is evaluated. If *e* is true, execution continues with the first statement in the **IF** block. If the **IF** block is empty, control is passed to the next **END IF** statement that has the same **IF** level as the block **IF** statement. If *e* is false, program control is transferred to the next **ELSE IF**, **ELSE**, or **END IF** statement that has the same **IF** level as the block **IF** statement.

After the last statement of the **IF** block is executed (and provided it does not transfer control), control is automatically transferred to the next **END I***F* statement at the same **IF** level as the block **IF** statement.

Restriction

Control cannot be transferred into an **IF** block from outside the **IF** block.

Example

The following example shows a test conditional **IF** block.

```
IF(Q .LE. R) THEN
    PRINT ('Q IS LESS THAN OR EQUAL TO R')
ELSE
    PRINT ( 'Q IS GREATER THAN R')
END IF
```

PAUSE

The **PAUSE** statement suspends an executing program.

Syntax

PAUSE [*n*]

where *n* is a string of not more than five digits or a character constant.

Method of Operation

A **PAUSE** statement without an *n* specification suspends execution of a program and issues the following message:

```
PAUSE statement executed
```

```
To resume execution, type go. Any other inputwill terminate job.
```

A **PAUSE** statement with an *n* specification displays the specified character constant or digits and issues the pause message. For example, the following statement

```
PAUSE "Console Check"
```

results in the following message being displayed:

```
PAUSE Console Check statement executed
```

```
To resume execution, type go. Any other inputwill terminate job.
```

If execution is resumed, the execution proceeds as though a **CONTINUE** statement were in effect.

At the time of program suspension, the optional digit string or character constant becomes accessible to the system as program suspension status information.

RETURN

The **RETURN** statement returns control to the referencing program unit. It can appear only in a function or subroutine subprogram.

Syntax

In a function subprogram

```
RETURN
```

In a subroutine subprogram

RETURN [*e*]

where *e* is an integer expression specifying an alternate return.

A noninteger expression can be used for *e*. Noninteger expressions are converted to integer, and the fractional portions discarded, before control is returned to the alternate return argument.

Method of Operation

A **RETURN** statement terminates the reference of a function or subroutine and transfers control back to the currently referenced program unit. In a function subprogram, the value of the function then becomes available to the referencing unit. In a subroutine, return of control to the referencing program unit completes execution of the **CALL** statement.

A **RETURN** statement terminates the association between the dummy arguments of the external procedure and the current actual arguments.

In a subroutine subprogram, if e is not specified in a **RETURN** statement or if the value of *e* is less than or greater than the number of asterisks in the **SUBROUTINE** or **ENTRY** statement specifying the currently referenced name, then control returns to the **CALL** statement that initiated the subprogram. Otherwise, the value of e identifies the *e*th asterisk in the dummy argument list of the currently referenced name. Control returns to the statement identified by the alternate return specifier in the **CALL** statement that is associated with the *e*th asterisk in the dummy argument list.

The execution of a **RETURN** statement causes all entities in an external procedure to become undefined except for entities that are

- specified in a **SAVE** statement
- blank
- specified in a named common.
- initialized in a **DATA** statement that has neither been redefined nor become undefined

STOP

The **STOP** statement terminates an executing program.

Syntax

```
STOP [n]
```

where *n* is a string of not more than five digits or a character constant.

Method of Operation

The **STOP** statement terminates an executing program. If *n* is specified, the digit string or character constant becomes accessible to the system as program termination status information.