Chapter 12

Procedures: Subroutines and Functions

This chapter describes the form and execution of function and subroutine subprograms. The general form of a subprogram definition is:

```
SUBR-or-FUNC procedure-name ( dummy-arg-name(s) )
specification-part
execution-part
END
```

where SUBR-or-FUNC is one of the keywords SUBROUTINE or FUNCTION, which identifies it as a subroutine or a function subprogram, respectively. Subroutines and functions are collectively called *procedures*. A procedure encapsulates computations defined in its *execution-part* that may be invoked during program execution by a *reference* to the procedure. Procedure references are described in Section 12.5.

A subprogram definition may include a dummy-arg-*name(s)* list that names one or more *dummy arguments* representing entities to be associated with the procedure at run time. A reference to a procedure associates an *actual argument* (an expression, procedure name, or statement label) with each dummy argument in a procedure definition. A dummy argument that is itself specified as a procedure or appears in a procedure reference is a *dummy procedure*.

The INTENT attribute of a dummy argument limits the way in which it may be used in a subprogram. A dummy argument having intent IN may not be changed by the subprogram. A dummy argument having intent OUT is initially undefined in the subprogram. A dummy argument with intent INOUT may be referenced or defined. A dummy data argument whose intent is not specified is subject to the limitations of the data entity that is associated with the actual argument. That is, a reference to the dummy data object may appear if the actual argument is defined, and may be defined or changed only if the actual argument is definable. A constant expression passed as an actual argument may not be defined by the subprogram.

Copyright © 1989–1994 Thinking Machines Corporation

A subroutine is invoked (referenced) by a CALL statement. A function is invoked by using its name in a function reference primary (R12-11, p. 170) in an expression. The invocation of a function computes a value that is then used in evaluating the expression. Both functions and subroutines can change the values of any variables accessible to the procedure: either variables passed as arguments, or variables in common blocks accessible to the procedure.

The *interface* of a procedure consists of the name of the procedure, the type and shape of each of its dummy arguments, and other characteristics of the procedure (see Section 12.6). A procedure interface is *explicit* if the interface is given by an *interface block* in the scope of a reference to that procedure, and is *implicit* otherwise. Procedure interface blocks are described in Section 12.6.1.

Procedures may be classified into three categories:

- Intrinsic functions
- External subroutines and functions
- Statement functions

A function that is provided as an inherent part of CM Fortran is an *intrinsic function*. (Chapter 17 describes all the intrinsic functions available in CM Fortran.) Some intrinsic functions are classified as *elemental* and they can be referenced with array arguments or scalar arguments (the function is applied to each element of its array argument). Elemental function reference is described in Section 12.5.2.2. An *external procedure* is a procedure that is defined either by an external subprogram or by a means other than CM Fortran. A function that is defined by a single statement is a *statement function*.

12.1 Subroutines: SUBROUTINE Statements

An *external subroutine* (R12-1) is a subprogram that collects together a body of executable statements that can be invoked using the CALL statement.

An *external-subroutine-subprogram* is: R12-1

SUBROUTINE subroutine-*name* [([*dummy-arg(s)*])]
[*specification-part*]
[*execution-part*]
END [SUBROUTINE [subroutine-*name*]]

A *dummy-arg* is one of: R12-2

- dummy-arg-*name*
- *

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- A given dummy-arg-*name* may appear only once in a *dummy-arg(s)* list.
- If a subroutine-*name* is present in the `END SUBROUTINE` statement, it must be identical to the subroutine-*name* specified in the `SUBROUTINE` statement.

The specifications of the dummy argument types and attributes collectively define the *interface* (Section 12.6) of the procedure.

The name of the subroutine is subroutine-*name*. The dummy arguments listed in the `SUBROUTINE` statement are associated through argument association (Section 12.6.3) with the actual arguments supplied in the `CALL` statement (Section 12.5.1) during execution of the subroutine. Execution of a subroutine is described in Section 12.5.1.

Following is a simple example of a subroutine `SUB` with three integer arguments. The subroutine passes the sum of its three arguments to the subroutine `FOO`.

```
SUBROUTINE SUB( A, B, C )
INTEGER A, B, C
CALL FOO( A+B+C )
END SUBROUTINE SUB
```

12.2 Function Subprograms: FUNCTION Statements

An *external function* (R12-3) is a subprogram that collects together a body of executable instructions that can be invoked by using the function name within an expression; the value returned by the function is used in place of the function reference in the expression.

An *external-function-subprogram* is: R12-3

[type-spec] `FUNCTION` function-*name* `(` *[* dummy-arg-*name(s)* *]* `)`
[specification-part]
[execution-part]
`END` *[* `FUNCTION` *[* function-*name* *]]*

- A given dummy-arg-*name* may appear only once in a dummy-arg-*name(s)* list.
- If a function-*name* is present in the `END FUNCTION` statement, it must be identical to the function-*name* specified in the `FUNCTION` statement.

The specifications of the function result attributes, the specification of the dummy argument attributes, and the information in the `FUNCTION` statement, collectively define the *interface* (Section 12.6) of the function.

Copyright © 1989–1994 Thinking Machines Corporation

The name of the function being defined is function-*name*. The result variable has the same name, and all occurrences of the function name in the *execution-part* statements of the subprogram are references to the result variable. The value of the result variable at the completion of the function is the value returned by the function. The result variable must be defined (set to a valid value) by the function.

The result type of a function is specified by a *type-spec* in the `FUNCTION` statement or by the appearance of the function name in a type declaration statement. It must not be specified both ways. If it is not specified either way, its result type is determined by the implicit typing rules in force within the function subprogram. If the function result is array valued, this fact must be indicated in type specification statements in the function body rather than with a *type-spec* in the `FUNCTION` statement.

The dummy arguments (if any) listed in a `FUNCTION` statement are associated through argument association (Section 12.6.3) with the actual arguments supplied in a function reference during execution of the function. Execution of a function is described in Section 12.5.2.

Following is a simple example of a function `FUN` with three integer arguments. The function returns the sum of its three arguments.

```
INTEGER FUNCTION FUN( A, B, C )
INTEGER A, B, C
FUN = A + B + C
END FUNCTION FUN
```

12.2.1 Array-Valued Function Subprograms

External function subprograms and some intrinsic functions in CM Fortran may return array-valued results. The result of an external function is defined to be array-valued by declaring its result variable to be an array in a type specification statement. An array-valued function result is a CM array.

Following is an implementation of a function `IOTA`, which returns an integer array of size `N`, with element values `1,2,...,N`.

```
FUNCTION IOTA(N)
INTEGER IOTA(N)
IOTA = [1:N]
END
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The example below illustrates the use of an array constructor and the intrinsic function DIAGONAL to construct an identity matrix of order N.

```
INTEGER FUNCTION MATID(N)
DIMENSION MATID(N,N)
MATID = DIAGONAL( [N[1]] )
END
```

NOTE: A procedure that references an array-valued function must explicitly declare the interface of the function in an interface block. A program fragment that references the procedures IOTA and MATID is given in Section 12.6.1.

12.3 Statement Functions

A *statement function statement* (R12-4) defines a function using a single statement. Within the program unit in which it is defined, the function can be invoked by using its name in a function reference in the same manner as an external function subprogram.

A *stmt-function-stmt* is: R12-4

function-*name* ([dummy-arg-*name(s)*]) = *scalar-expr*

- The *scalar-expr* defining a *stmt-function-stmt* may be composed only of primaries that are constants (literal and named), references to scalar variables and array elements (not to whole arrays), and references to functions and function dummy procedures, and operators. If a reference to another statement function appears in *scalar-expr*, its definition must have been provided earlier in the program unit.
- The function-*name* and each dummy-arg-*name* must be specified, explicitly or implicitly, to be scalar objects.
- A given dummy-arg-*name* may appear only once in the dummy-arg-*name(s)* list.
- Each scalar variable reference in the *scalar-expr* may be either a reference to a dummy argument of the statement function or a reference to a variable within the same program unit as the statement function statement.

Statement functions must be defined in the statement function part of a program unit, prior to any executable statements. The dummy argument names of a statement function have a scope of the statement function statement. A statement function may not be supplied as an actual argument to a procedure.

Copyright © 1989–1994 Thinking Machines Corporation

A statement function with *n* arguments and a *scalar-expr* of the form

function-name (ARG1, ARG2,..., ARG*n*) = *scalar-expr*

produces the same result value as an external function subroutine of the form

```
FUNCTION function-name( ARG1, ARG2,..., ARGn )
type-spec function-name
type-spec ARG1
type-spec ARG2
 ...
type-spec ARGn
function-name = scalar-expr
END FUNCTION function-name
```

where the type specification statements are those necessary to cause *function-name* and each dummy argument to be given explicitly the same type (and in the case of character types, the same length) as those names are given implicitly in the program unit containing the statement function.

The dummy arguments (if any) listed in a statement function statement are associated through argument association (Section 12.6.3) with the actual arguments supplied in a function reference (Section 12.5.2) during execution of the function.

Below is an example of two statement functions, QROOT1 and QROOT2, that compute the complex roots of the quadratic equation A*X**2 + B*X + C. The function QUADRATIC computes the value at point X in the complex plane of the quadratic equation defined by the values A, B, and C.

```
IMPLICIT COMPLEX (A-C,X)
COMPLEX QROOT1, QROOT2
QROOT1( A, B, C ) = ( -B + SQRT( B*B - 4*A*C ) ) / (2.*A)
QROOT2( A, B, C ) = ( -B - SQRT( B*B - 4*A*C ) ) / (2.*A)

QUADRATIC( A, B, C, X ) = A*X**2 + B*X + C
```

12.4 ENTRY Statement

One or more ENTRY statements (R12-5) may be used in a subprogram to define additional procedures for that subprogram that share declarations and executable statements with the subprogram.

An *entry-stmt* is: R12-5

ENTRY *entry-name [* (*[dummy-arg(s)]*) *]*

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

A *dummy-arg* is one of: R12-6

- *dummy-arg-name*
- *

- An ENTRY statement may appear only in an *external-subroutine-subprogram* or *external-function-subprogram*. An *entry-stmt* must not appear as part of an executable construct.
- An ENTRY statement may not appear in a function subprogram that returns an array-valued result.
- A given *dummy-arg-name* may appear only once in the *dummy-arg(s)* list.
- A dummy argument may be an alternate return specifier (an asterisk) only if the ENTRY statement is contained in an external subroutine subprogram.

If an ENTRY statement is contained in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*, and its dummy arguments are those specified with the ENTRY statement.

If an ENTRY statement is contained in a function subprogram, an additional function is defined by the subprogram. The name of the function and its result variable is *entry-name*, and its dummy arguments are those specified with the ENTRY statement. The characteristics (Section 12.6) of the function result are given by the type specification statements declaring *entry-name*.

In a function subprogram, if the characteristics of a function named with an ENTRY statement are the same as the characteristics of the function named with the FUNCTION statement, the function result variables are totally storage associated, provided that they have the same lengths if they are of type character. If the function results are of type character with different lengths, the result variables are partially associated.

An example of an external subroutine declaring several entry points is given below.

```
SUBROUTINE INT_STACK
INTEGER, PARAMETER :: SIZE = 5
INTEGER, SAVE, DIMENSION(SIZE) :: STACK
INTEGER, SAVE :: COUNT
COUNT = 0
RETURN
```


Copyright © 1989–1994 Thinking Machines Corporation

```
ENTRY PUSH( I )
IF ( COUNT < SIZE ) THEN
  COUNT = COUNT + 1
  STACK(COUNT) = I
  RETURN
ELSE
  STOP 'Stack Overflow'
END IF

ENTRY POP( I )
IF ( COUNT > 0 ) THEN
  I = STACK(COUNT)
  COUNT = COUNT - 1
  RETURN
ELSE
  STOP 'Stack Underflow'
END IF

ENTRY TOP( I )
IF ( COUNT > 0 ) THEN
  I = STACK(COUNT)
  RETURN
ELSE
  STOP 'Stack Underflow'
END IF

END
```

The subroutine implements an interface to a fixed-size stack of integer values. A call to INT_STACK initializes or reinitializes the stack to empty. A call to PUSH stores an integer value in the stack, unless the stack is full. A call to POP removes and returns the topmost element of a nonempty stack; a call to TOP returns the topmost element.

12.5 Procedure Reference

A subroutine is invoked by a reference to its name in a CALL statement; a function is invoked by the use of its name in an expression. When a procedure is invoked, an *instance* of that procedure is created and executed. Each instance has an execution sequence and a set of dummy arguments and nonsaved local objects whose values are independent of any previous instance of the procedure that has been executed. All other entities, including saved objects, variables in COMMON, and named constants, are associated with all instances of the procedure. For example, the value of a saved object may be defined in one execution

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

of a procedure and referenced in another. In CM Fortran, two instances of the same procedure can never coexist since recursive procedure invocations are not permitted.

12.5.1 Subroutine Reference: CALL Statement

A subroutine is invoked by execution of a `CALL` statement (R12-7). When a subroutine is invoked, all actual argument expressions are evaluated, then the actual arguments are associated with the dummy arguments, and then the subroutine is executed. When the action specified by the subroutine is completed, execution of the `CALL` statement is also completed. If a `CALL` statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the subroutine.

A *call-stmt* is: R12-7
`CALL` *subroutine-name [* `(` *actual-arg-list* `)` *]*

An *actual-arg-list* is: R12-8
[actual-arg [`,` *actual-arg]...]*

An *actual-arg* is one of: R12-9
- *expr*
- *procedure-name*
- *alternate-return-spec*

An *alternate-return-spec* is: R12-10
`*` *label*

- If an *actual-arg* is a *procedure-name* that names an intrinsic function, the name must be the *specific name* (Section 17.16) for the function.
- The *label* used in the *alternate-return-spec* must be the statement label of a branch target statement that appears in the same program unit as the *call-stmt*.

In a subroutine reference, the *actual argument list* (R12-8) identifies the correspondence between the actual arguments supplied (if any) and the dummy arguments of the procedure. Each actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; that is, the *i*th actual argument is associated with the *i*th dummy argument.

An *alternate return specifier* (R12-10) introduces a label into an argument list to allow the called subprogram to direct the execution of the caller upon return. This is done by specifying an integer value in a `RETURN` statement, as described in Section 12.9.

Copyright © 1989–1994 Thinking Machines Corporation

Examples of CALL statements:

```
CALL HOME()

CALL ENTER_NAME( NAME(1:I) )

CALL ALPHA( MAX( X, 0. ), SIN )
```

An example of a CALL statement with alternate return specifiers:

```
      CALL SUBR( X, Y, Z, *100, *200, *300 )
      ...
100   CONTINUE
      ...
      GO TO 999
200   CONTINUE
      ...
      GO TO 999
300   CONTINUE
      ...
999   CONTINUE
```

where labels 100, 200, and 300 are alternate return points. If subroutine SUBR terminates its execution through a RETURN statement specifying a value of 1, 2, or 3, then control is transferred to the statements 100, 200, or 300, respectively; otherwise, control is transferred to the statement immediately following the call to SUBR.

12.5.2 Function Reference

A function is invoked during the evaluation of an expression that includes a *function reference* (R12-11). When a function reference is executed, all actual argument expressions (R12-9) are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is used in place of the reference in evaluating the expression.

A *function-reference* is: R12-11
function-name (*actual-arg-list*)

An *actual-arg-list* is: R12-12
[actual-arg [, actual-arg]...]

A (function reference) *actual-arg* is one of: R12-13
- *expr*
- *procedure-name*

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- If an *actual-arg* is a *procedure-name* that names an intrinsic function, the name must be the *specific name* (Section 17.16) for the function.

The type (and length, for character expressions) of an expression in which a function reference appears does not affect and is not affected by the evaluation of the actual arguments of the function, except that the result returned by a generic intrinsic function may assume a type that depends on the type of its arguments.

In a function reference, the actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. In a function reference, each actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; that is, the *i*th actual argument is associated with the *i*th dummy argument.

If more than one function reference appears in a statement, the references may be executed in any order (subject to the restriction that a function result must be evaluated after the evaluation of its arguments) and their values must not depend on the order of execution.

Examples of function references:

```
X = 3. + FUN( Y-BASE, Z-BASE )

AREA = FINTEGRATE( SIN, 0., PI*2. )
```

The first example is an assignment statement that includes a reference to a user-defined function `FUN` with two actual arguments that are expressions. The second example is an assignment statement that includes a reference to function `FINTEGRATE`, the first argument of which is the specific name of the intrinsic function `SIN`.

If a function reference causes the value of an actual argument of the function to be modified, that argument or any entities associated with that argument must not appear elsewhere in the same statement. For example, the statements

```
A(I) = F(I)
Y = G(X) + X
```

are prohibited if the reference to `F` changes the value of `I` or if the reference to `G` changes the value of `X`.

Copyright © 1989–1994 Thinking Machines Corporation

Also, the execution of a function reference in a statement must not alter the value of any variable in a COMMON block if that alteration affects the value of any other function reference in the statement. For example, in the code fragment

```
COMMON /GLOBAL/ X
Z = FUN1(3.) + FUN2(X) + Y
```

the reference to function FUN1 must not alter the value of the variable X, since that would affect the result returned by function FUN2.

An exception to the rule above is that the execution of a function reference in the logical expression of an IF statement or a WHERE statement *is* permitted to alter the value of variables in the statement that is executed when the value of the expression is true. For example, in the statements

```
IF ( F(X) ) A = X
WHERE ( G(X) ) B = X
```

F or G may alter the value of X.

12.5.2.1 Intrinsic Function Reference

A function reference to one of the intrinsic functions defined in Chapter 17 is an *intrinsic function reference* (R12-14). Like a reference to an external function or a statement function, an intrinsic function reference may associate actual arguments with dummy arguments by their position in the actual argument list. An intrinsic function reference also may specify a keyword (R12-17) for some or all of its actual arguments to cause each keyword argument (R12-16) to be associated with the dummy argument whose name is the same as the keyword.

An (intrinsic) *function-reference* is: R12-14
intrinsic-function-name (*actual-keyword-arg-list*)

An *actual-keyword-arg-list* is: R12-15
[actual-keyword-arg [, actual-keyword-arg]...]

An *actual-keyword-arg* is: R12-16
[keyword =] actual-arg

A *keyword* is: R12-17
dummy-arg-name

- Each *keyword* must be the name of a dummy argument defined for the intrinsic function being referenced.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- A *keyword* may be omitted from an *actual-keyword-arg* only if the *keyword* is omitted from each preceding *actual-keyword-arg* in the argument list (that is, all positional arguments must precede all keyword arguments.)

Some intrinsic functions have optional dummy arguments that need not be supplied with an actual argument when the function is invoked. Chapter 17 lists all the intrinsic functions and their dummy argument names, with an indication of which arguments are optional. Exactly one actual argument must be associated with each non-optional dummy argument of the function, and each actual argument must be associated with a dummy argument.

Below are examples of references to the intrinsic function MAXVAL, whose last two arguments (DIM and MASK) are optional:

```
RMAX = MAXVAL( A )
RMAX = MAXVAL( A, 2 )
RMAX = MAXVAL( A, 2, A>0 )
RMAX = MAXVAL( ARRAY=A, DIM=2 )
RMAX = MAXVAL( ARRAY=A, DIM=2, A>0 )
RMAX = MAXVAL( DIM=2, ARRAY=A )
```

The first three references use positional arguments and the last three use keyword arguments. Note that the actual argument order does not match the dummy argument order in the last example.

12.5.2.2 Elemental Intrinsic Function Reference

An intrinsic function that is not a dummy procedure is said to be *elemental* if all its dummy arguments and its result are scalar objects. Arguments in a reference to an elemental function may be arrays provided that all array arguments have the same shape. Such a reference is called an *elemental reference*. The result of the reference has the type of the function named in the reference and the same shape as the array arguments; the value of each element in the result is obtained by evaluating the function using the scalar arguments and the corresponding elements of the array arguments. For example, if X and Y are arrays of shape [*m*,*n*],

```
MAX( X, 0., Y )
```

is an array expression of shape [*m*,*n*] whose elements have values

```
MAX( X(i,j), 0., Y(i,j) ), i = 1,2,...,m, j=1,2,...,n
```

CM Fortran does not support elemental references for user-defined functions.

Copyright © 1989–1994 Thinking Machines Corporation

12.6 Procedure Interfaces

The interface of a procedure determines the forms of reference through which it may be invoked. The interface for an external procedure or dummy procedure is specified by a `FUNCTION`, `SUBROUTINE`, or `ENTRY` statement and by specification statements for the dummy arguments and the result (for functions). These statements may appear in a procedure definition, in an interface block, or both (except that an `ENTRY` statement must not appear in an interface block). The names of all procedures accessible in a single program unit or interface block must be distinct, regardless of whether their interfaces differ.

The *interface* of a procedure consists of the name of the procedure, the name of each of its dummy arguments, and the characteristics of the procedure. The *characteristics* of a procedure are:

- The classification of the procedure as a function or a subroutine.
- The characteristics of each of its dummy arguments.
- For functions, the characteristics of the result value, which include its rank and result type and whether (for character functions) the result is assumed-length. If an array bound of an array-valued function result is not a constant expression, that fact is also a characteristic.

The *characteristics* of a dummy argument are:

- For a dummy object:
  - its type
  - its shape
  - its intent attribute, if any
  - for a dummy array object:
    - whether it is an assumed-size or assumed-shape array
    - its processor *home* (CM or front-end)
    - its *layout* (if it has a CM home), which includes the *orientation*, *processor ordering*, and *communications weight* associated with each of its dimensions (see Section A.2 for a definition of these terms)
  - for a dummy object of type character, its length, or the fact that its length is assumed
- For a dummy procedure, whether or not its interface is explicit, and, if so, the characteristics of the dummy procedure
- For an asterisk dummy argument, the fact that it represents a statement label

The next section describes the use of interface blocks, which allow the interface of an external procedure to be specified explicitly.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

12.6.1 Interface Blocks

An *interface block* (R12-18) specifies the interface of a particular external procedure and determines the forms of reference through which the procedure may be invoked.

An *interface-block* is one of: R12-18
- *function-interface-block*
- *subroutine-interface-block*

A *function-interface-block* is: R12-19

```
INTERFACE
[ type-spec ] FUNCTION function-name ( [ dummy-arg-name(s) ] )
[ implicit-part ]
[ declaration-part ]
END INTERFACE
```

A *subroutine-interface-block* is: R12-20

```
INTERFACE
SUBROUTINE subroutine-name [ ( [ dummy-arg-name(s) ] ) ]
[ implicit-part ]
[ declaration-part ]
END INTERFACE
```

♦ An *interface-block* must not contain an *entry-stmt*, a *format-stmt*, or a *data-stmt*.

An interface block specifies the interface of the procedure named in the `FUNCTION` or `SUBROUTINE` statement in the interface block. This allows compile-time checking of the type and shape of arguments.

An interface block may appear within the specification part of a main program or subprogram, or within the declarations of another interface block. Only one interface block may be provided for a particular procedure within a program unit or interface block. The name and characteristics of a procedure as specified by an interface block must be *identical* to the name and characteristics of the procedure as defined. The name and characteristics of a procedure are fixed, but the names of its dummy arguments may differ in different interface blocks.

An interface block may describe the interface of a procedure defined as a dummy argument of another procedure (possibly in a surrounding interface block). A dummy argument must not be so named more than once in the interface block, and such a dummy argument must not appear in any other specification statement in the interface block.

Copyright © 1989–1994 Thinking Machines Corporation

The interface block in the following example

```
INTERFACE
  SUBROUTINE CALC( MASK, N, CUBE, EPSILON )
  LOGICAL MASK(N)
  DOUBLE PRECISION CUBE(N,N,N)
  DOUBLE PRECISION EPSILON
END INTERFACE
```

specifies the interface of a subroutine `CALC` with two adjustable array arguments (`MASK` and `CUBE`) and two scalar arguments (`N` and `EPSILON`). References to `CALC` in the scope of the interface block will be checked during compilation to ensure that all arguments are conformable and of compatible types.

A program fragment that references the array-valued functions `IOTA` and `MATID` defined in Section 12.2.1 is:

```
...
INTERFACE
  FUNCTION IOTA(N)
  INTEGER IOTA(N)
END INTERFACE

INTERFACE
  FUNCTION MATID(N)
  INTEGER MATID(N,N)
END INTERFACE

...
PARAMETER ( M = 20 )
INTEGER ID(M,M), NUMS(M)
...
ID = MATID(M)
NUMS = IOTA(M)
...
```

An interface block is given below for a subroutine `APPLY` specifying dummy argument `A` to be a rank-two assumed-shape array and dummy argument `FN` to be a function of type real with one real argument.

```
INTERFACE
  SUBROUTINE APPLY( A, FN )
  REAL A(:,:)
  INTERFACE
    FUNCTION FN( B )
    REAL FN, B
  END INTERFACE
END INTERFACE
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

References to subroutine APPLY within a program unit containing the interface block will be checked for conformance to the interface.

12.6.1.1 Order of Statements in an Interface Block

The syntax rules given in Section 12.6.1 and other parts of this document specifying the order of statements within an interface block are illustrated in Figure 5. In the figure, vertical lines delineate kinds of statements that may be interspersed and horizontal lines delineate kinds of statements that must not be interspersed.

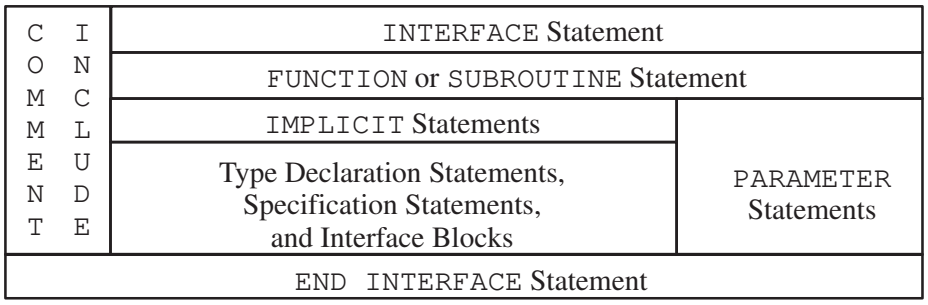

Figure 5. Required order of statements in an interface block.

- COMMON statements, EQUIVALENCE statements, DATA statements, and SAVE statements may appear in an interface block. DATA statements and SAVE statements appearing in an interface block are ignored.
- Interface blocks may be nested within interface blocks to a depth of two. (This is useful when using procedure arguments.)

A similar figure is given in Section 3.5.1 for main programs and subprograms, and in Section 9.2.1 for block data program units.

12.6.2 Implicit and Explicit Interfaces

If a procedure is accessible in a program unit or interface block, its interface is either *explicit* or *implicit* in that program unit or interface block.

- The interface of an intrinsic function is always explicit within a program unit or an interface block.

Copyright © 1989–1994 Thinking Machines Corporation

- Within a program unit, the interface of an external procedure or dummy procedure is explicit if an interface block for the procedure is supplied in that program unit, and implicit otherwise.
- Within an interface block, the interface of a dummy procedure is explicit if an interface block for the dummy procedure exists in the containing program unit or interface block.

An intrinsic function is not accessible in a program unit or interface block if an external function with the same name is declared using the `EXTERNAL` statement (Section 12.7).

A procedure *must* have an explicit interface if the procedure is a function that returns an array-valued result. (If no interface block is present defining the function result shape, it is assumed to be scalar.)

In a program unit that references an external function whose interface is implicit, only the type of the function result is specified, either implicitly by the implicit typing rules in effect for the program unit, or explicitly through the appearance of the function name in a type specification statement. The actual arguments passed must have the type and shape of the dummy arguments defined by the procedure.

A procedure *must* have an explicit interface if it has a dummy argument that is a pointer.

Though not strictly necessary, it is recommended that an explicit interface be used in calling any procedure that has one or more dummy CM array arguments. The explicit interface enables the compiler to flag mismatches in shape, type, and layout, and may increase program performance.

12.6.3 Argument Association

Argument association is the means by which actual arguments in a procedure reference are associated with the dummy arguments of the referenced procedure. Argument association provides one way entities known in one program unit may be accessed in another program unit. Argument association of an actual argument with a dummy argument may cause them to become sequence associated (Section 12.6.4).

All argument associations established by a reference to a procedure are terminated when the execution of that procedure completes. A dummy argument of that procedure may be associated with an entirely different actual argument in a subsequent execution of the procedure.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The name of a dummy argument may be different from the name, if any, of its associated actual argument. (An actual argument may be a nameless data entity, such as an expression that is not simply the name of a variable or a constant.) The dummy argument name is the name by which the associated actual argument is known in the referenced procedure.

12.6.3.1 Arguments Associated with Dummy Objects

If a dummy argument is a data object (not a dummy procedure or an asterisk), the associated actual argument must be an expression of the same type or an object of the same type. A character actual argument may have a length greater than that of the character dummy argument if the interface is implicit, in which case the effect is as if the leftmost substring of the appropriate length were used as the actual argument.

The shape of an actual argument must agree with the shape of the corresponding dummy argument, except in the following cases:

- The interface of the procedure is implicit, in which case the actual argument is sequence-associated (Section 12.6.4) with the dummy argument. (Note that sequence association only occurs between actual and dummy array arguments that are front-end arrays.)
- The dummy argument is an assumed-shape array, in which case the dummy argument assumes the shape of the actual argument.
- The reference is to an elemental intrinsic function, in which case the reference is treated as a collection of references to the function, each with scalar arguments from corresponding elements of the array arguments.

If a dummy argument has the `INTENT(OUT)` or `INTENT(INOUT)` attribute, the actual argument must be definable.

12.6.3.2 Pointers as Actual Arguments

When a pointer is passed to a procedure without an interface block, the pointer is *dereferenced*. In other words, the object the pointer points to is passed to the procedure, just as in normal assignment. Therefore, a pointer can be passed to a dummy argument that is not a pointer.

Copyright © 1989–1994 Thinking Machines Corporation

12.6.3.3 Pointers as Dummy Arguments

If the dummy argument is a pointer, the actual argument must be a pointer (not necessarily allocated) and the types, type parameters, and ranks must agree. The dummy argument pointer points at the target of the actual pointer at the invocation of the procedure. The pointer can be deallocated or allocated during the execution of the procedure. When execution of the procedure completes, the target of the actual argument pointer becomes that of the dummy argument.

The following are constraints on declaring dummy array arguments as pointers:

- If a dummy array argument is a pointer, the calling procedure must contain an interface block that declares the dummy argument as a pointer.

```
INTEGER, POINTER, DIMENSION(:,:,:) :: ACTUAL_D3
...
INTERFACE
   SUBROUTINE SUB(D3)
   INTEGER, POINTER, DIMENSION(:,:,:) :: D3
END INTERFACE
...
ALLOCATE (ACTUAL_D3 (N,M,N+M))

CALL SUB( ACTUAL_D3 )
...
```

- An actual argument that is a pointer may be passed to a non-pointer dummy argument. In this case, no interface block is required.

- A dummy array argument that is a pointer may not have the `INTENT` attribute.

12.6.3.4 Arguments Associated with Dummy Procedures

If a dummy argument is a dummy procedure, the actual argument associated with it must be the name of an external procedure, a dummy procedure, or a specific intrinsic function. The characteristics of the associated actual procedure must be the same as the characteristics of the dummy procedure, regardless of whether the interface of the dummy is implicit or explicit. The actual argument procedure may have dummy character arguments with assumed lengths. If the procedure is a function, it must be referenced as a function; if it is a subroutine, it must be referenced as a subroutine.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

```
INTEGER FUNC
CALL FUNC            ! can't call FUNC as subroutine
CALL XYZZY( A, B )   ! can't reference XYZZY as subroutine
X = XYZZY( 3 )       !   and also as function
I = FUNC( 42 )
```

Some specific intrinsic functions are not permitted to be passed as arguments. Section 17.16 lists the CM Fortran intrinsic functions that have specific names; those marked with an asterisk may not be passed as arguments.

12.6.3.5 Arguments Associated with Alternate Return Indicators

If a dummy argument of a subroutine is an asterisk, the associated actual argument must be an alternate return specifier (R12-10, p. 169). The label in the alternate return specifier must identify an executable construct in the program unit containing the procedure reference.

An alternate return specifier may not be passed as an actual argument in a function reference. Note that the syntax of function subprograms and statement function statements does not permit the declaration of asterisk dummy arguments.

12.6.3.6 Restrictions on Entities Associated with Dummy Arguments

While an entity is associated with a dummy argument, the following restrictions hold:

- No action may be taken that affects the value or availability of the entity or any part of it, except through the dummy argument.
- If any part of the entity is defined through the dummy argument, then at any time during the execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument.

These restrictions are illustrated by the following example:

```
COMMON /CONTRIVED/ LASTVAL, MAXVAL
CALL SET( LASTVAL, 23. )
...
END
```


Copyright © 1989–1994 Thinking Machines Corporation

```
SUBROUTINE SET( VAR, VAL )
REAL LASTVAL, MAXVAL
COMMON /CONTRIVED/ LASTVAL, MAXVAL
VAR = VAL
LASTVAL = VAL  ! invalid if LASTVAL passed as dummy to VAR
MAXVAL = MAX( LASTVAL, MAXVAL ) ! ditto for reference to MAXVAL
END
```

The assignment to the common variable LASTVAL in subroutine SET is invalid according to the first restriction since that assignment affects the value of the dummy argument VAR. The reference to common variable LASTVAL is invalid according to the second restriction because it is changed by the assignment to dummy argument VAR.

Note that if there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure, the overlapped portions are unchangeable during the execution of the procedure. For example, in

```
CALL SUB( A(1:5), A(3:9) )
```

the subarray A(3:5) cannot be changed through the first dummy argument because it is part of the argument associated with the second dummy argument, and cannot be changed through the second dummy argument because it is part of the argument associated with the first dummy argument. The subarray A(1:2) can be modified through the first dummy argument, and A(6:9) can be modified through the second dummy argument.

NOTE: Except for arguments of type character, partial association must not occur through argument association. Partial association of non-character dummy objects may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements (see Sections 8.1 and 12.4).

12.6.4 Sequence Association of Arguments

If the interface for a procedure is implicit, an actual argument can be sequence-associated with a dummy argument in a reference to that procedure. If an actual argument is sequence-associated with a dummy argument, the rank and shape of the actual argument *need not* agree with the rank and shape of the dummy argument, but the number of elements in the element sequence of the actual argument must not be less than the number of elements in the element sequence of the dummy argument. If the dummy argument is assumed-size, the number of elements in the dummy argument is exactly the number of elements in the element sequence.

In a reference to a procedure with an *implicit* interface, every actual argument is *sequence-associated* with its corresponding dummy argument. In a reference to a procedure with an

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

explicit interface, an actual argument is sequence-associated with its corresponding dummy argument only if the actual argument represents an element sequence. An actual argument represents an *element sequence* if it is:

- The name of a whole array that is a sequence array (see below) — the element sequence consists of the array elements in array element order.
- An array element designator (R7-14, p. 74) of a sequence array — the element sequence consists of that array element and each element that follows it in array element order.
- A substring (R7-9, p. 73) of an array element of a sequence array — the element sequence consists of the character storage units beginning with the first character storage unit in that array element substring and continuing to the end of the array.
- The character storage units are viewed as elements consisting of consecutive groups of character storage units, the first of which is the array element substring itself. Each substring has the length of the array element substring. This has the effect of superimposing a contiguous sequence of substrings of one length over a contiguous sequence of strings of another length; the result is that some elements of the resulting element sequence may consist of storage units from several different elements of the original array.

A *sequence array* is one of:

- an explicit-shape front-end array, whether compiled by CM Fortran or some other Fortran compiler
- an assumed-size front-end dummy array, whether compiled by CM Fortran or some other Fortran compiler

A *nonsequence array* is one of:

- an explicit-shape CM array
- an assumed-shape dummy array (always a CM array)
- an array section (including a section created by vector-valued subscripting).

Each element of an array-valued actual argument or of a sequence in a sequence association is associated with the element of the dummy array that has the same position in array element order.

The effects of the shape matching rules for procedure references (not including references to elemental intrinsic functions), and the explicit interface requirements are summarized in Figure 6, page 187.

Copyright © 1989–1994 Thinking Machines Corporation

12.7 EXTERNAL Statement

The EXTERNAL statement (R12-21) lists one or more names that represent external procedures or dummy procedures, and permits such names to be used as actual arguments.

An *external-stmt* is: R12-21
EXTERNAL *external-name(s)*

- Each *external-name* must be the name of an external procedure, a dummy argument, or a block data program unit.

The appearance of an intrinsic function name in an EXTERNAL statement causes that name to become the name of some external subprogram and the intrinsic function of the same name is *not available* in the program unit or interface block. Note that the redeclaration of the type of an intrinsic function (without specifying its name in an EXTERNAL statement) is not sufficient to make it an external procedure, even if the type is inconsistent with the type of the result of an intrinsic function of the same name.

Only one appearance of a given name is permitted in all the EXTERNAL statements appearing in a program unit or interface block. A name that appears in an EXTERNAL statement must not also appear as a specific procedure name in an interface block in the scoping unit.

An example of an EXTERNAL statement is given below.

```
SUBROUTINE SUB( FOCUS )
EXTERNAL FOCUS
EXTERNAL MATMUL
...
```

In the example, the dummy argument FOCUS is specified to be a dummy procedure (the code fragment does not indicate whether it is a function or a subroutine). References to the name MATMUL are regarded as references to a user-supplied external procedure MATMUL rather than to the intrinsic function of the same name.

12.8 INTRINSIC Statement

An INTRINSIC statement (R12-22) lists one or more names that represent intrinsic functions. It also permits a name that represents a specific intrinsic function to be used as an actual argument.

An *intrinsic-stmt* is: R12-22
INTRINSIC *intrinsic-procedure-name(s)*

Copyright © 1989–1994 Thinking Machines Corporation

The appearance of a name in an INTRINSIC statement confirms that the name is the name of an intrinsic function. The appearance of a generic intrinsic function name in an INTRINSIC statement does not cause that name to lose its generic property. (See Section 17.16 for a list of generic and specific function names.)

Only one appearance of a given name is permitted in all the INTRINSIC statements appearing in a program unit or interface block. Note that a name must not appear in both an EXTERNAL and an INTRINSIC statement in the same sequence of specification part statements.

An example of an INTRINSIC statement is given below.

```
SUBROUTINE SUB
INTRINSIC MATMUL
...
END
```

The name MATMUL is confirmed to be an INTRINSIC function. If the subprogram is compiled using a compiler other than CM Fortran, an error should be indicated if the compiler does not support the intrinsic function MATMUL.

12.9 RETURN Statement

Execution of a RETURN statement (R12-23) terminates the execution of a function or subroutine program.

A *return-stmt* is: R12-23
RETURN *[* scalar-integer-*expr]*

- A *return-stmt* may appear only in the *execution-part* of a function or subroutine subprogram.
- The scalar-integer-*expr* is allowed only on a RETURN statement appearing in a subroutine subprogram.

If the statement specifies a scalar integer expression that has a value *n* between 1 and the number of asterisks in the dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specified in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Execution of an END FUNCTION or END SUBROUTINE statement is equivalent to executing a RETURN statement with no expression.

Copyright © 1989–1994 Thinking Machines Corporation

An example of a RETURN statement:

```
SUBROUTINE DOIT( IT, *, *, * )
RETURN IT
END SUBROUTINE
...
CALL DOIT( 2, *100, *120, *180 )
...
```

The RETURN statement in the example above transfers control to one of the three statement labels passed to the subroutine when it is called, if the argument IT has the value 1, 2, or 3; otherwise, control is transferred to the statement following the CALL statement with which it is invoked.

12.10 External Procedures Defined by Other Means

Procedures may be defined by means other than CM Fortran. A reference to such a procedure is made as though it were defined by an external subprogram. The interface to a non-CM-Fortran procedure may be specified in an interface block. (See the *CM Fortran User's Guide* for information about the definition of procedures by means other than CM Fortran.)

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

			DUMMY ARGUMENT				
			Explicit-shape (CM)	**Explicit-shape** (FE)	**Assumed-shape** (CM only)	**Assumed-size** (FE only)	**Scalar** (FE only)
ACTUAL ARGUMENT	NONSEQUENCE	**CM array or section**	allowed[1]	not allowed	allowed	not allowed	not allowed
		CM array element	not allowed	not allowed	not allowed	not allowed	allowed
	SEQUENCE	**Front-end explicit-shape array**	not allowed	allowed[2]	not allowed	allowed[2]	not allowed
		Front-end assumed-size array	not allowed	allowed[2]	not allowed	allowed[2]	not allowed
		Front-end array element	not allowed	allowed	not allowed	allowed	allowed
		Scalar non-element	not allowed	not allowed	not allowed	not allowed	allowed

[1] The *shape* of the actual must match that of the dummy, regardless of whether the interface is implicit or explicit. The *layout* of the actual may differ from that of the dummy if the interface is explicit.

[2] If the interface is implicit, the *shape* of the actual may differ from the shape of the dummy (but the actual must be at least the size of dummy).

Figure 6. Rules for passing actual arguments to dummy arguments.

Copyright © 1989–1994 Thinking Machines Corporation