Chapter 6

Subprograms

To operate on scalars and front-end arrays, a CM Fortran program can define and invoke procedures — subroutines and functions — in the Fortran 77 fashion. To operate on CM arrays and array sections (that is, arrays referenced in the Fortran 90 manner), you need to be aware of some differences in argument passing. These differences concern

- the semantics of passing CM arrays as arguments
- the requirement that the homes of actual and dummy arguments must match
- the requirement that the shapes and layouts of actual and dummy CM array arguments must match
- the ability to declare dummy array arguments that assume the shape or layout of the actual argument passed
- the need for interface blocks — explicit subprogram interfaces in the calling program — in some argument-passing situations

CM Fortran provides some Fortran 90 features that increase the usefulness and/or the efficiency of subprograms:

- the **INTENT** attribute, which saves time and memory when passing array sections as arguments
- automatic arrays, which are local arrays allocated dynamically
- array-valued user functions

This chapter also describes the use of **COMMON** blocks as an alternative to argument passing.

Copyright © 1989–1994 Thinking Machines Corporation

6.1 Array Objects as Arguments

CM Fortran adopts the Fortran 90 semantics for passing CM array arguments; it retains the Fortran 77 semantics for front-end array arguments. In Fortran 77, where arrays are passed by reference, an array name as an argument indicates the array's first element. In CM Fortran, where CM arrays are passed *by descriptor*, a CM array name as an argument indicates all its elements.

CM Fortran requires that all actual and dummy array arguments match in type and home. In addition, CM array arguments must match in shape and layout. Two new forms of dummy array argument — assumed-shape and assumed-layout dummies — are useful in meeting these requirements (see Section 6.2). Interface blocks, which give the calling program information about the actual arguments a subprogram expects, permit compile-time error checking (see Section 6.3).

This section describes the Fortran 90 semantics of argument passing and the CM Fortran requirements. All the information here applies to whole arrays and to array sections as arguments. Some additional requirements for passing array sections are described in Section 6.4.

6.1.1 Semantics of CM Array Arguments

To illustrate the difference between Fortran 90 and Fortran 77 argument semantics, assume that the following array has been declared in a main program to receive some information from a database. The main program can then pass the array as an actual argument to various subroutines that sort, compute, and display the data.

```
INTEGER MY_DATA(100)
CALL SORT( MY_DATA )
...
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(100)
...
```


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

The meaning of the reference to the actual argument **MY_DATA** depends on whether the array has been established in the calling procedure as a front-end array or a CM array:

- *As a front-end array reference*, **MY_DATA** refers to the first element of the array, and the declaration of the dummy argument **ARRAY** indicates how many elements are needed.
- *As a CM array reference*, **MY_DATA** refers to all the elements of the array, and the declaration of the dummy argument must be conformable with it.

In CM Fortran, an array argument with an implicit or explicit triplet subscript — that is, a CM array name or explicit section — is passed as a pointer to the descriptor of the argument array. An array descriptor is a front-end structure that identifies the array and describes its properties, including its shape.

The distinction in argument semantics becomes more obvious if we decide to sort only part of the array. If we want to pass, say, the second half of array **MY_DATA** to subroutine **SORT**, the syntax makes it clear that the Fortran 77 argument is (the address of) a scalar value, while the Fortran 90 argument is (a pointer to) an array object:

- If **MY_DATA** is a front-end array, a scalar subscript is used:

```
CALL SORT( MY_DATA(51) )     ! Actual arg is 51st element
...
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)            ! Dummy is 50 elements
...
```

- If **MY_DATA** is a CM array, a triplet subscript is used:

```
CALL SORT( MY_DATA(51:100) ) ! Actual arg is 50 elements
...
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)            ! Dummy conforms to actual
...
```

Actual CM array arguments may not be referenced by all scalar subscripts in the Fortran 77 fashion. This kind of reference, like **MY_DATA(51)**, passes a single front-end value, not a CM array that begins at that element. The result is a run-time error.

Copyright © 1989–1994 Thinking Machines Corporation

NOTE

This chapter uses literal constants in declaring CM array arguments for the sake of clarity in comparing their shapes. Naturally, a general-purpose procedure is more likely to use variables or named constants in declaring array arguments.

6.1.2 Homes of Array Arguments

As shown in Chapter 2, the home of an array — front end or CM — is determined by *how the array is referenced* within a program unit: main program, function, or subroutine. Program units are compiled separately from one another, and the compiler carries no information about arrays (or any other object) from one compilation to another.

CM Fortran requires that actual and dummy arguments have the same home. The separate compilation of program units means that an array might come to have different homes in different program units — unless the programmer explicitly prevents this from happening.

Mismatched Homes

As an example of mismatched array homes, imagine that the main database program simply initializes **MY_DATA** with a **READ** statement and then calls subroutines **SORT**, **COMPUTE**, and so on. The compiler allocates array **MY_DATA** on the front end in the main program. However, if subroutine **SORT** uses its dummy array in a Fortran 90 array operation, the compiler expects that array to be allocated on the CM.

When at run time the front-end array is passed as an argument to **SORT**, *the result will be an error or incorrect results.* The same result occurs when a CM array is passed to a procedure that expects a front-end array. The error of mismatched array homes cannot be detected at compile time in CM Fortran.

The error of mismatched homes can, however, be detected at run time if the program is compiled with the switch **-safety=1** or any higher value.

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

Avoiding Mismatched Homes

To avoid run-time errors, the programmer must ensure that the homes of actual and dummy array arguments match. If an array is referenced in the Fortran 90 manner in any program unit, then all other program units that use that array should cause it to be allocated on the CM.

There are three ways to force an array to have a CM home:

- If appropriate, declare the array in a **COMMON** block instead of passing it as an argument. All common arrays are allocated on the CM unless otherwise specified (see Section 6.8).
- Alternatively, use the array in a Fortran 90 array operation in every program unit that references the array, even in program units where such an operation is not really needed.
- Alternatively, and somewhat more elegantly, use the compiler directive **LAYOUT** to control the home of the array in each program unit.

The **LAYOUT** directive, described in Chapter 12, serves primarily to control the mapping of array elements onto CM memory. When the directive applies the keyword **:NEWS** to any dimension of an array, that array is allocated on the CM no matter how it is used in the program unit.

For example, the following directive line, which simply specifies the default CM layout, forces **MY_DATA** to be allocated on the CM:

```
      INTEGER MY_DATA(100)
CMF$  LAYOUT MY_DATA (:NEWS)
```

Syntax Note: Directive lines must begin with **CMF$**, starting in column 1. Column position is otherwise unimportant. Spaces must separate the components of a compiler directive.

Conversely, this directive

```
      DIMENSION   C( 1000,1024 )
CMF$  LAYOUT      C( :SERIAL, :SERIAL )
```

guarantees that array **C** will be allocated entirely in front-end memory, laid out in the normal linear, column-major fashion. Such an array may not be used in a Fortran 90 array operation in any program unit in the program.

Copyright © 1989–1994 Thinking Machines Corporation

NOTE

We recommend supplying a **LAYOUT** directive for every array declared. Mismatched homes are one of the most common errors in CM Fortran programming.

When you use the **LAYOUT** directive to control array homes, avoid using the compiler switch **-nodirective**, which disables all directives in a program. By disabling **LAYOUT** directives, this switch can change array homes and thus cause a valid program to fail.

Verifying Array Homes

It may be useful during program development and debugging to check where the compiler has allocated particular arrays. The compiler switch **-list** provides this information.

This switch produces the file *program-name*.**lis**, which contains information about a number of program features. Under the "Arrays" section of the listing, the column "Home" specifies either CM or FE for each array in the compiled program. A sample listing is

```
% cmf -list my-program.fcm
% more my-program.lis
. . .
ARRAYS
  Offset    Size  Type  Block/Class     Home      Name
     ---     10*  L*4   local           CM        ARRAY_1
     ---      40  R*4   local           CM        ARRAY_2
       0      80  C*8   local           FE        ARRAY_3
     120      40  I*4   local           FE        ARRAY_4
```

See the *CM Fortran User's Guide* for explanation of the other entries in this listing.

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

6.1.3 Types and Shapes of Array Arguments

Actual array arguments must always be of the same type as their corresponding dummy arguments in a subprogram. For both CM arrays and front-end arrays, mismatched types can result in a run-time error or incorrect results.

If the actual and dummy arguments are CM arrays, they must also be of the same shape. In CM Fortran (unlike Fortran 90), it is an error to reshape a CM array across procedure boundaries.

Reshaping Arrays

CM Fortran programmers must avoid the common Fortran 77 practice of reshaping arrays across program boundaries, such as

```
INTEGER MY_DATA(N,M)          ! Actual is 2D
...
CALL RANK_IT( MY_DATA )       ! Error: shape mismatch
...
END

SUBROUTINE RANK_IT( ARRAY )
INTEGER ARRAY(N*M)            ! Dummy is 1D
...
```

This practice assumes that array elements have a prescribed storage order so that particular elements can be mapped between two differently shaped arrays in a predictable way. This *sequence association* is true of front-end arrays but not of CM arrays.

It is preferable instead to declare all arrays separately, each in the desired shape. If you need to reshape a particular array — for instance, to change **A(N,M)** to **A(N*M)** — use the intrinsic function **RESHAPE** (see Chapter 9). Advanced users who know the array's exact layout in memory may choose to *alias* **A_1(N*M)** to **A(N,M)**. See the *CM Fortran Libraries Reference Manual* for information on array aliasing.

Copyright © 1989–1994 Thinking Machines Corporation

"Hidden" Array Arguments

CM Fortran programmers must also avoid the common Fortran 77 practice of allocating a large chunk of memory and then indexing into it to select pieces of memory as needed. This practice assumes that arrays are passed by reference to the first element and that you can reshape an array across program boundaries. Both assumptions are true of front-end arrays, but not of CM arrays.

If your intention is to manage memory dynamically, it is better to use the CM Fortran dynamic allocation features (Chapter 7), which allow you to allocate and deallocate arrays as needed.

Array Arguments with Elided Axes

Another Fortran 77 practice to avoid is passing an *n*-minus-*k*-dimensional contiguous slice of an *n*-dimensional array to subprograms in the following way: all indices in the call other than rightmost *k* indices are the lower bound of the corresponding dimension.

```
REAL X(100,200,10)
CALL SUB_3(X(1,1,3))
...
SUBROUTINE SUB_3(Y)
REAL Y(100,200)
...
```

This practice violates CM Fortran argument-passing restrictions by passing a scalar value as an array argument. To express the same intent correctly, pass a 2-dimensional section of the 3-dimensional array. The lower bound indices turn into colons, indicating the whole dimension.

```
REAL X(100,200,10)
...
CALL FOO(X(:,:,3))
...
SUBROUTINE FOO(Y)
REAL Y(100,200)
...
```

NOTE: You can improve the performance of this kind of argument passing by declaring the axes with the scalar indices as **`:SERIAL`** in a **`LAYOUT`** directive. See the *CM-5 CM Fortran Performance Guide*.

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

6.1.4 Layouts of Array Arguments

If a directive changes the layout of a CM array — as opposed to simply controlling its home — that array is referred to as a *noncanonical* array. When the array is passed to a subprogram, the dummy argument must have the same noncanonical layout as the actual argument.

Like mismatched shape or home, mismatched array layouts across program boundaries cause a run-time error or incorrect results. The subprogram mistakes the location of particular array elements and garbage elements. For example, this fragment passes a CM array **x** to a subroutine that expects a noncanonical array. The calling program has no layout directive for **x**; the array operation does establish it as a CM array, but it is given the canonical layout.

```
      REAL X(100,200)
      ...
      X = [ some array operation ]
      CALL FOO(X)
      ...
      SUBROUTINE FOO(Y)       ! Error: layout mismatch
      REAL Y(100,200)
CMF$  LAYOUT Y(:SERIAL,:NEWS)
      ...
```

6.1.5 Avoiding Argument-Passing Errors

The CM Fortran compiler does not perform interprocedural analysis. It cannot, therefore, detect argument mismatches in type, home, shape, or layout.

The **-safety** switch generates run-time checks for array home and rank. It does not detect mismatches in type, shape, or layout. The **-list** switch reports the type and home of all arrays at compile time.

Two methods you can use to help avoid argument-passing errors are described in the following sections:

- Declare dummy CM array arguments such that they will assume the shape and/or layout of the actual argument passed (Section 6.2).

Copyright © 1989–1994 Thinking Machines Corporation

- Provide *interface blocks* in the calling procedure that show the type, shape, and (if noncanonical) the layout of the arrays the subprogram expects (Section 6.3). Interface blocks enable the compiler to check for argument mismatches.

NOTE: Some additional constraints on passing array sections as arguments make interface blocks mandatory for many noncanonical array sections. See Section 6.4.

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

6.2 Declaring Dummy Array Arguments

A dummy CM array can be either *explicit-shape* or *assumed-shape*, depending on how much information its declaration provides about its dimensions. A particular form of the **LAYOUT** directive can also make a dummy argument assumed-layout.

6.2.1 Explicit-Shape Dummies

An explicit-shape dummy array is one whose dimension sizes (or bounds) are all declared, either with constants or with dummy variables. If a dimension or bound is declared with a variable (whose value is not known until run time), the dummy array is called an *adjustable* array.

```
SUBROUTINE SUB1( A, B, C, D, E, F )

REAL A(100,100)       ! explicit-shape
REAL B(-10:5, 0:100) ! explicit-shape
INTEGER C,D
REAL E(C,D)           ! explicit-shape, adjustable
REAL F(-10:20, C:D)   ! explicit-shape, adjustable
```

An explicit-shape dummy array may be allocated either on the front end or on the CM, depending on how it is used in the procedure or constrained by a **LAYOUT** directive.

Any actual argument passed to an explicit-shape dummy argument must match it in type and home. If the dummy is a CM array in the subprogram, the actual argument must be established in the calling program as a CM array of the same shape and layout.

6.2.2 Assumed-Shape Dummies

An assumed-shape dummy array has an explicit rank, but all its dimension extents are unspecified. The dummy array assumes the shape of the actual argument passed. An assumed-shape dummy is a CM array, regardless of how it is used, and the corresponding actual argument must be established in the calling routine as a CM array of the same rank.

Copyright © 1989–1994 Thinking Machines Corporation

Some examples of assumed-shape dummy arrays are:

```
SUBROUTINE SUB2( F, G, H, N )

REAL F(:,:,:)
REAL G( 0:, 100: )
INTEGER N
REAL H( N: )
```

Note that no upper bounds are specified; declaring **`A(N,:,:)`** is illegal.

6.2.3 Assumed-Size Dummies

Note the difference in syntax between an assumed-*shape* dummy array and an assumed-*size* dummy array, where the last dimension depends on the size of the actual argument passed:

```
SUBROUTINE SUB3( CM_ARRAY, FE_ARRAY )

REAL CM_ARRAY(:,:)      ! assumed-shape; CM array
REAL FE_ARRAY(100,*)    ! assumed-size; FE array
```

Assumed-size arrays are a feature CM Fortran draws from Fortran 77; only front-end arrays can be assumed-size. Assumed-shape arrays are a feature CM Fortran draws from Fortran 90; only CM arrays can be assumed-shape.

> **Syntax Note:** Assumed-size arrays cannot be declared with Fortran 90 attributed type declarations. Use only Fortran 77−style declarations for them.

CM Fortran does not support assumed-size dummy CM arrays because Fortran 77 permits collapsing multiple axes into the **`*`** axis. This violates the restriction against reshaping CM arrays across program boundaries. Use assumed-size dummy arrays only for front-end array arguments. For CM array arguments, use assumed-shape dummies.

For example, when a procedure uses a dummy array as a CM array, don't do this:

```
SUBROUTINE SUB(A)
REAL A(1024,*)
```


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

Do this instead:

```
      SUBROUTINE SUB(A)
      REAL A(:,:)
```

6.2.4 Assumed-Layout Dummies

The compiler directive **LAYOUT** accepts an assumed-layout axis descriptor, specified with a colon for each axis. When this form of **LAYOUT** is supplied for an assumed-shape or adjustable dummy array, the dummy assumes the layout of the actual argument passed. An assumed-layout dummy array is constrained to a CM home, and only CM actual arguments may be passed to it.

See Section 6.1.2 above or Chapter 12 for more about **LAYOUT** syntax.

For example, the following program passes a noncanonical argument to two subroutines that use assumed-shape dummies. The second subprogram fails because of a layout mismatch.

```
      PROGRAM PROG1
      DOUBLE PRECISION B(1000,1000,1000)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
      ...
      CALL SUB_WORKS(B)
      CALL SUB_BREAKS(B)
      ...
      END PROGRAM PROG1

      SUBROUTINE SUB_WORKS(A)
      DOUBLE PRECISION A(:,:,:)
CMF$  LAYOUT A(:,:,:)
      ...
      END SUBROUTINE SUB_WORKS

      SUBROUTINE SUB_BREAKS(A)
      DOUBLE PRECISION A(:,:,:)   ! Default layout
      ...
      END SUBROUTINE SUB_BREAKS
```


Copyright © 1989–1994 Thinking Machines Corporation

The following example uses the assumed-layout directive to write two general-purpose subroutines that can be used with any possible layout of the array **A**. Subroutine **MM_2** uses the assumed-layout directive with an adjustable array, and subroutine **MM_3** uses it with an assumed-shape array.

```
      SUBROUTINE MM(A)
      REAL A(1024,1024)

      CALL MM_2(A,1024,1024)
      CALL MM_3(A)
      ...
      END

      SUBROUTINE MM_2(A,N,M)
      REAL A(N,M)           ! adjustable array
CMF$  LAYOUT A(:,:)
      ...
      END

      SUBROUTINE MM_3(A)
      REAL A(:,:)           ! assumed-shape array
CMF$  LAYOUT A(:,:)
      ...
      END
```

NOTE: The assumed-layout directive may *not* be used on an assumed-size dummy array (which is constrained to a front-end home), nor may it be used for the return value of array-valued functions (see Section 6.5).

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

6.3 Making Subroutine Interfaces Explicit

The separate compilation of program units means that the compiler cannot ordinarily compare the properties of arguments across program boundaries. However, the compiler can check for mismatches if you make the subprogram interface explicit to the calling routine.

The interface is made explicit by means of an *interface block*, which duplicates the subprogram's interface within the calling routine. For example, recall the interface to subroutine **SORT**:

```
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(100)
```

To make that interface explicit to the main program that calls subroutine **SORT**, insert the following interface block into the *specification part* of the main program:

```
INTERFACE
    SUBROUTINE SORT( ARRAY )
    INTEGER ARRAY(100)
END INTERFACE
```

Interface blocks are not required in CM Fortran for most argument-passing situations (the exceptions are listed below). You may, however, find them useful as a safety measure during program development.

The interface block shown contains no information on the home of the argument array, since the home is determined by the presence or absence of an array operation in the executable part of the program. Thus, the compiler cannot detect an error of mismatched array homes.

If, however, you use a **LAYOUT** directive to determine the array's home and if you use interface blocks, the compiler can detect mismatches in array home and layout. The directive line must appear in the interface block, as well as in the declaration of the dummy argument.

```
          INTERFACE
               SUBROUTINE SORT( ARRAY )
               INTEGER ARRAY(100)
CMF$           LAYOUT ARRAY (:NEWS)
          END INTERFACE
```


Copyright © 1989–1994 Thinking Machines Corporation

For example, consider four different programs that call this subroutine:

```
      SUBROUTINE SUB1(A)
      DOUBLE PRECISION A(1000,1000,1000)
CMF$  LAYOUT A(:NEWS,:SERIAL,:NEWS)
      ...
      END
```

The first program (below) works — the type, shape, home, and layout of the actual argument match those of the dummy — but it does not permit compile-time checking for mismatches:

```
      PROGRAM WORKS
      DOUBLE PRECISION B(1000,1000,1000)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
      ...
      CALL SUB1(B)
      ...
      END PROGRAM WORKS
```

The second program fails at run-time because of a layout mismatch that the compiler could not detect. There may also be a home mismatch if array **B** is not used in an array operation and thus has a front-end home in the caller.

```
      PROGRAM BREAKS
      DOUBLE PRECISION B(1000,1000,1000)
      ...
      CALL SUB1(B)
      ...
      END PROGRAM BREAKS
```

The third program also fails. Its interface block neglects to include the subroutine's **LAYOUT** directive:

```
      PROGRAM BREAKS_ALSO
      DOUBLE PRECISION B(1000,1000,1000)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
      INTERFACE
          SUBROUTINE SUB1(A)
          DOUBLE PRECISION A(1000,1000,1000)
      END INTERFACE
      ...
      CALL SUB1(B)
      ...
      END PROGRAM BREAKS_ALSO
```


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

The fourth program illustrates the recommended way to write a program that calls **SUB1**, if you want the compiler to check for argument mismatches:

```
      PROGRAM WORKS_BEST
      DOUBLE PRECISION B(1000,1000,1000)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
      INTERFACE
          SUBROUTINE SUB1(A)
          DOUBLE PRECISION A(1000,1000,1000)
CMF$      LAYOUT A(:NEWS,:SERIAL,:NEWS)
      END INTERFACE
      ...
      CALL SUB1(B)
      ...
      END PROGRAM WORKS_BEST
```

Interface blocks are required for correct program behavior in three cases, discussed elsewhere in this manual:

- when passing noncanonical array sections (Section 6.4.2)
- when invoking a user-defined array-valued function (Section 6.5.3)
- when passing an array pointer to a dummy array pointer (Chapter 7)

You should also use interface blocks when using the **INTENT** attribute (Section 6.4.4), or the attribute has no effect.

Copyright © 1989–1994 Thinking Machines Corporation

6.4 Array Sections as Arguments

The example at the beginning of this chapter showed a section of array **MY_DATA** passed as an argument to subroutine **SORT**. To sort only the second half of the data:

```
INTEGER MY_DATA(100)
. . .
CALL SORT( MY_DATA(51:100) ) ! Actual is 50 elements
. . .
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)            ! Dummy conforms to actual
. . .
```

Similarly, to select and sort every fourth data element, pass a section with a stride of 4:

```
CALL SORT( MY_DATA(1:100:4) ) ! Actual is 25 elements
. . .
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(25)             ! Dummy conforms to actual
. . .
```

Array sections can be used anywhere that whole CM arrays are used, with no difference in the semantics of the array reference. (Recall that a reference to a whole array is simply the default form of a triplet subscript.) However, the implementation of array-section arguments is different from whole-array arguments, and you might need to take extra steps to ensure correctness or good performance.

6.4.1 Passing Array Sections

Array sections differ from whole arrays in the way the system passes them as arguments. Whole arrays are passed in place, meaning that the argument is a pointer to the array's descriptor. In contrast, array sections as arguments are first copied (using an expensive **send** operation) to a temporary location. The actual argument is a pointer to a descriptor for that temporary array. If the subprogram alters the array section, then it is also copied back into the original (parent) array upon exit from the subprogram.

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

(Some sections taken from an array that is subject to a **LAYOUT** directive are passed in place. See Section 6.4.3 for a discussion of this special case.)

The listing file produced by the **-list** option reports whether arguments are copied to temporaries.

One way to avoid any needless copying is to avoid passing array sections as arguments when possible. If the algorithm permits, you can pass the whole array and have the subroutine select the desired section. For example,

```
INTEGER MY_DATA(100)

INTERFACE                     ! Optional interface block
    SUBROUTINE SORT( ARRAY )
    INTEGER ARRAY(100)
END INTERFACE
. . .
CALL SORT( MY_DATA )         ! Actual arg is 100 elements
. . .
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(100)            ! Dummy conforms to actual
ARRAY(1:100:4) = [some sorting algorithm] ! Operate on section
END
```

Other ways to avoid needless copying are to use the **INTENT** attribute (Section 6.4.4) or to pass array sections that are contiguous in memory (Section 6.4.3).

6.4.2 Passing Noncanonical Array Sections

The temporary array to which an array-section argument is copied is given a canonical layout, even if the section is taken from a noncanonical parent array. If the subprogram expects a canonical argument (that is, if it has no **LAYOUT** directive on the dummy array), all is well. If, however, the subprogram expects a noncanonical argument — even one whose layout matches the section's parent — the result is a layout mismatch and a run-time error.

Copyright © 1989–1994 Thinking Machines Corporation

For correct program behavior, you must supply an interface block whenever you pass an array section to a subprogram that expects a noncanonical argument.

For example, the following program fails, even though the actual and dummy arguments match in shape and layout. The section is passed as a pointer to a canonical 500 x 1000 x 1000 array.

```
      PROGRAM FAILS
      REAL B(1000,1000,1000)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
      ...
      CALL SUB(B(:500,:,:))
      ...
      END

      SUBROUTINE SUB(A)
      REAL A(500,1000,1000)
CMF$  LAYOUT A(:NEWS,:SERIAL,:NEWS)
      ...
      RETURN
      END
```

The following program works correctly, since the interface block tells the compiler that the subroutine expects a noncanonical actual argument. The compiler can then give the temporary array the expected layout.

```
      PROGRAM WORKS
      REAL B(1000,1000,1000)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
      INTERFACE
          SUBROUTINE SUB(A)
          REAL A(500,1000,1000)
CMF$      LAYOUT A(:NEWS,:SERIAL,:NEWS)
      END INTERFACE
      ...
      CALL SUB(B(:500,:,:))
      ...
      END
```


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

```
      SUBROUTINE SUB(A)
      REAL A(500,1000,1000)
CMF$  LAYOUT A(:NEWS,:SERIAL,:NEWS)
      ...
      RETURN
      END
```

6.4.3 Passing Contiguous Array Sections

One exception to the rule that array-section arguments are copied to a temporary location is sections that reference a contiguous region of CM memory. Contiguous sections are passed in place, like whole arrays, and thus do not require interface blocks for correct behavior.

```
      PROGRAM WORKS
      REAL C(5,10,10)
CMF$  LAYOUT C(:SERIAL,:NEWS,:NEWS)
      ...
      CALL SUB2(C(3,:,:))
      ...
      END
```

Most commonly, a contiguous region is specified with scalar indices for all **:SERIAL** axes and colons for all **:NEWS** axes. Several other kinds of references indicate a contiguous section, and the setting of the switch **-[no]axisreorder** may cause the stated rule not to apply. See the *CM-5 CM Fortran Performance Guide* for details.

Interface blocks are not required when passing contiguous array sections. You may want to use them to facilitate error checking.

Copyright © 1989–1994 Thinking Machines Corporation

6.4.4 The INTENT Attribute

When non-contiguous array sections are passed as arguments, the **INTENT** attribute can increase program performance by preventing needless copying of data between the parent array and the temporary array created for the section. The **INTENT** attribute, applied to any dummy argument, indicates whether that argument will be used to receive data from, or return data to, the calling procedure, or both.

One way to apply the attribute to a dummy array is the **INTENT** statement:

```
SUBROUTINE SUB4( A,B,C )
REAL A(100,100), B(100,100), C(100,100)

INTENT(OUT) :: A  ! A values will only be written out
INTENT(IN)  :: B  ! B values will only be read in
INTENT(INOUT) :: C ! C values will be read and written
```

If the **INTENT** statement indicates that the dummy will not receive data from the calling procedure, then the system will not copy in the values of the actual argument. If the **INTENT** statement indicates that the dummy will not return data to the calling procedure, then the system will not copy values back into the original array.

Alternatively, the **INTENT** attribute can be applied to a dummy array in a Fortran 90 attributed type declaration:

```
SUBROUTINE SUB5( D,E,F )
REAL, DIMENSION(100,100), INTENT(IN)  :: D,E
REAL, DIMENSION(100,100), INTENT(OUT) :: F
```

Since the calling procedure needs this information about the intended use of the dummy argument, you must provide an interface block when specifying an **INTENT** attribute, and the interface block must contain the **INTENT** statement or attribute declaration. Without an interface block, the **INTENT** statement or attribute has no effect.

6.4.5 Example of INTENT

Imagine that a main game-playing program does not initialize an array, but instead simply passes half the array to subroutine **SORT** to be initialized. If the dummy array in the subroutine has the attribute **INTENT(OUT)**, the sys-

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

tem will not bother to copy in the values from the original array, but it will copy the sorted values back out to the original array:

```
INTEGER TOKENS(100)
INTERFACE                ! Interface block required
    SUBROUTINE SORT( ARRAY )
    INTEGER ARRAY(50)
    INTENT(OUT) :: ARRAY
END INTERFACE
. . .
CALL SORT( TOKENS(1:50) )
. . .
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)
INTENT(OUT) :: ARRAY
. . .
```

Subroutine **PLAY**, on the other hand, needs to receive the values of the sorted tokens *and* pass the values back to the main program so that they can be passed on to subroutine **DISPLAY**. Thus, **PLAY** can give the dummy argument the attribute **INTENT(INOUT)**, although the effect on copying behavior is the same as not specifying the **INTENT** attribute.

Finally, subroutine **DISPLAY** needs to receive the values as computed by **PLAY**, but it need not store the final state of the tokens back into the original array. This subroutine therefore gives the array the attribute **INTENT(IN)**:

```
INTEGER TOKENS(100)
INTERFACE                    ! Interface block required
    SUBROUTINE DISPLAY( ARRAY)
    INTEGER ARRAY(50)
    INTENT(IN) :: ARRAY
END INTERFACE
. . .
CALL DISPLAY( TOKENS(1:50) )
. . .
```


Copyright © 1989–1994 Thinking Machines Corporation

```
END

SUBROUTINE DISPLAY( ARRAY )
INTEGER ARRAY(50)
INTENT(IN) :: ARRAY
. . .
```


Copyright © 1989–1994 Thinking Machines Corporation

6.5 User-Defined Functions

All the considerations that this chapter describes for defining and calling subroutines apply also to user-defined functions. This section discusses some additional features of user functions in CM Fortran.

6.5.1 Statement Functions

CM Fortran supports Fortran 90 *statement functions,* which define a function in a single statement. A statement function can operate only on scalar data (that is, it cannot use array operations), and it can be invoked only within the program unit where it is defined. See the *CM Fortran Language Reference Manual* for examples.

6.5.2 Scalar-Valued Functions

When a function takes CM array arguments, its returned value can be defined as either a scalar or an array. An example of a function that returns a scalar is:

```
INTEGER FUNCTION AVERAGE( ARRAY )
INTEGER ARRAY (:)

AVERAGE = SUM( ARRAY ) / SIZE( ARRAY )
END
```

The argument passed to this function must be a CM array. The dummy **ARRAY** is assumed-shape and is passed as an argument to the intrinsic reduction function **SUM**. Either of these factors would be sufficient to cause the dummy to be a CM array. The functions **SUM** and **SIZE** return their scalar results to the control processor, which executes the division operation.

6.5.3 Array-Valued Functions

The behavior of an array-valued function is like that of a subroutine that takes an array as an argument and stores its results there. It is often simply a matter of style whether to write a procedure as a subroutine or a function.

Copyright © 1989–1994 Thinking Machines Corporation

The result of an array-valued function must be a CM array. The function body must force the array onto the CM with an array operation or **LAYOUT** directive. The result array may not be assumed-layout (Section 6.2.4) and it may not be deferred-shape (a pointer or allocatable array, as described in Chapter 7). In addition, the program unit that invokes the function must include an interface block for the function (Section 6.3).

The following example shows a user-defined array-valued function whose size is not known until run time:

```
      FUNCTION IOTA(N,M)
      INTEGER N,M
      INTEGER IOTA(N,M)
      ...
      IOTA = 300          ! CM array operation
      END

      PROGRAM FOO
      INTEGER A(10,2048)

      INTERFACE           ! interface block required
        FUNCTION IOTA(N,M)
        INTEGER IOTA(N,M)
      END INTERFACE

      A = IOTA(10,2048)
      END
```

The compiler also supports user-defined functions returning a noncanonical array, as in this example:

```
      FUNCTION IOTA()
      INTEGER IOTA(10,2048)
CMF$  LAYOUT IOTA(:SERIAL, :NEWS)
      ...
      IOTA = 300
      RETURN
      END
```


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

```
      PROGRAM FOO
      INTEGER A(10,2048)

      INTERFACE
        FUNCTION IOTA()
        INTEGER IOTA(10,2048)
CMF$    LAYOUT IOTA(:SERIAL, :NEWS)
      END INTERFACE

      A = IOTA()
      END
```


Copyright © 1989–1994 Thinking Machines Corporation

6.6 Declaring Automatic Arrays

An array that is local to a subroutine or function is temporary: it is allocated upon entry to the procedure and deallocated upon exit from the procedure. A local array can be declared with explicit bounds, or it can be made to conform to an adjustable or assumed-shape dummy array.

In cases where a dummy array is non-adjustable, the declaration of a conformable local array is straightforward. As in Fortran 77, the local array is declared with the same constant size (or bounds) as the dummy array. For example:

```
SUBROUTINE SUB4( A )

REAL A(100,100)
REAL TEMP(100,100)
```

When a local CM array needs to conform to an adjustable dummy array, the local is declared with the same variables that specify the shape of the dummy. Such a local array is called an *automatic* array; space for it is allocated at run time on the CM. The array **TEMP** in the following fragment is an automatic array:

```
SUBROUTINE SUB5( B,I,J )

INTEGER I,J
REAL B(I,J)           ! Dummy B is adjustable
REAL TEMP(I,J)        ! Automatic TEMP conforms to B
TEMP = B              ! TEMP and B are CM arrays
```

You can also declare an automatic array to be conformable with an assumed-shape dummy array. Doing so requires finding, at run time, the dimensions of the corresponding dummy array. The intrinsic functions that return this information are shown in Section 6.7.

Automatic arrays need not be associated with dummy arguments, of course. Array **TEMP** in the following fragment is also a CM automatic array. It is allocated at run time, with its shape dependent on the values of the actual arguments **I** and **J**.

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

```
SUBROUTINE SUB4( I,J )

INTEGER I,J
REAL TEMP(I,J)          ! TEMP is automatic
TEMP = TEMP + 1         ! TEMP is a CM array
```

Automatic arrays are useful, for example, in writing a program that scales at run time to the size of a data set. The following program reads from a file the information needed to determine array size. It then calls a subprogram, which uses the size data to specify the dimensions of two automatic arrays **A** and **B**.

```
PROGRAM ANY_SIZE
INTEGER N,M

READ (*,*) N,M          ! Get size
CALL DUMMYMAIN( N,M )

END

SUBROUTINE DUMMYMAIN( N,M )
INTEGER N,M
REAL A(N,M), B(N,M)     ! automatic arrays

. . . code that uses A and B in array operations . . .

RETURN
END
```

Alternatively, you can achieve this effect with the CM Fortran dynamic array allocation features, described in Chapter 7.

Copyright © 1989–1994 Thinking Machines Corporation

6.7 Retrieving Array Properties

CM Fortran provides a number of intrinsic functions that inquire about the size, shape, or bounds of an array or one of its dimensions. These functions are **SIZE**, **SHAPE**, **RANK**, **UBOUND**, and **LBOUND**.

> **NOTE:** For back-compatibility, CM Fortran also accepts the earlier names, **DSIZE**, **DSHAPE**, **DUBOUND**, and **DLBOUND**.

For convenience, the following examples illustrate these functions in relation to arrays whose shapes are declared — a rather pointless exercise. These functions are particularly useful in relation to adjustable dummy array arguments, assumed-shape arrays, and automatic arrays — that is, arrays whose size and shape are not established until run time.

Unlike the array-processing intrinsics, these inquiry functions can operate on front-end arrays as well as CM arrays.

6.7.1 Array Shape

The function **SHAPE** takes an array (or section) and returns a vector whose elements are the lengths of the array's dimensions. The result is returned on the front end (regardless of the home of the argument array). For example,

```
REAL, DIMENSION (5,10) :: A

PRINT *, SHAPE( A )                    ! Prints [ 5,10 ]
PRINT *, SHAPE( A(3,:) )               ! Prints [ 10 ]
```

SHAPE is useful when retrieving the properties of adjustable and assumed-shape arrays; it cannot be used with assumed-*size* arrays. Alone among the inquiry functions, **SHAPE** can also take a scalar argument. In this case, the value returned is an array of size zero (not the scalar zero).

6.7.2 Array Size

The function **SIZE** takes an array and an optional dimension argument. It returns a scalar that is the number of elements in the whole array (or section) or in the specified dimension. For example,

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

```
REAL, DIMENSION (5,10) :: A

PRINT *, SIZE( A )                      ! Prints 50
PRINT *, SIZE( A(3,:) )                 ! Prints 10
PRINT *, SIZE( A, DIM=1 )               ! Prints 5
```

If the argument is an assumed-size array, **SIZE** cannot retrieve the size of the last dimension — or, therefore, the size of the whole array. For assumed-size arrays, the dimension argument must be supplied and it must not be the last dimension. For example,

```
DIMENSION B(5,*)

PRINT *, SIZE( B, DIM=1 )               ! Prints 5
PRINT *, SIZE( B, DIM=2 )               ! Not supported
PRINT *, SIZE( B )                      ! Not supported
```

NOTE: The Fortran 90 attributed declaration syntax is not accepted for assumed-size arrays. Array **B** may not be declared as

```
REAL, DIMENSION (5,*) :: B              ! Not supported
```

6.7.3 Dimension Bounds

The functions **UBOUND** and **LBOUND** take an array and return, as a front-end vector, all its upper or lower bounds. For example,

```
REAL, DIMENSION (5,10) :: A
REAL, DIMENSION (-10:100, 20:50) :: C

PRINT *, UBOUND( A )                ! Prints [ 5,10 ]
PRINT *, LBOUND( A )                ! Prints [ 1,1 ]
PRINT *, LBOUND( C )                ! Prints [ -10,20 ]
PRINT *, UBOUND( C(-10:0,:) )       ! Prints [ 0,50 ]
```

If the dimension argument is supplied, these functions return a scalar that is the requested bound of that dimension:

```
REAL, DIMENSION (-10:100, 20:50) :: C

PRINT *, UBOUND( C, DIM=1 )         ! Prints 100
PRINT *, LBOUND( C, DIM=2 )         ! Prints 20
PRINT *, LBOUND( C(40:,:40), DIM=1)! Prints 40
```


Copyright © 1989–1994 Thinking Machines Corporation

If the argument is an assumed-size array, **UBOUND** cannot retrieve the upper bound of the last dimension — or, therefore, those of the whole array. For assumed-size arrays, the dimension argument must be supplied and it must not be the last dimension. For example,

```
DIMENSION B(5,*)

PRINT *, UBOUND( B, DIM=1 )       ! Prints 5
PRINT *, UBOUND( B, DIM=2 )       ! Not supported
PRINT *, UBOUND( B )              ! Not supported
```

6.7.4 Inquiry Example

All the inquiry intrinsic functions can be used in specification statements as well as in executable statements.The function **UBOUND** is particularly useful for finding the upper bounds of an assumed-shape dummy array. This information can be used to declare an automatic (temporary) array that conforms to the dummy array:

```
SUBROUTINE SUB7( A )
REAL A(:,:)
REAL TEMP( UBOUND( A,1 ), UBOUND( A,2 ))

TEMP=A     ! Automatic TEMP and dummy A are CM arrays
```

The two arrays in this subroutine would be taken by the compiler to be CM arrays even if they weren't used in an array assignment. Array **A** is a CM array because it is assumed-shape; array **TEMP** is a CM array because it is automatic. The actual argument passed to **A** must be established as a CM array in the calling routine.

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

6.8 Common Arrays

CM Fortran allows any array to be placed in a common block, either on the CM or on the control processor, and then be shared by program units. However, the homes of common arrays are determined differently from those of local arrays, and the homes constrain the ways in which the arrays can be used. It is not possible to allocate a given common block in both the serial and parallel memories.

6.8.1 Common Array Homes

The homes of common arrays are determined in the specification part of the program — *not* by how the arrays are used. Programmers must take care that the use made of a particular common array in every program unit is appropriate to its home:

- A common array that is allocated on the control processor cannot be used in an array operation anywhere in the program.
- A common array that is allocated on the CM can be referenced in the Fortran 77 manner (with scalar subscripts rather than triplet subscripts), but the operation is likely to be very slow because the system copies the array values one at a time between system components.

  The utility subroutines **CMF_FE_ARRAY_TO/FROM_CM** can perform a faster bulk data transfer for those cases where it is necessary to use any CM array in a serial operation. (See the *CM Fortran Libraries Reference Manual* for information on utility routines.)
- A common array that is allocated on the CM cannot be used by a program unit or library procedure that was compiled by a foreign compiler.

All common arrays retain their values when control passes from one program unit to the next. You can apply the **SAVE** attribute to a common block (though not to an individual array within a common block). This action is not necessary, however, since the CM system retains values in **COMMON** in any case.

Copyright © 1989–1994 Thinking Machines Corporation

6.8.2 Declaring Common Arrays

Like local arrays, common arrays can be declared either with Fortran 77 syntax or with Fortran 90 attributed type declarations (as described in Chapter 4), with no difference in the semantics of the specification.

Any array (or scalar) can be placed in a block of common storage by means of the **COMMON** statement:

COMMON [/*block-name*/] *var-list*

Unlike local arrays, common arrays are allocated by default on the CM. The home then determines the permissible uses of the common array, rather than vice versa. This section describes the various methods of overriding the default CM allocation of common arrays.

Character Type and LAYOUT Directive

A common array is allocated on the control processor if it is of type character or if it is constrained to a front-end home by the compiler directive **LAYOUT**. All the following arrays are front-end arrays and cannot be used in array operations:

```
      CHARACTER A          ! Character type is FE only

      REAL, DIMENSION(100,100) :: B
CMF$  LAYOUT B(:SERIAL,:SERIAL)   ! all :SERIAL is FE

      COMMON /BLOCK_1/ A(10),B    ! block is FE
```

COMMON Directive

The compiler directive **COMMON** can specify the home of a common block by designating the block as **FEONLY** or **CMONLY**. Both the following arrays are allocated on the front end and cannot be used in array operations.

```
      REAL, DIMENSION(100,100) :: D,E
      COMMON /BLOCK_2/ D,E

CMF$  COMMON FEONLY /block_2/
```


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

The **COMMON** directive has no effect on an array's home if the home is already constrained by **LAYOUT** or if the array is of type character.

Compiler Switches

The CM Fortran compiler switch **-fecommon** places common blocks on the front end, overriding the default placement of common blocks on the CM. This switch does *not* affect arrays that are constrained to a CM home by the directives **LAYOUT** or **COMMON**.

Another compiler switch, **-nodirectives**, disables all the compiler directives in the program. Using this switch causes all arrays and common blocks to revert to their default homes: character arrays (as always) to the front end, all non-character common arrays to the CM, and all non-character local arrays to one machine or the other depending on how they are used. This switch should be used with care, since, in possibly changing array homes, it can cause valid programs to fail.

6.8.3 Initializing Common Arrays

The initialization techniques described in Chapter 4 for local arrays apply also to common arrays: any array, front-end or CM, local or common, can be initialized by means of a **DATA** statement or a **DATA** attribute associated with an array constructor in the type declaration.

However, some special actions are needed to initialize a CM common array "statically," that is, with a **DATA** statement or attribute. In fact, there is no static initialization of CM arrays, since CM space is not allocated until run time when the number of parallel processors is known. When the system encounters a **DATA** statement or attribute for a CM array, it places the values in front-end storage and moves them to the CM at run time.

For local arrays, this action happens transparently; the programmer need only ensure that there is adequate front-end space available for storing the CM values. For common arrays, however, the programmer must do the following to ensure that the necessary storage is allocated on the front end:

Copyright © 1989–1994 Thinking Machines Corporation

- Initialize CM common arrays in a **BLOCK DATA** program unit. As with Fortran 77 common arrays, the **DATA** statement or attribute that initializes a CM common array cannot appear in a main program or external procedure, and the common block must be named.

- Indicate to the compiler that CM common arrays will be statically initialized. This can be done either with the compiler switch **-common_initialized** or with another form of the compiler directive **COMMON**:

  ```
  CMF$   COMMON INITIALIZED /block-name/
  ```

  This form of the **COMMON** directive indicates that the common block is to reside on the CM (the default home in any case) and that front-end storage is to be allocated for its initial values.

The **COMMON** directive indicating that a CM common array is initialized must appear in the **BLOCK DATA** program unit and in all program units where the array is used. For example,

```
      PROGRAM CM_DATA
      REAL A(1024)
      COMMON /CMBLK/ A
CMF$ COMMON INITIALIZED /CMBLK/

      PRINT *, A(1)

      STOP
      END

      BLOCK DATA
      REAL A(1024)
CMF$ COMMON INITIALIZED /CMBLK/
      COMMON /CMBLK/ A
      DATA A/1024*0.2/
      END
```


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