Chapter 10

Statement Functions and Subprograms

This chapter contains the following subsections:

- "Overview"
- "Statement Functions"
- "Parameter Passing"
- "Function and Subroutine Subprograms"
- "FUNCTION"
- "SUBROUTINE"
- "ENTRY"
- "INCLUDE"

Statement functions and subprograms are program units that receive control when referenced or called by a statement in a main program or another subprogram. A subprogram is either written by the user or supplied with the Fortran compiler. This chapter discusses user−written subprograms; compiler−supplied functions and subroutines are discussed in Appendix A, "Intrinsic Functions."

Overview

This chapter explains the syntax and rules for defining three types of program units:

- *Statement functions* consist of a single arithmetic statement defined within the main program unit or a subprogram.
- *Function subprograms* consist of one or more statements defined external to the main program unit. They are invoked when referenced as a primary in an expression contained in another program unit.
- *Subroutine subprograms* consist of one or more program statements defined as external to the main program unit. It is invoked when referenced in a **CALL** statement (See Chapter 6, "Control Statements") in another program unit.

This chapter also explains the syntax and rules for the **FUNCTION**, **SUBROUTINE**, **ENTRY**, and **INCLUDE** statements, that are used to specify function and subroutine subprograms.

Statement Functions

A statement function definition is similar in form to an arithmetic, logical, or character assignment statement. The name of a statement function is local to the program unit in which it is defined. A statement function definition must appear only after the specification statements and before the first executable statement of the program unit in which it appears.

Defining a Statement Function

A statement function statement has the form

fun ([*d* [,*d*]...]) = *e*

where

fun is a symbolic name of the function.

d is a dummy argument.

e is an expression.

Each dummy argument *d* is a variable name called a statement function dummy argument. The statement function dummy argument list indicates the order, number, and type of arguments for the statement function. All arguments need not have the same data type. A specific dummy argument may appear only once in the list. A variable name that serves as a dummy argument can also be the name of a local variable or common block in the same program unit.

Each primary of the expression *e* can include

- constants
- symbolic names of constants
- variable references
- array element references
- library function references
- references to other statement functions
- function subprogram references
- dummy subprogram references
- an expression composed of the above forms and enclosed in parentheses

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a function is a reference to the statement function dummy argument. A dummy argument that appears in a **FUNCTION** or **SUBROUTINE** statement may be referenced in the expression of a function statement with the subprogram.

A dummy argument that appears in an **ENTRY** statement may be referenced in the expression of the statement function only if the dummy argument name appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement preceding the statement function definition.

Referencing a Statement Function

A statement function is referenced by using its name with actual arguments, if any, enclosed in parentheses. The form of a statement function reference is

fun([*exp*[,*exp*]...])

where

fun is a statement function name.

exp is an expression.

Operational Conventions and Restrictions

Expressions must agree in order, number, and type with the corresponding dummy arguments. An expression can be any expression except a character expression involving concatenation in which the length attribute of one of the operands is specified with an asterisk.

Execution of a statement function reference results in

- Evaluation of actual arguments (*exp*) that are expressions.
- Association of actual arguments with their corresponding dummy arguments.
- Evaluation of the expression e in the statement function definition.
- Type conversion of the resulting value to the data type of the function, if necessary. This value is returned as the value of the statement function reference.
- A statement function can be referenced only in the program unit that contains its definition. A statement function can reference another statement function that has been defined before the referencing function but not one that is defined after the referencing function.
- A statement function name is local to the program unit and must not be used as the name of any other entity in the program unit except the name of a common block.
- The symbolic name used to identify a statement function cannot appear as a symbolic name in any specification statement except a type statement (to specify the type of the function) or as the name of a common block in the same program unit.
- A dummy argument of a statement function must not be redefined or become undefined through a function subprogram reference in the expression of a statement function.
- The symbolic name of a statement function cannot be an actual argument and must not appear in an **EXTERNAL** statement.
- A statement function in a function subprogram cannot contain a function reference to the name of an entry to the function subprogram.
- The length specification of a statement function dummy argument of type character must be an integer constant.

Parameter Passing

Parameter passing involves function and subroutine arguments. This section explains the difference between actual and dummy arguments. It also describes the Fortran extensions known as built–in functions.

Arguments

Dummy arguments are used in function subprograms, subroutine programs, and statement functions to indicate the types of actual arguments and whether each argument is a single value, an array of values, a subprogram, or a statement label. Dummy argument names must not appear in **EQUIVALENCE**, **DATA** , **PARAMETER**, **SAVE**, **INTRINSIC**, or **COMMON** statements, except as common block names. Dummy argument names must not be the same as the subprogram names in **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statements in the same program unit.

Actual arguments are the items that are specified in the call to the function. Actual arguments are bound to the corresponding dummy arguments when the subprogram call is reached. Actual arguments can change with each call to the subprogram. Of course, the types of the paired actual argument and the dummy argument must match. The types do not have to match if the actual argument is a subroutine name or an alternate return specifier.

When a function or a subroutine reference is executed, an association is established between the corresponding dummy and actual arguments. The first dummy argument becomes associated with the first actual argument, the second dummy argument becomes associated with the second actual argument, and so on.

An array can be passed to a function or subroutine as an actual argument if the corresponding dummy argument is also an array declared in a **DIMENSION** or type statement but not in a **COMMON** statement. The size of the array in the calling program unit must be smaller than or equal to the size of the corresponding dummy array in the subprogram. The array in the function or subroutine can also have adjustable dimensions.

Built−In Functions

Built−in functions provide communication with non−Fortran programs that require arguments passed in a specific format. (See Chapter 3 of the *Fortran 77 Programmer's Guide* for information about communicating with programs written in the C and Pascal languages.)

Use the built−in functions **%VAL**, **%REF**, and **%DESCR** along with arguments within an argument list. The built−in function **%LOC** is intended for global use.

%VAL

The bulit−in **%VAL** function passes the argument as a 32−bit value; the function extends arguments smaller than 32 bits to 32−bit signed values. The function has the following syntax:

`%VAL(`*a*`)`

where *a* is an argument within an argument list.

%REF

The built−in **%REF** function passes an argument by reference. It has the syntax

`%REF(`*a*`)`

where *a* is an argument within an argument list.

%DESCR

The built–in **%DESCR** function has no functionality, but is included for compatibility with VAX Fortran. It has the syntax

`%DESCR(`*a*`)`

where *a* is an argument within an argument list.

%LOC

The built–in **%LOC** function returns a 32–bit run–time address of its argument. It has the syntax

`%LOC(`*a*`)`

where *a* is an argument whose address is to be returned.

Function and Subroutine Subprograms

A function subprogram consists of a **FUNCTION** statement followed by a program body that terminates with an **END** statement. It has the following characteristics:

- defined external to the main program unit
- referenced as a primary in an expression contained in another program unit
- considered part of the calling program

A Fortran program can call a subroutine subprogram written in any language supported by the RISCompiler System. (See Chapter 3 of the *Fortran 77 Programmer's Guide* for information on writing Fortran programs that interact with programs in other languages.)

A subroutine subprogram consists of a **SUBROUTINE** statement, followed by a program body that terminates with an **END** statement (See Chapter 6, "Control Statements") and is defined external to the main program.

Referencing Functions and Subroutines

A function subprogram is referenced as a primary in an expression, while a subroutine subprogram is referenced with a **CALL** statement (See Chapter 6, "Control Statements") contained in another program. Reference to a function subprogram has the form

fun`([`*a*`[,`*a*`]...])`

where *fun* is a symbolic name of the function subprogram and *a* is an actual argument.

If *fun* is of type character, then its length must not have been specified with an asterisk (*) in the calling subprogram.

You can write subroutines that call themselves either directly or through a chain of other subprograms if the automatic storage of variables is in effect. The **-automatic** command line option (described in Chapter 11, "Compiler Options"), by default, causes the automatic storage of variables.

The actual arguments comprise an argument list and must agree in order, number, and type with the corresponding dummy arguments in the referenced function or subroutine. An actual argument in a function reference must be one of the following:

- an expression, except a character expression, involving concatenation of an operand whose length is specified by an asterisk
- an array name
- an intrinsic function name
- an external function or subroutine name
- a dummy function or subroutine name
- a Hollerith constant

An actual argument may be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

The use of a dummy name allows actual names to be passed through several levels of program units.

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.

The same rules apply to the actual arguments in a subroutine reference, except that in addition to the forms described above, the actual dummy argument of a subroutine may be an alternate return specifier. An alternate return specifier has the form *s, where *s* is the statement label of an executable statement appearing in the same program unit as the **CALL** statement.

For example,

```
SUBROUTINE MAXX(A,B,*,*,C)
```

The actual argument list passed in the **CALL** must include alternate return arguments in the corresponding positions of the form *s. The value specified for s must be the label of an executable statement in the program unit that issued the call.

An actual argument can also be omitted by specifying only the comma delimiters without an argument in between. In this case, the omitted argument is treated as if it were **%VAL** (0).

Note that the use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type. If an external function or subroutine or dummy name is used as an actual argument, the name must appear in an **EXTERNAL** statement. If an intrinsic name is used as an actual argument, the name must appear in an **INTRINSIC** statement and must be one of those listed in Appendix A, "Intrinsic Functions," as a specific name. It must not be one of the intrinsics for type conversion, for choosing the largest or smallest value, or for lexical relationship.

Executing Functions and Subroutines

Execution of an reference to a function subprogram and subroutine subprogram results in

- evaluation of expressions that constitute actual arguments

- association of actual arguments from the calling program unit with the corresponding dummy arguments in the subprogram
- execution of the statements comprising the subprogram based on the execution control sequence of the program unit
- return of program control to the calling program unit when either a **RETURN** statement is encountered or the execution control flows into the **END** statement

The name of a function subprogram must appear as a variable at least once in the subprogram and must be defined at least once during each subprogram execution. Once the variable is defined, it may be referenced elsewhere in the subprogram and become redefined. When program control is returned to the calling program, this value is returned as the value of the function reference. If this variable is a character variable with a length specified by an asterisk, it may not appear as an operand in a concatenation operation but can be defined in an assignment statement.

A subroutine does not return an explicit value to the point of invocation in the calling program unit. However, both the subroutine and the function can return values to the calling program unit by defining their dummy arguments during execution.

FUNCTION

The **FUNCTION** statement is the first statement of a function subprogram. It specifies the symbolic name of the function and its type.

Syntax

`[`*typ*`] FUNCTION` *fun* `[*`*len*`] ([`*d*`[,`*d*`]...])`

where

typ	optionally specifies the data type of the function name, which determines the value returned to the calling program. The following forms for *typ* are allowed:

INTEGER	DOUBLE PRECISION	LOGICAL
INTEGER*2	COMPLEX	LOGICAL*1
INTEGER*4	COMPLEX*8	LOGICAL*2
REAL	COMPLEX*16	LOGICAL*4
REAL*4	DOUBLE COMPLEX	CHARACTER [**len*]
REAL*8		

fun	is a symbolic name of the function subprogram in which the **FUNCTION** statement appears.
len	specifies the length of the data type; fun must be a nonzero, unsigned constant. Do not specify *len* when the function is type **CHARACTER** with an explicit length following the keyword **CHARACTER**.
w	is a dummy argument and can be a variable, array name, or dummy subprogram

name.

Rules for Use

- A **FUNCTION** statement must appear only as the first statement of a function subprogram.
- The type specification may be omitted from the **FUNCTION** statement, and the function name may be specified in a type statement in the same program unit. If neither of these options is used, the function is implicitly typed.
- The symbolic name of a function is a global name and must not be the same as any other global or local name, except a variable name, in the function subprogram.
- If the function type is specified in the **FUNCTION** statement, the function name must not appear in a type statement.
- In the type specification **CHARACTER**, *len* can have any of the forms allowed in a **CHARACTER** statement, except that an integer constant expression must not include the symbolic name of a constant. If the name of the function is type character, then each entry name in the function subprogram must be type character. If the length is declared as an asterisk, all such entries must have a length declared with an asterisk.
- A function specified as a subprogram may be referenced within any other subprogram or in the main program of the executable program.

Restrictions

- A function subprogram cannot contain a **BLOCK DATA**, **SUBROUTINE**, or **PROGRAM** statement.
- A function name cannot have its type explicitly specified more than once in a program unit.
- In a function subprogram, a dummy argument name cannot appear in an **EQUIVALENCE**, **PARAMETER**, **SAVE**, **INTRINSIC**, **DATA**, or **COMMON** statement, except as a common block name.
- A character dummy argument with a length specified as an asterisk must not appear as an operand for concatenation, except in a character assignment statement.
- The compiler system permits recursion if the automatic storage of variables is in effect. The **-automatic** command line option (See Chapter 11, "Compiler Options,"), by default, causes the automatic storage of variables.

SUBROUTINE

A **SUBROUTINE** statement must be the first statement of a subroutine subprogram.

Syntax

```
SUBROUTINE sub[( [d [,d]...] )]
```

where

sub is a symbolic name of the subroutine program unit.

d is a dummy argument and may be a variable name, array name, dummy subprogram name, or asterisk. The asterisk denotes an alternate return.

Rules for Use

- A **SUBROUTINE** statement must be the first statement of a subroutine subprogram.
- If there are no dummy arguments, use either of the following forms:

  ```
  SUBROUTINE sub
  SUBROUTINE sub()
  ```

- One or more dummy arguments can become defined or redefined to return results.
- The symbolic name of a subroutine is global and cannot be the same as any other global or local name in the program unit.
- A **CALL** statement within the body of a subroutine may reference the subroutine itself (recursion) if the automatic storage attribute is specified. See Chapter 4, "Specification Statements," for more information.

Restrictions

- A subroutine subprogram cannot contain a **BLOCK DATA**, **FUNCTION**, or **PROGRAM** statement.
- In a subroutine, a dummy argument name is local to the program unit and cannot appear in an **EQUIVALENCE**, **SAVE**, **INTRINSIC**, **DATA**, or **COMMON** statement, except as a common block name.
- A character dummy argument whose length is specified as an asterisk cannot appear as an operand for concatenation, except in a character assignment statement.

ENTRY

The **ENTRY** statement specifies a secondary entry point in a function or subroutine subprogram. It allows a subprogram reference to begin with a particular executable statement within the function or subroutine subprogram in which the **ENTRY** statement appears.

Syntax

```
ENTRY en[( [d[,d]...] )]
```

where

en is a symbolic name of the entry point.

d is a dummy argument.

If there are no dummy arguments, you can use the following forms:

```
ENTRY en
ENTRY en()
```

Method of Operation

Each **ENTRY** statement in a function or subroutine provides an additional name you can use to invoke that subprogram. When you invoke it with one of these names, it begins execution at the first executable statement following the entry statement that provided the name.

Within a function, each of its names (the one provided by the **FUNCTION** statement, plus the ones provided by the **ENTRY** statements) acts like a variable. By the time the function returns, you must have defined the function return value by assigning it to one of these variables.

If any of these variables is of type character, all must be of type character; otherwise, the variables need not all have the same data type. Such variables are in effect equivalenced, and therefore

- You need not assign the return value to the name you used to invoke the function; instead, you can assign it to any of the names of the same data type.
- If you assign the return value a name that does not have the same data type as the one you used to invoke the function, then the return value becomes undefined.

Rules for Use

- The **ENTRY** statement may appear anywhere within a function subprogram after the **FUNCTION** statement or within a subroutine after a **SUBROUTINE** statement.
- A subprogram can have one or more **ENTRY** statements.
- The entry name *en* in a function subprogram can appear in a type statement.
- In a function, a local variable with the same name as one of the entries can be referenced.
- A subprogram can call itself directly if the automatic storage of variables is in effect. The **-automatic** command line option (See Chapter 11, "Compiler Options"), by default, causes the automatic storage of variables.
- The order, number, type, and names of the dummy arguments in an **ENTRY** statement can be different from the dummy arguments in the **FUNCTION**, **SUBROUTINE**, or other **ENTRY** statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.

Restrictions

- An **ENTRY** statement must not appear between a block **IF** statement and its corresponding **END IF** statement or between a **DO** statement and the terminal statement of the **DO** loop.

- Within a subprogram, an entry name may not also serve as a dummy argument in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement or be in an **EXTERNAL** statement.
- In a function subprogram, an entry name may also be a variable name provided the variable name is not in any statement (except a type statement) preceding the **ENTRY** statement of that name. After the **ENTRY** statement, the name can be used as a variable name.
- In a function subprogram, if an entry name is of type character, each entry name and the name of the function subprogram must also be of type character and must have the same length declared. If any are of length (*), then *all* must be of length (*).
- In a subprogram, a name that appears as a dummy argument in an **ENTRY** statement is subject to the following restrictions:
  - It must not appear in an executable statement preceding that **ENTRY** statement unless it also appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement preceding the executable statement.
  - It must not appear in the expression of a statement function unless the name is also a dummy argument of the statement function. It can appear in a **FUNCTION** or **SUBROUTINE** statement or in an **ENTRY** statement preceding the statement function.

INCLUDE

The **INCLUDE** statement incorporates the contents of a designated file into the Fortran compilation directly following this statement.

Syntax

`INCLUDE "`*filename*`"`

where *filename* is a character string constant that specifies the file to be included.

Rules for Use

- An **INCLUDE** statement can appear anywhere within a program unit.
- On encountering an **INCLUDE** statement, the compiler stops reading statements from the current file and reads the statements in the included file. At the end of the included file, the compiler resumes reading the current file with the statement following the **INCLUDE** statement.

Search Path

On encountering an **INCLUDE** statement, the compiler first searches for a file called *filename*, then for a file named **/usr/include/***filename.*

Restrictions

- An included file or module cannot begin with a continuation line. Each Fortran statement must be completely contained within a single file.

- An **INCLUDE** statement cannon contain continuation lines. The first non comment line following the **INCLUDE** statement cannot be a continuation line.
- An **INCLUDE** statement cannot be labeled. It must not have a statement number in the statement number field.