The
Connection Machine
System

# CM Fortran Utility Library Reference Manual

Preliminary Documentation for Version 2.0 Beta

January 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, January 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

# Contents

Field Test Support .......... vi

About This Manual .......... vii

**Chapter 1 Introduction .......... 1**

1.1 Why a Utility Library? .......... 1

1.2 Contents of the Utility Library .......... 2

1.3 The Utility Library Header File .......... 3

1.4 Restrictions on Utility Procedures .......... 3

**Chapter 2 Inquiries, Random Numbers, and Dynamic Allocation .......... 5**

2.1 System Inquiry Functions .......... 5

2.1.1 Language Comparison .......... 6

2.2 Array Inquiry Subroutine .......... 7

2.2.1 Language Comparison .......... 7

2.3 Random Number Generation .......... 8

2.3.1 Language Comparison .......... 8

2.4 Dynamic Array Allocation .......... 9

2.4.1 Allocation Example .......... 10

2.4.2 Controlling Array Layout .......... 11

2.4.3 Detailed-Layout Allocation Example .......... 11

2.4.4 Restrictions .......... 13

2.4.5 Language Comparison .......... 13

**Chapter 3 Data Motion .......... 15**

3.1 Array Transfers .......... 15

3.1.1 Language Comparison .......... 16

3.2 Scatters with Combining .......... 17

3.2.1 Constructing Send Address Arrays .......... 17

3.2.2 Address-Construction and Scatter Example .......... 19

3.2.3 Language Comparison .......... 20

3.3 Parallel Prefix Operations ........ 21
3.3.1 Scan Segments ........ 22
3.3.2 Scanning Example ........ 23
3.3.3 Language Comparison ........ 24
3.4 Ranking and Sorting ........ 25
3.4.1 Axis Segments ........ 25
3.4.2 Ranking and Sorting Examples ........ 27
3.5 Table Look-ups ........ 28
3.5.1 Language Comparison ........ 29
3.6 Gathers/Scatters on Serial Axes ........ 30
3.6.1 Conditions for Fast Performance ........ 30
3.6.2 Gather/Scatter Examples ........ 32
3.6.3 Language Comparison ........ 32

**Chapter 4 Parallel I/O ........ 33**

4.1 CM File Operations ........ 33
4.1.1 Opening, Closing, and Removing CM Files ........ 34
4.1.2 Reading and Writing CM Files ........ 35
4.1.3 Manipulating CM Files ........ 39
4.1.4 Example of CM File Operations ........ 42
4.2 Parallel I/O via Devices and Sockets ........ 43

**Appendix Dictionary of Utility Procedures ........ 47**

CMF_ALLOCATE_ARRAY ........ 50
CMF_ALLOCATE_DETAILED_ARRAY ........ 53
CMF_ALLOCATE_LAYOUT_ARRAY ........ 57
CMF_ALLOCATE_TABLE ........ 59
CMF_ARCHITECTURE ........ 61

CMF_AREF_1D ........ 62
CMF_ASET_1D ........ 64
CMF_AVAILABLE_MEMORY ........ 66
CMF_CM_ARRAY_FROM_FILE ........ 67
CMF_CM_ARRAY_FROM_FILE_FMS ........ 69

CMF_CM_ARRAY_FROM_FILE_SO ........ 71
CMF_CM_ARRAY_TO_FILE ........ 73
CMF_CM_ARRAY_TO_FILE_FMS ........ 75
CMF_CM_ARRAY_TO_FILE_SO ........ 77
CMF_DEALLOCATE_ARRAY ........ 79

CMF_DEALLOCATE_TABLE .......... 80
CMF_DEPOSIT_GRID_COORDINATE .......... 81
CMF_DESCRIBE_ARRAY .......... 83
CMF_FE_ARRAY_FROM_CM .......... 84
CMF_FE_ARRAY_TO_CM .......... 85

CMF_FILE_CLOSE .......... 86
CMF_FILE_FDOPEN .......... 87
CMF_FILE_GET_FD .......... 88
CMF_FILE_LSEEK .......... 89
CMF_FILE_LSEEK_FMS .......... 91

CMF_FILE_OPEN .......... 93
CMF_FILE_REWIND .......... 94
CMF_FILE_TRUNCATE .......... 95
CMF_FILE_UNLINK .......... 97
CMF_GET_GEOMETRY_ID .......... 98

CMF_LOOKUP_IN_TABLE .......... 99
CMF_MAKE_SEND_ADDRESS .......... 101
CMF_MY_SEND_ADDRESS .......... 103
CMF_NUMBER_OF_PROCESSORS .......... 105
CMF_ORDER .......... 106

CMF_RANDOM .......... 107
CMF_RANDOMIZE .......... 109
CMF_RANK .......... 110
CMF_SCAN .......... 114
CMF_SEND .......... 119
CMF_SIZEOF_ARRAY_ELEMENT .......... 122
CMF_SORT .......... 123

# Field Test Support

Field test software users are encouraged to communicate with Thinking Machines Corporation as fully as possible throughout the test period. Please report any errors you may find in this software and suggest ways to improve it.

When reporting an error, please provide as much information as possible to help us identify the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information is extremely helpful in this regard.

If your site has an applications engineer or a local site coordinator, please contact that person directly for field test support. Otherwise, please contact Thinking Machines' home office customer support staff:

| | |
|---|---|
| **Internet Electronic Mail:** | `customer-support@think.com` |
| **uucp Electronic Mail:** | `ames!think!customer-support` |
| **U.S. Mail:** | Thinking Machines Corporation<br>Customer Support<br>245 First Street<br>Cambridge, Massachusetts 02142-1264 |
| **Telephone:** | (617) 234-4000 |

# About This Manual

## Objectives of This Manual

This manual provides reference and usage information about the procedures in the CM Fortran Utility Library.

## Intended Audience

This manual assumes familiarity with CM Fortran programming.

## Organization of This Manual

The chapters of this manual describe the functional categories of utility procedures and suggest how to use them. The appendix is a dictionary of the individual procedures.

## Revision Information

This is a preliminary draft of a new manual. The Utility Library was previously documented in an appendix to the *CM Fortran User's Guide*, Version 1.1.

## Related Documents

- The *CM Fortran Reference Manual* defines the language; the *CM Fortran User's Guide* provides information about using the compiler.
- The dictionary entries in this manual are available on line as man pages. View them with the command `man` on CM-5 or `cmman` on CM-2/200, specifying the utility procedure name in uppercase.

# Chapter 1

# Introduction

The Utility Library provides convenient access from CM Fortran to the capabilities of lower-level CM software. The purpose is typically to achieve functionality or performance beyond what is currently available from the compiler.

As the compiler continues to develop, some of the utility procedures become redundant with CM Fortran language features. This manual compares utility procedures with the corresponding language features in the current release and notes any differences in behavior or performance.

## 1.1 Why a Utility Library?

CM Fortran programmers can use Utility Library procedures in situations where one is normally tempted to make explicit calls to lower-level software. There are several advantages to using the Utility Library instead in these situations:

- *Convenience*. The utility procedures take CM Fortran array names and other CM Fortran data objects as arguments. There is no need to convert CM Fortran objects into the data types used by lower-level software.

- *CM Portability*. With the few exceptions noted, the utility procedures support all CM hardware configurations and execution models, regardless of the particular lower-level software involved. There is no need to recode these calls to port a program from one CM system to another, even though the underlying system software may be quite different.

- *Support*. The Utility Library is a supported part of the CM Fortran product. Unlike some of the underlying system software, the library remains stable over time so that programs using it do not require maintenance.

## 1.2 Contents of the Utility Library

The chapters that follow describe the Utility Library procedures under these functional categories.

- Inquiries
  - System inquiry
  - Array inquiry
- Random number generation
- Dynamic array allocation
- Data motion (interprocessor)
  - Array transfers
  - Scatters with combining (plus array address construction)
  - Parallel prefix operations
  - Ranking and sorting
- Data motion (local)
  - Table look-ups
  - Gathers/scatters on serial axes
- Parallel I/O
  - CM file operations
  - CM I/O via devices or sockets

## 1.3 The Utility Library Header File

Each program unit that uses procedures from the Utility Library must include its header file:

```
INCLUDE '/usr/include/cm/CMF_defs.h'
```

The pathname of **`CMF_defs.h`** may be different if your system administrator has revised the CM directory structure.

The compiler command **`cmf`** links with the Utility Library automatically; no explicit linking is required.

## 1.4 Restrictions on Utility Procedures

- *Aligned arrays.* The utility procedures do not operate on arrays that are aligned with other arrays of higher rank or aligned with non-zero dimension offset(s) with any other array.
- *Lower bounds.* The utility procedures assume that all array dimensions have a lower bound of 1. Any other lower bound value is ignored. (An exception is the parallel I/O procedures, which accept arrays with any lower bound value.)

A few restrictions apply only to particular procedures. These are noted both in the text discussing the functional categories and in the individual procedure descriptions in the appendix.

Chapter 2

# Inquiries, Random Numbers, and Dynamic Allocation

## 2.1 System Inquiry Functions

Three functions report information about the CM system that is executing the program. They all take no arguments and return integer scalar results.

`CMF_ARCHITECTURE` returns a predefined constant that identifies the CM hardware platform and execution model:

```
ARCH = CMF_ARCHITECTURE( )
```

`CMF_NUMBER_OF_PROCESSORS` reports the number of processing elements available:

```
NUM = CMF_NUMBER_OF_PROCESSORS( )
```

The table below shows the return values of these two inquiry functions. Notice that the CM system component that serves as the "processing element" is different for the various platforms and execution models.

CM Fortran hardware platforms and execution models.

| | Compiler options | CMF_ARCHITECTURE returns | CMF_NUMBER_OF_ PROCESSORS returns |
|---|---|---|---|
| **CM-5** | | | |
| Vector units | `-cm5 -vu` | `CMF_CM5_VU` | number of vector units |
| Nodes | `-cm5 -sparc` | `CMF_CM5_SPARC` | number of nodes |
| **CM-200** | | | |
| Slicewise | `-cm200 -slicewise` | `CMF_CM200_SLICEWISE` | number of nodes |
| Paris | `-cm200 -paris` | `CMF_CM200_PARIS` | number of processors |
| **CM-2** | | | |
| Slicewise | `-cm2 -slicewise` | `CMF_CM2_SLICEWISE` | number of nodes |
| Paris | `-cm2 -paris` | `CMF_CM2_PARIS` | number of processors |
| **CM Fortran Simulator** | `-cmsim` | `CMF_CMSIM` | number of processors (1) |

See the *CM Fortran User's Guide* for more information on execution models and the hardware platforms they support.

A third inquiry function, `CMF_AVAILABLE_MEMORY`, reports the number of bytes of memory still available in each processing element:

```
MEM = CMF_AVAILABLE_MEMORY( )
```

NOTE: This function returns incorrect results for the vector unit model in Version 2.0 Beta.

### 2.1.1 Language Comparison

No comparable language feature.

## 2.2 Array Inquiry Subroutine

The subroutine `CMF_DESCRIBE_ARRAY` prints information about a CM array to standard output:

```
CALL CMF_DESCRIBE_ARRAY( ARRAY )
```

The output includes the home, rank, and dimension extents of the array, as well as detailed information about its layout on the processing elements.

The Utility Library also provides two special-purpose array inquiry functions.

- `CMF_GET_GEOMETRY_ID` is used only in constructing destination addresses for scatter operations; it is described in Section 3.2.1.
- `CMF_SIZEOF_ARRAY_ELEMENT` is used only for certain operations on CM files; it is described in Section 4.1.3.

### 2.2.1 Language Comparison

No comparable language feature.

## 2.3 Random Number Generation

Two subroutines serve to fill a CM array with pseudo random numbers:

```
CALL CMF_RANDOMIZE( SEED )

CALL CMF_RANDOM( DEST, LIMIT )
```

`CMF_RANDOMIZE` sets a seed for the random number generator used by `CMF_RANDOM`. `CMF_RANDOM` uses the initialized random number generator to store a pseudo random number in each element of the `DEST` array.

The `LIMIT` argument should always be specified as 1.0 for floating-point values. For integers, the argument serves as the exclusive upper bound of the values generated. If you do not want to set a limit for integer values, specify the `LIMIT` argument as 0.

The random number generator algorithm used by these routines is Wolfram's Rule 30 Cellular Automaton, described in Stephen Wolfram, "Random Sequence Generation by Cellular Automata," *Advances in Applied Mathematics* 7, pp. 123-69 (1986). This paper may be more readily available as a reprint in Stephen Wolfram, *Theory and Application of Cellular Automata*, World Scientific (1986).

The cellular automaton is run on a finite string of bits, i=0,...,N-1, with periodic boundary conditions (so that site N is equivalent to site 0). In the CM implementation N = 59.

- For integer data the random numbers are generated by simply running the automaton for 32 generations.
- For real, double-precision real, complex, or double-precision complex data, the random numbers are generated by running the automaton for *s* generations (where *s* is the mantissa length), and setting the exponent bits and sign bit so that the result is uniformly distributed between 2.0 and 1.0. Then 1.0 is subtracted from the result to yield a number that is uniformly distributed between 0.0 and 1.0.

### 2.3.1 Language Comparison

No comparable language feature.

## 2.4 Dynamic Array Allocation

Three subroutines allocate CM arrays at run time, giving the programmer different levels of control over the array's layout. A fourth subroutine deallocates an array created by any of the other three.

```
      CALL CMF_ALLOCATE_ARRAY
     &    ( FE_ARRAY, EXTENTS, RANK, TYPE )

      CALL CMF_ALLOCATE_LAYOUT_ARRAY
     &    ( FE_ARRAY, EXTENTS, RANK, TYPE, ORDERS, WEIGHTS )

      CALL CMF_ALLOCATE_DETAILED_ARRAY
     &    ( FE_ARRAY, EXTENTS, RANK, TYPE, ORDERS,
     &      SUBGRIDS, PMASKS )

      CALL CMF_DEALLOCATE_ARRAY( FE_ARRAY )
```

The `FE_ARRAY` argument is an integer front-end vector whose length is the predefined constant `CMF_SIZEOF_DESCRIPTOR`. This array is treated as the *descriptor* of a CM array; the remaining arguments specify information to be placed in the slots of the descriptor. All three variants take as arguments:

| | |
|---|---|
| ▪ `EXTENTS` | a front-end vector that contains dimension extents |
| ▪ `RANK` | a scalar integer that indicates rank |
| ▪ `TYPE` | A pre-defined integer constant that indicates type: `CMF_LOGICAL`, `CMF_S_INTEGER`, `CMF_FLOAT`, `CMF_DOUBLE`, `CMF_COMPLEX`, `CMF_DOUBLE_COMPLEX` |

The `FE_ARRAY` argument cannot be used as a CM array within the program unit that calls the allocation subroutine, since that program unit treats it as a front-end array. Instead, you pass the `FE_ARRAY` argument (that is, the descriptor) to another program unit that explicitly declares it a CM array. This method is illustrated in the following example.

### 2.4.1 Allocation Example (Canonical Layout)

```
      SUBROUTINE ALLOCATE()
      IMPLICIT NONE
      INTEGER NEW_ARRAY(CMF_SIZEOF_DESCRIPTOR)
      INTEGER EXTENTS(7), RANK, I
      PARAMETER (RANK=3)

      INCLUDE '/usr/include/cm/CMF_defs.h'

      DO I=1,RANK
         EXTENTS(I) = I * 10
      END DO

      CALL CMF_ALLOCATE_ARRAY
     &   (NEW_ARRAY, EXTENTS, RANK, CMF_S_INTEGER)

      CALL PRINT_DIMS3D(NEW_ARRAY)

      CALL CMF_DEALLOCATE_ARRAY(NEW_ARRAY)
      END SUBROUTINE ALLOCATE


      SUBROUTINE PRINT_DIMS3D(IN)
      IMPLICIT NONE
      INTEGER IN(:,:,:)

      PRINT *,"Shape of DUMMY is (",DUBOUND(IN,1),
     &        ",",DUBOUND(IN,2),
     &        ",",DUBOUND(IN,3),")"

      END SUBROUTINE PRINT_DIMS3D
```

### 2.4.2 Controlling Array Layout

The "layout" and "detailed" variants of the allocation procedures take additional front-end vector arguments that contain layout information for each of the array dimensions. The significance of these arguments is comparable to the various forms of the `cmf` compiler directive `LAYOUT`.

- `ORDERS` contains symbolic constants indicating the ordering of each dimension: `CMF_SERIAL_ORDER`, `CMF_NEWS_ORDER`, or (for CM-2/200 only) `CMF_SEND_ORDER`.
- `WEIGHTS` is a vector of integers indicating relative dimension weights.
- `SUBGRIDS` is a vector of integers indicating the desired subgrid length for each dimension (comparable to the `:BLOCK` item in the detailed-layout directive).
- `PMASKS` is a vector of integers that serve as bit-masks to indicate the desired processors (comparable to the `:PDESC` item in the detailed-layout directive). If `ORDERS` contains the value `CMF_SERIAL_ORDER` for any dimension, then `PMASKS` must contain `0` for that dimension.

There is no form directly comparable to the `:BLOCK :PROCS` form of the detailed `LAYOUT` directive. However, if `PMASKS` contains all zeros, the system computes the number of processors for each axis as extent / subgrid-length, rounded if necessary to the next power of 2.

### 2.4.3 Allocation Example (Detailed Layout)

```
IMPLICIT NONE
INCLUDE '/usr/include/cm/CMF_defs.h'
INTEGER NEWARRAY(CMF_SIZEOF_DESCRIPTOR)
INTEGER EXTENTS(7),ORDERS(7),SUBGRIDS(7),PMASKS(7)
INTEGER RANK,I
INTEGER NPN,NPN_FRAC,FRAC,SG1,SG2
REAL A(200)

PARAMETER (RANK = 2)
PARAMETER (FRAC = 4)
PARAMETER (SG1 = 5, SG2 = 40)
```

```
      A = 1.0  ! initialize if CM-2 running in auto-attach mode

      NPN = CMF_NUMBER_OF_PROCESSORS()
      NPN_FRAC = NPN/FRAC

      PMASKS(1)  = (NPN_FRAC - 1)  * FRAC
      PMASKS(2)  = FRAC - 1

      SUBGRIDS(1)  = SG1
      SUBGRIDS(2)  = SG2

      EXTENTS(1)  = NPN_FRAC * SG1
      EXTENTS(2)  = FRAC * SG2

      DO I = 1,RANK
        ORDERS(I) = CMF_NEWS_ORDER
      END DO

      CALL CMF_ALLOCATE_DETAILED_ARRAY
     &  (NEWARRAY,EXTENTS,RANK,CMF_FLOAT,ORDERS,SUBGRIDS,PMASKS)

      CALL USE_NEWARRAY(NEWARRAY,EXTENTS)

      CALL CMF_DEALLOCATE_ARRAY(NEWARRAY)
      END

      SUBROUTINE USE_NEWARRAY(A,EXT)
      INTEGER EXT(2)
      REAL A(EXT(1),EXT(2)), B(EXT(1),EXT(2))
CMF$  LAYOUT A(:,:)
CMF$  ALIGN  B(I,J) WITH A(I,J)

      B = CSHIFT(A,DIM=1,SHIFT=1)

C     Other operations on arrays A and B

      RETURN
      END
```

### 2.4.4 Restrictions

In addition to the general restrictions listed in Section 1.4, the following restrictions apply only to the dynamic allocation utilities.

- All four dynamic allocation utilities are incompatible with run-time safety, including argument checking and NaN checking. Do not use `-safety` or `-argument_checking` to compile a program that uses these procedures.
- The procedure `CMF_ALLOCATE_DETAILED_ARRAY` is not supported under the Paris execution model on CM-2 or CM-200.

### 2.4.5 Language Comparison

The dynamic allocation utility procedures are largely, but not completely, redundant with the CM Fortran statement `ALLOCATE`, which creates *deferred-shape* CM arrays. Some differences are:

- Deferred-shape arrays cannot appear in `COMMON`, so their names are not available to all program units. In contrast, arrays created with `CMF_ALLOCATE_ARRAY` or one of its variants can be globally available.
- Data types and ranks of deferred-shape arrays must be known at compile time. With `CMF_ALLOCATE_ARRAY`, they can be decided at run time (although used only in subroutines where the appropriate type and rank are declared).
- If a deferred-shape array is subject to a `LAYOUT` directive, the directive must appear in the specification part of the program unit (before any executable code). If you use the utility `CMF_ALLOCATE_LAYOUT_ARRAY` or `CMF_ALLOCATE_DETAILED_ARRAY` instead, you can compute before the call to determine layout-related values, such as subgrid lengths.
- The dynamic allocation utilities are incompatible with run-time safety, but deferred-shape arrays can be used in programs compiled with `-safety`.

Neither the Utility Library nor the CM Fortran language provides for dynamic allocation of front-end arrays or scalars. For this purpose, use the CM Fortran subroutines `FMALLOC` and `FFREE` in `libcmf77.a` (described in the *CM Fortran User's Guide*). These subroutines provide an interface to the standard `malloc` and `free` functionality that, together with the `%VAL` operator, enable you to manage front-end storage.

# Chapter 3

# Data Motion

This chapter describes the utility procedures that perform three distinct kinds of data movement:

- Array transfers between the control processor and the parallel unit
- Data communication among the parallel processing elements
  - Scatters with combining
  - Parallel prefix operations
  - Ranking and sorting
- Data motion on serial (locally stored) array dimensions
  - Table look-ups
  - Gathers/scatters on serial axes

## 3.1 Array Transfers

Two subroutines perform block transfers of array data between the serial control processor and the parallel processing unit:

```
CALL CMF_FE_ARRAY_TO_CM( DEST, SOURCE )

CALL CMF_FE_ARRAY_FROM_CM( DEST, SOURCE )
```

`CMF_FE_ARRAY_TO_CM` copies the contents of a front-end array `SOURCE` into a CM array `DEST`. `CMF_FE_ARRAY_FROM_CM` performs the opposite procedure. The source and destination arrays must match in shape and type.

### 3.1.1 Language Comparison

The `FORALL` statement can express CM-FE array transfers, such as:

```
FORALL (I=1:N) FE_ARRAY(I) = CM_ARRAY(I)
```

However, in Version 2.0 this statement generates a `DO` loop with calls to **`read-to-processor`** or **`write-from-processor`**; that is, it transfers array data between the system components one element at a time. For this release, the array-transfer utilities give better performance.

## 3.2 Scatters with Combining

The **CMF_SEND_** family of subroutines are used to scatter elements from a source array to specified locations in a destination array. If more than one value is sent to a single location, the values are combined according to the operation specified in the subroutine name:

```
   CALL CMF_SEND_combiner
  &    ( DEST, SEND_ADDRESS, SOURCE, MASK )
```

The combiners are **OVERWRITE**, **ADD**, **MAX**, **MIN**, **IOR**, **IAND**, and **IEOR**.

- **CMF_SEND_OVERWRITE** operates on CM arrays of any type. It arbitrarily chooses one of the colliding values to store in the destination location.
- **CMF_SEND_ADD** operates on any numeric type.
- **CMF_SEND_MAX** and **_MIN** operate on integer and real arrays (single- or double-precision).
- **CMF_SEND_IOR**, **_IAND**, and **_IEOR** operate on integer and logical arrays. They correspond to logical inclusive OR, logical AND, and logical exclusive OR, respectively. Integer operations are done on a bitwise basis.

The **MASK** argument controls which elements of **SOURCE** are selected for the operation. The **SEND_ADDRESS** argument is a CM array of destination addresses, constructed with the procedures described below. It must be conformable with the **SOURCE** array.

### 3.2.1 Constructing Send Address Arrays

A send address is an internal format for the linearized address of an *n*-dimensional coordinate. As such, it specifies an absolute location for a data element that is independent of its relative grid location.

Three procedures are used to convert grid coordinates (specifying the desired locations in the **DEST** array) into send addresses for use with **CMF_SEND_** :

```
   GEOMETRY = CMF_GET_GEOMETRY_ID( ARRAY )
   CALL CMF_MAKE_SEND_ADDRESS( ARRAY )
   CALL CMF_DEPOSIT_GRID_COORDINATE
  &    (GEOMETRY, SEND_ADDRESS, AXIS, COORDINATE, MASK)
```

(A related subroutine, **CMF_MY_SEND_ADDRESS (ARRAY)**, fills an array with the send addresses of its own elements.)

To construct a send-address array for use with **CMF_SEND_**, perform the following steps:

1. Declare an array to hold the send addresses. The array must have the same shape and layout as the **SOURCE** array with which it will be used.

   ```
   REAL*8 SEND_ADDRESS
   DIMENSION SEND_ADDRESS( ... )  ! same shape as source
   ```

   NOTE: The **SEND_ADDRESS** array may be declared as **INTEGER**, or as **DOUBLE PRECISION** or **REAL*8**. The CM-2/200 computes send addresses as 4-byte values; the CM-5 uses 8-byte send addresses. Both platforms accept either 4-byte or 8-byte send-address arrays. However, there may be a performance penalty for using 4-byte addresses on the CM-5, as the system coerces the values to 8-byte length. There is no performance penalty for using 8-byte send-address arrays on the CM-2, although there is some waste of memory. For maximum portability, CM Fortran programs should declare send-address arrays as **DOUBLE PRECISION** or **REAL*8**. **INTEGER** send-address arrays should only be used in programs to be run on the CM-2, and only when conserving memory is an issue.

2. Call **CMF_MAKE_SEND_ADDRESS** to initialize the send address array.

   ```
   CALL CMF_MAKE_SEND_ADDRESS( SEND_ADDRESS )
   ```

3. Use the function **CMF_GET_GEOMETRY_ID** to retrieve the geometry identifier of the **DEST** array:

   ```
   GEOMETRY = CMF_GET_GEOMETRY_ID( DEST )
   ```

   A geometry contains information about the shape and layout of a CM array, in this case, the array for which send addresses are being constructed.

4. Call **CMF_DEPOSIT_GRID_COORDINATE** on the coordinates for one axis.

   ```
    CALL CMF_DEPOSIT_GRID_COORDINATE
   &  (GEOMETRY, SEND_ADDRESS, AXIS, COORDINATE, MASK)
   ```

   The subroutine **CMF_DEPOSIT_GRID_COORDINATE** incorporates the grid coordinates for one axis into the send addresses being constructed. The **COORDINATE** array contains the grid coordinates for the axis of **GEOMETRY** specified by **AXIS**.

NOTE: The grid coordinates passed to **CMF_DEPOSIT_GRID_COORDINATE** should be 1-based. If you have specified a lower bound other than 1 for an array, you must adjust the coordinates specified in **COORDINATE** by subtracting 1 less than the lower bound.

5. Call **CMF_DEPOSIT_GRID_COORDINATE** again for each remaining axis of the **DEST** array, incorporating into the send address the **COORDINATE** values for that axis.

6. Pass the array of send addresses to the desired **CMF_SEND_** procedure.

### 3.2.2 Address-Construction and Scatter Example

The example below shows how to construct send addresses for a call to **CMF_SEND_ADD**.

```
      SUBROUTINE HISTOGRAM(OUT, IN, V1, V2)
      IMPLICIT NONE
      REAL, ARRAY(:,:) :: OUT, IN
      INTEGER, ARRAY(:,:) :: V1,V2
      REAL*8,  ARRAY(DUBOUND(IN,1),DUBOUND(IN,2)) :: SADDR
      INTEGER GEOM
CMF$  ALIGN SADDR(I,J) WITH IN(I,J)

      INCLUDE '/usr/include/cm/CMF_defs.h'
      ...
C   Get OUT array's geometry identifier
      GEOM = CMF_GET_GEOMETRY_ID(OUT)

C   Construct send addresses for OUT array
      CALL CMF_MAKE_SEND_ADDRESS(SADDR)
      CALL CMF_DEPOSIT_GRID_COORDINATE(GEOM,SADDR,1,V1,.TRUE.)
      CALL CMF_DEPOSIT_GRID_COORDINATE(GEOM,SADDR,2,V2,.TRUE.)

C   Perform send-with-add
      CALL CMF_SEND_ADD(OUT, SADDR, IN, .TRUE.)

      RETURN
      END
```

### 3.2.3 Language Comparison

Beginning with Version 2.0, the **FORALL** statement generates parallel send-with-combiner instructions for *n*-to-*m*-dimensional scatters when the possibility of data collisions exists. Except for arrays of high rank (as noted below), the performance of **FORALL** is comparable to that of **CMF_SEND**_*combiner*.

To express send-with-combiner operations with **FORALL**, supply an index array (conformable with the source array) for each dimension of the destination array. Then use a reduction function to combine multiple values being sent to the same destination element.

For example, a 1-to-1-dimensional send-with-add operation is written as:

```
FORALL(I=1:8) A(I)=SUM(B(1:1000),MASK=V(1:1000).EQ.I)
```

where
`A` is `A(8)` of numeric type.
`B` is `B(1000)` of numeric type.
`V` is `V(1000)` of type integer.

A 1-to-1-dimensional send-with-add operation that adds in the original destination value is written as:

```
FORALL(I=1:N) A(I) = A(I) + SUM(B(:), MASK=V(:).EQ.I)
```

For a 2-to-2-dimensional send-with-add, use an index array (conformable with the source array) for each dimension of the destination array:

```
 FORALL(I=1:N,J=1:M)
&      OUT(I,J) =
&      SUM(IN(:,:),
&          MASK=(X(:,:).EQ.I).AND.(Y(:,:).EQ.J))
```

A 1-to-2-dimensional send-with-add operation is written as:

```
 FORALL(I=1:N,J=1:M)
&      OUT(I,J)=
&      SUM(IN(:), MASK=X(:).EQ.I .AND. Y(:).EQ.J)
```

A permanent restriction on this use of **FORALL** is that it generates parallel instructions *only if* the rank of **OUT** *plus* the rank of **IN** is below a certain threshold. The threshold in Version 2.0 is 7. For arrays of higher rank, use the utility procedure **CMF_SEND**_*combiner* for best performance.

## 3.3 Parallel Prefix Operations

The subroutines in this section perform parallel prefix operations, or *scans*, on one axis of an array:

```
    CALL CMF_SCAN_combiner ( DEST, SOURCE, SEGMENT, AXIS
   &        DIRECTION, INCLUSION, SEGMENT_MODE, MASK )
```

These subroutines apply a binary operator cumulatively over the elements of the **SOURCE** array **AXIS**, combining each value with the cumulative result from all the values that precede it. The result for each element is stored in the corresponding element of the **DEST** array.

The combiners are **COPY**, **ADD**, **MAX**, **MIN**, **IOR**, **IAND**, and **IEOR**.

- **CMF_SEND_COPY** operates on CM arrays of any type. It copies the first element of an axis to all the other elements of that axis.
- **CMF_SEND_ADD** operates on any numeric type.
- **CMF_SEND_MAX** and **_MIN** operate on integer and real arrays (single- or double-precision).
- **CMF_SEND_IOR**, **_IAND**, and **_IEOR** operate on integer and logical arrays. They correspond to logical inclusive OR, logical AND, and logical exclusive OR, respectively. Integers operations are done on a bitwise basis.

**DIRECTION** can be **CMF_UPWARD** or **CMF_DOWNWARD**. If the value is **CMF_UPWARD**, the values are combined from the lower numbered elements toward the higher. If the value is **CMF_DOWNWARD**, the values are combined from higher numbered elements toward the lower.

The scan can be limited to selected elements of the array axis through the **MASK** argument, a logical CM array conformable with **SOURCE** and **DEST**. Selected elements are those that correspond to a **.TRUE.** element in the **MASK** array. Array elements that correspond to a **.FALSE.** value in **MASK** are excluded from the computation, and the corresponding element of **DEST** is not changed.

In addition, the array elements along the axis may be partitioned into distinct sets, called *segments*, through the use of the **SEGMENT**, **SEGMENT_MODE**, and **INCLUSION** arguments. Each segment is treated as a separate set of values. **SEGMENT** is a logical CM array conformable with **SOURCE** and **DEST**; **SEGMENT_MODE** and **INCLUSION** are predefined integer constants.

### 3.3.1 Scan Segments

Each element of **SEGMENT** that contains **.TRUE.** marks the corresponding element of **SOURCE** as a segment boundary (the start or end of a segment). Segments begin (or end) with an element in which the value of **SEGMENT** is **.TRUE.**, and continue up (or down) the axis through all elements for which the value of **SEGMENT** is **.FALSE.**. The effect of these boundaries depends on the value of **SEGMENT_MODE**.

- If **SEGMENT_MODE** is **CMF_NONE**, the scan operation proceeds along the entire length of the array axis and the values in **SEGMENT** have no effect.
- If **SEGMENT_MODE** is **CMF_SEGMENT_BIT**, then:
  - The **MASK** argument does not affect the use of the **SEGMENT** array. That is, elements containing **.TRUE.** in the **SEGMENT** array create a segment boundary even if the corresponding value of **MASK** is **.FALSE.**. (The **MASK** array still selects the elements of **SOURCE** to be included, as described above.)
  - A **SEGMENT** value of **.TRUE.** indicates the start of a segment for both upward and downward scans.
  - When the **INCLUSION** argument is **CMF_EXCLUSIVE**, the first **DEST** element in each segment is set to zero. (There is no scan result value for this element because in exclusive mode the first element of each segment of **SOURCE** is excluded from the scan.)
- If the value is **CMF_START_BIT**, then:
  - The **MASK** argument applies to the **SEGMENT** array as well as to the **SOURCE** array. That is, elements containing **.TRUE.** in **SEGMENT** array create a segment boundary only if the corresponding element of **MASK** is also **.TRUE.**.
  - A **SEGMENT** value of **.TRUE.** indicates the *start* of a segment for upward scans, but the *end* of a segment for downward scans. That is, the **SOURCE** element corresponding to a **.TRUE. SEGMENT** element is the first element in a segment for an upward scan, but the last element in a segment for a downward scan. In downward scans, the new segment begins with the first unmasked element following the segment boundary.

- When the **INCLUSION** argument is **CMF_EXCLUSIVE**, the first **DEST** element in each segment (which is set to zero in **CMF_SEGMENT_BIT** scans) is used to store the final scan result of the preceding segment. Note that this result value does not contribute to the scan result for the segment in which it is stored.

## 3.3.2 Scanning Example

The table below shows the results for a single row along an axis being scanned by the subroutine **CMF_SCAN_ADD**. The **SOURCE** argument is an integer array filled with the value 1. The **MASK** and **SEGMENT** arguments are logical arrays with the values indicated at the top of the table (where **T** stands for **.TRUE.** and **F** stands for **.FALSE.**).

The table shows scan results for both directions, both inclusion modes, and all three segmentation modes. The dots indicate masked elements; the underlining groups elements that are considered part of the same segment.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **MASK** | T | T | T | T | F | F | F | F | T | T | F | F | T | T | T | F |
| **SEGMENT** | F | F | T | F | F | F | T | F | F | F | F | F | F | T | F | F |
| **SOURCE** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| DIRECTION | INCLUSION | SEGMENT-MODE | DEST |
|---|---|---|---|
| upward | exclusive | none | `0 1 2 3 . . . . 4 5 . . 6 7 8 .` |
| downward | exclusive | none | `8 7 6 5 . . . . 4 3 . . 2 1 0 .` |
| upward | inclusive | none | `1 2 3 4 . . . . 5 6 . . 7 8 9 .` |
| downward | inclusive | none | `9 8 7 6 . . . . 5 4 . . 3 2 1 .` |
| upward | exclusive | segment | `0 1 0 1 . . . . 0 1 . . 2 0 1 .` |
| downward | exclusive | segment | `1 0 1 0 . . . . 2 1 . . 0 1 0 .` |
| upward | inclusive | segment | `1 2 1 2 . . . . 1 2 . . 3 1 2 .` |
| downward | inclusive | segment | `2 1 2 1 . . . . 3 2 . . 1 2 1 .` |
| upward | exclusive | start | `0 1 2 1 . . . . 2 3 . . 4 5 1 .` |
| downward | exclusive | start | `2 1 5 4 . . . . 3 2 . . 1 1 0 .` |
| upward | inclusive | start | `1 2 1 2 . . . . 3 4 . . 5 1 2 .` |
| downward | inclusive | start | `3 2 1 5 . . . . 4 3 . . 2 1 1 .` |

### 3.3.3 Language Comparison

A scan operation is expressed with **FORALL** as:

```
FORALL (I=1:N) A(I) = SUM( B(1:I) )
```

In Version 2.0, this statement generates a **sum** of **spread** rather than a **scan** instruction. The utility procedure **CMF_SCAN_***combiner* gives better performance.

## 3.4 Ranking and Sorting

Two subroutines determine the numerical rank of the values along a dimension of a CM array; a third sorts the values by rank.

`CMF_ORDER` places the numerical rank of each element along the specified axis of a source array into the corresponding element of the destination array, under the control of a logical mask. The source, destination, and mask arguments must be conformable arrays.

```
      CALL CMF_ORDER (DEST, SOURCE, AXIS, MASK)
```

`CMF_RANK` performs the same operation, but it also enables you to break the axis into segments. The direction argument (either `CMF_UPWARD` or `CMF_DOWNWARD`) determines whether the smallest or the largest value is given rank 1.

```
      CALL CMF_RANK (DEST, SOURCE, SEGMENT, AXIS,
     &       DIRECTION, SEGMENT_MODE, MASK)
```

`CMF_SORT` places the sorted values themselves in the destination array. It, too, enables you to control the direction of the sort and to segment the source axis.

```
      CALL CMF_SORT (DEST, SOURCE, SEGMENT, AXIS,
     &       DIRECTION, SEGMENT_MODE, MASK)
```

Language comparison: CM Fortran has no ranking or sorting functions.

### 3.4.1 Axis Segments

`CMF_RANK` and `CMF_SORT` take `SEGMENT` and `SEGMENT_MODE` arguments that partition the source array axis into distinct segments. Each segment is treated as a separate set of values for ranking purposes.

- The `SEGMENT` argument is logical array that is conformable with `SOURCE` and `DEST`. Each element of `SEGMENT` that contains `.TRUE.` marks the corresponding element of `SOURCE` as a segment boundary. The effect of these boundaries depends on the value of `SEGMENT_MODE`.

The **SEGMENT_MODE** argument is a pre-defined integer constant, one of **CMF_NONE**, **CMF_SEGMENT_BIT**, or **CMF_START_BIT**.

- If **SEGMENT_MODE** is **CMF_NONE**, the elements are sorted along the entire length of the array axis and the values in **SEGMENT** have no effect.
- If **SEGMENT_MODE** is **CMF_SEGMENT_BIT**, then:
  - A **SEGMENT** value of **.TRUE.** indicates the start of a segment for both upward and downward sorts.
  - The **MASK** argument does not affect the use of the **SEGMENT** array. That is, elements containing **.TRUE.** in the **SEGMENT** array create a segment boundary even if the corresponding value of **MASK** is **.FALSE.**. (The **MASK** array still selects the elements of **SOURCE** to be included.)
- If **SEGMENT_MODE** is **CMF_START_BIT**, then:
  - A **SEGMENT** value of **.TRUE.** indicates the *start* of a segment for upward sorts, but the *end* of a segment for downward sorts. That is, the **SOURCE** element corresponding to a **.TRUE.** **SEGMENT** element is the first element in a segment for an upward sort, but the last element in a segment for a downward sort. In downward sorts, the new segment begins with the first unmasked element following the segment boundary.
  - The **MASK** argument applies to the **SEGMENT** array as well as to the **SOURCE** array. That is, elements containing **.TRUE.** in the **SEGMENT** array create a segment boundary only if the corresponding element of **MASK** is also **.TRUE.**.

Specific behavior of **CMF_RANK** and **CMF_SORT** on segmented axes is illustrated in the examples shown below. Note that the segmentation is not carried over into the destination array:

- **CMF_RANK** ranks each element within its own segment, but the numbering of the elements is continuous along the entire length of the axis. In the final example below, **DEST** is `[XXX 1  3  2 ]`, not `[XXX 1  2  1 ]`.
- **CMF_SORT** sorts each segment independently, but the values are placed in the destination without regard to segments. In the final example below, **DEST** is `[7.0,  2.0,  3.0,  XXX]`, not `[7.0,  XXX,  2.0,  3.0]`).

## 3.4.2 Ranking and Sorting Examples

Upward sort and rank:

```
If SOURCE =         [1.0 7.0 3.0 2.0]
and SEGMENT =       [T   F   F   F  ]

then rank DEST =    [1   4   3   2]
and sort DEST =     [1.0 2.0 3.0 7.0]
```

Downward sort and rank:

```
If SOURCE =         [1.0 7.0 3.0 2.0]
and SEGMENT =       [T   F   F   F  ]

then rank DEST =    [4   1   2   3]
and sort DEST =     [7.0 3.0 2.0 1.0]
```

Upward sort and rank with mask:

```
If SOURCE =         [1.0 7.0 3.0 2.0]
and SEGMENT =       [T   F   F   F  ]
and MASK =          [T   T   F   T  ]

then rank DEST =    [1   3   XXX   2]
and sort DEST =     [1.0 2.0 7.0 XXX]
```

Segmented upward sort and rank:

```
If SOURCE =         [1.0 7.0 3.0 2.0]
and SEGMENT =       [T   F   T   F  ]

then rank DEST =    [1   2   4   3  ]
and sort DEST =     [1.0 7.0 2.0 3.0]
```

Segmented upward sort and rank with mask:

```
If SOURCE =         [1.0 7.0 3.0 2.0]
and SEGMENT =       [T   F   T   F  ]
and MASK =          [F   T   T   T  ]

then rank DEST =    [XXX 1   3   2  ]
and sort DEST =     [7.0 2.0 3.0 XXX]
```

## 3.5 Table Look-Ups

Three procedures are used to perform "table look-ups," that is, vector indirection on a serial dimension of a CM array. Under the conditions noted below, the look-up utility uses the indirect addressing hardware on the CM processing elements to perform local memory accesses, rather than generating communication.

```
      TABLE_ID = CMF_ALLOCATE_TABLE
     &    ( TYPE, ELEMENT_COUNT, INITIAL_VALUES )

      CALL CMF_LOOKUP_IN_TABLE
     &    ( DEST, TABLE_ID, INDEX, MASK )

      CALL CMF_DEALLOCATE_TABLE(TABLE_ID)
```

The function **CMF_ALLOCATE_TABLE** allocates and initializes a look-up table, placing a copy in the memory of each processing element; it returns an integer that serves as a pointer to the table. **TYPE** is the type of data to be stored in the table; it is specified as one of:

**CMF_LOGICAL**, **CMF_S_INTEGER**,
**CMF_FLOAT**, **CMF_DOUBLE**,
**CMF_COMPLEX**, **CMF_DOUBLE_COMPLEX**

The elements of **INITIAL_VALUES** must be of the appropriate type.

**CMF_LOOKUP_IN_TABLE** uses an array of (integer) indices to retrieve values from the look-up table, and stores them in a destination array of the same type. **CMF_DEALLOCATE_TABLE** deallocates a look-up table. For example,

```
      REAL DEST(8192), TABLE_VALUES(100)
      INTEGER TABLE
      INTEGER INDEX(8192)
      ...

      TABLE = CMF_ALLOCATE_TABLE
     &    (CMF_FLOAT, 100, TABLE_VALUES)

      CALL CMF_LOOKUP_IN_TABLE
     &    (DEST, TABLE, INDEX, .TRUE.)
      ...

      CALL CMF_DEALLOCATE_TABLE(TABLE)
      ...
```

### 3.5.1 Language Comparison

Under certain circumstances, the table look-up procedures are significantly faster than assignments of conventionally allocated arrays. The circumstances are:

- The contents of the look-up table rarely or never change.
- The look-up table is relatively small, that is, it fits into the memory of a single processing element. The size restriction by CM Fortran execution model is:

| | |
|---|---|
| CM-5 VU model: | Table size is limited by the amount of memory on a vector unit. |
| CM-5 nodes model : | Table size is limited by the amount of memory on a SPARC node. |
| CM-2/200 slicewise model: | Table size is limited by the amount of memory on a processing node (which corresponds to a unit of the 64-bit floating-point accelerator). |
| CM-2/200 Paris model: | Table size is limited by the amount of memory on a processing node (which corresponds to 32 bit-serial processors). |

## 3.6 Gathers/Scatters on Serial Axes

Two subroutines transfer array-indexed values between two CM arrays. Under the conditions noted below, these procedures use the special indirect addressing hardware for local transfers.

```
CALL CMF_AREF_1D(DEST, ARRAY, INDEX, MASK)

CALL CMF_ASET_1D(ARRAY, SOURCE, INDEX, MASK)
```

The **ARRAY** argument can be multidimensional. The "**1D**" in the procedure names refers to the fact that the indirect addressing occurs only on a single axis.

**CMF_AREF_1D** extracts array-indexed values from the serial axis of **ARRAY**. **INDEX** is an **INTEGER** array of the same shape and layout as **DEST**. Each element of **INDEX** provides an index into **ARRAY** for the value to be stored in the corresponding element of **DEST**.

**CMF_ASET_1D** performs the opposite operation. **INDEX** is an **INTEGER** array of the same shape and layout as **SOURCE**. In this operation, each element of **INDEX** specifies the location in **ARRAY** at which to store the corresponding element of **SOURCE**.

### 3.6.1 Conditions for Fast Performance

These subroutines use the fast indirect addressing hardware when the **ARRAY** argument meets the following conditions:

- Its first dimension must be serially ordered (that is, local to a processing element).
- It must have one more dimension than the **INDEX**, **MASK**, and **DEST** arrays.
- Excluding its first axis, its remaining axes must have the same shape and layout as the **INDEX**, **MASK**, and **DEST** arrays.

In addition, these subroutines are substantially faster when

- the **MASK** argument is the scalar **.TRUE.**.
- the product of the dimensions of the **INDEX** argument is an integer multiple of the number of processing elements executing the program. (This number is returned by the function **CMF_NUMBER_OF_PROCESSORS**.)

Two restrictions that affect the performance of these subroutines are:

- The subroutines do not use the indirect addressing hardware under the Paris execution model on CM-2/200, even if the other constraints are met. Their performance under the Paris model is therefore slower than under the other CM Fortran execution models.
- The serial dimension of **ARRAY** must fit into the memory of a single processing element. The size restriction by CM Fortran execution model is:

| | |
|---|---|
| ▪ CM-5 VU model: | Serial dimension extent is limited by the amount of memory on a vector unit. |
| ▪ CM-5 nodes model: | Serial dimension extent is limited by the amount of memory on a SPARC node. |
| ▪ CM-2/200 slicewise model: | Serial dimension extent is limited by the amount of memory on a processing node (which corresponds to a unit of the 64-bit floating-point accelerator). |
| ▪ CM-2/200 Paris model: | Not applicable. |

### 3.6.2 Gather/Scatter Examples

This call to **CMF_AREF_1D** is functionally equivalent to the **DO** loop shown:

```
      INTEGER I
      INTEGER DEST(8192), ARRAY(10,8192), INDEX(8192)
CMF$  LAYOUT ARRAY(:SERIAL, :NEWS)
      LOGICAL MASK(8192)
      ...
      DO I=1,8192
         IF (MASK(I)) DEST(I) = ARRAY(INDEX(I),I)
      END DO
      ...
      CALL CMF_AREF_1D(DEST, ARRAY, INDEX, MASK)
      ...
```

This call to **CMF_ASET_1D** is functionally equivalent to the **DO** loop shown:

```
      INTEGER I
      INTEGER SOURCE(8192), ARRAY(10,8192), INDEX(8192)
CMF$  LAYOUT ARRAY(:SERIAL, :NEWS)
      LOGICAL MASK(8192)
      ...
      DO I=1,8192
         IF (MASK(I)) ARRAY(INDEX(I),I) = SOURCE(I)
      END DO
      ...
      CALL CMF_ASET_1D(ARRAY, SOURCE, INDEX, MASK)
      ...
```

### 3.6.3 Language Comparison

The **FORALL** statement expresses the operations shown in the examples above as follows:

```
FORALL(I=1:8192, MASK(I)) DEST(I) = ARRAY(INDEX(I),I)

FORALL(I=1:8192, MASK(I)) ARRAY(INDEX(I),I) = SOURCE(I)
```

In Version 2.0, however, these statements generate **send** (scatter) or **get** (gather) instructions rather than using the local indirect addressing hardware. As long as the stated constraints are met, the utility procedures **CMF_AREF_1D** and **CMF_ASET_1D** give better performance.

# Chapter 4

# Parallel I/O

The Utility Library procedures in this chapter support CM parallel I/O. Parallel I/O refers to transferring data in multiple streams between the CM processing elements and an external device. The procedures fall into two categories:

- Operations on files of the CM file system
- I/O via sockets and devices (including CM-HIPPI)

#### Language Comparison

The CM Fortran `READ` and `WRITE` statements perform serial I/O only. A CM array is first moved to the control processor and then transferred in a single stream to a UNIX file on a storage device. For CM arrays, especially for large ones, the Utility Library I/O procedures give better performance.

## 4.1 CM File Operations

The CM file system — the destination of parallel write operations — resides on storage devices on the CMIO bus, such as the DataVault mass storage system. Operations on these files are supported by the CM File System library, CMFS. The utility procedures in this section provide a convenient interface to selected procedures in this library.

For more information on the CM file system and library, see the CM I/O documentation for CM-5 or for CM-2/200. Note that support for the CM Scalable Disk Array and its Scalable File System (SFS) begins with CM Fortran Version 2.1.

### 4.1.1 Opening, Closing, and Removing CM Files

The subroutines in this section open, close, or remove (unlink) CM files.

#### Opening a CM File

`CMF_FILE_OPEN` opens the CM file specified by `PATH` (a character string) and associates the file with the integer argument `UNIT`. The value returned in `IOSTAT` indicates whether the operation succeeded.

```
CALL CMF_FILE_OPEN( UNIT, PATH, IOSTAT )
```

#### File Units

The I/O procedures currently support 29 simultaneously open file units. For each CM file to be opened, you choose a value in the range 1 through 29. The number becomes associated with a file when it is used as the `UNIT` argument (variable, parameter, or literal constant) to `CMF_FILE_OPEN`. You then supply the appropriate unit number to other I/O procedures when you wish to operate on this file.

These parallel I/O unit numbers have no relation to standard CM Fortran unit numbers, as described in the *CM Fortran Reference Manual* for the `READ` and `WRITE` statements.

#### Error Status

All the parallel I/O procedures take an integer `IOSTAT` argument, into which the error status of the operation is placed:

- A positive value in `IOSTAT` indicates success.
- A negative value in `IOSTAT` indicates failure.
- For the parallel read utilities only (see below), a zero value in `IOSTAT` indicates an end-of-file condition.

Other than sign or zero, there is no significance to any of the particular values returned.

### Closing a CM File

**CMF_FILE_CLOSE** closes the file associated with **UNIT**.

```
CALL CMF_FILE_CLOSE( UNIT, IOSTAT )
```

### Removing a CM File

**CMF_FILE_UNLINK** removes the entry for the file specified by **PATH** from the file's directory.

```
CALL CMF_FILE_UNLINK( PATH, IOSTAT )
```

If this entry is the last link to the file and no process has the file open, then the file is deleted and all resources associated with it are reclaimed. If, however, the file is open in any process, the resource reclamation is delayed until the file is closed, even though the directory entry has disappeared.

## 4.1.2 Reading and Writing CM Files

The CM Fortran Utility Library provides procedures that read or write CM arrays in parallel, that is, in multiple streams directly between the memory of CM processors and a CM file on a storage device.

These procedures are available in three variants, reflecting different trade-offs between speed and flexibility. The variants are distinguished by suffix (or lack of): no-suffix or generic, FMS, or SO. They take the same arguments.

```
CALL CMF_CM_ARRAY_TO_FILE      (UNIT, ARRAY, IOSTAT)
CALL CMF_CM_ARRAY_FROM_FILE    (UNIT, ARRAY, IOSTAT)

CALL CMF_CM_ARRAY_TO_FILE_FMS      ...
CALL CMF_CM_ARRAY_FROM_FILE_FMS    ...

CALL CMF_CM_ARRAY_TO_FILE_SO       ...
CALL CMF_CM_ARRAY_FROM_FILE_SO     ...
```

Always read a file with the same variant that was used to write it.

## Arguments

| | |
|---|---|
| `UNIT` | Integer variable, parameter, or literal constant in the range 1:29. This is the unit number that became associated with a file by the initial call to `CMF_FILE_OPEN` (see Section 4.1.1). |
| `ARRAY` | CM array of any type and layout. This array is the source or destination of the I/O operation. |
| `IOSTAT` | Integer variable. The value returned in this argument indicates the error status of the operation: |

- A positive value in `IOSTAT` indicates success.
- A negative value in `IOSTAT` indicates failure.
- For the parallel read utilities only, a zero value in `IOSTAT` indicates an end-of-file condition.

**NOTE**

Like all procedures in the Utility Library, these I/O procedures cannot be used with any array that is aligned with another array of higher rank or aligned with non-zero axis offset(s) with any other array.

Unlike other Utility Library procedures, these I/O procedures do support arrays with lower bounds other than 1.

## Behavior

The three sets of read/write procedures give different combinations of speed and portability. The FMS ("fixed machine size") routines are the fastest but the least flexible. The SO ("serial order") routines are slower but the most portable across CM configurations and execution models. The generic (no-suffix) routines are a compromise between the two for general-purpose use.

The generic and FMS procedures treat file data in a *parallel* order, which we call the *geometry* of the file. File geometry reflects the shape and layout of the first array written to that file. In consequence:

- All subsequent writes to a parallel-ordered file must be of arrays of the same shape and layout as the first, and any read operation from the file must be to an array of the same shape and layout.

- Parallel-ordered files may contain extraneous data (padding) in scattered locations. As long as you observe the restrictions on using the FMS and generic routines (summarized in the table below), the padding is handled transparently when the file is read.

In the interest of speed, the FMS procedures impose the further restriction that write and read operations of a CM file must be from the same execution model, and the same machine size (number of processing elements). Hence the term "fixed machine size." These procedures are not portable from one CM model (CM-2/200 versus CM-5) to the other.

The generic procedures, in contrast, are limited only by array shape, layout, and CM model. They can write and then read from different execution models and machine sizes, although the following restrictions do apply on CM-2/200 only:

- An array written under one CM-2/200 execution model (Paris or slicewise) and read under the other execution model must have canonical layout. You can work around this restriction by assigning a non-canonical array to a canonical temporary before writing it to a file.

- An array written from one CM-2/200 machine size and read into a different machine size must be at least the size of the larger machine. That is, the array must have at least as many elements as the number of bit-serial processors in the larger machine (even under the slicewise execution model).

The SO procedures treat file data in *serial* order. Serial ordering is the same as array-element ordering and the same as the output of the Fortran `WRITE` statement. For example, the SO utility stores the elements of array `A(2,3)` in the following order:

```
A(1,1)
A(2,1)
A(1,2)
A(2,2)
A(1,3)
A(2,3)
```

Unlike the generic and FMS variants, the SO procedures do not pad files. Because they read and write only the array elements, not any extraneous data, these utilities operate independently of array shape and layout, and are completely portable across CM hardware models, execution models, and machine sizes. The SO utilities are also compatible with parallel I/O via sockets and devices (Section 4.2).

The following table summarizes the behavior of the three variants of the parallel read/write utility procedures. The "portability" entry refers to restrictions on subsequent reads and writes of a CM file after the first array has been written to it.

Variants of `CMF_CM_ARRAY_TO/FROM_FILE`.

| | **FMS** | **Generic** | **SO** |
|---|---|---|---|
| **CM-5** | | | |
| **File order** | parallel | parallel | serial |
| **Padding** | yes | yes | no |
| **Portability** | CM-5 only | CM-5 only | any CM or device |
| | same partition size | any partition size | any partition or machine size |
| | same exec. model | any exec. model | any exec. model |
| | same array shape | same array shape | any array shape |
| | same array layout | same array layout | any array layout |
| **CM-2/200** | | | |
| **File order** | parallel | parallel | serial |
| **Padding, if any** | yes | yes | no |
| **Portability** | CM-2/200 only | CM-2/200 only | any CM or device |
| | same machine size | any machine size* | any machine or partition size |
| | same exec. model | any exec. model | any exec. model |
| | same array shape | same array shape | any array shape |
| | same array layout | same array layout** | any array layout |

* If written from one machine size and read into a different machine size, the array must be at least the size of the larger machine.

** If written from one execution model and read into the other execution model, the array must have canonical layout.

### 4.1.3 Manipulating CM Files

The procedures in this section rewind, seek within, or truncate a CM file.

#### Determining File Geometry

All seek, rewind, and truncate operations on CM files must be preceded by a read or write operation. It is necessary first to determine the geometry of a newly opened file, even a serial-order file, by performing a read or write of the file.

For the CM-5 only, a further restriction is that the element size in any file manipulation (rewind, seek, or truncate) must be the same as the element size in the read or write operation that initially determined the file's geometry in that session.

#### Rewinding a File

`CMF_FILE_REWIND` moves the file pointer to the beginning of a CM file.

```
CALL CMF_FILE_REWIND( UNIT, IOSTAT )
```

#### Seeking within a File

Three procedures serve to reposition the file pointer in a CM file:

```
CALL CMF_FILE_LSEEK( UNIT, OFFSET, IOSTAT )

CALL CMF_FILE_LSEEK_FMS( UNIT, OFFSET, IOSTAT )

OFFSET = CMF_SIZEOF_ARRAY_ELEMENT( ARRAY )
```

`CMF_FILE_LSEEK` operates on files written with the generic and SO write procedures; use `CMF_FILE_LSEEK_FMS` on files written with the FMS write procedure.

The seek utilities operate slightly differently on serial-ordered files (those written with the SO procedure), compared with parallel-ordered files (those written with the generic or FMS procedure).

- In serial-ordered files, **CMF_FILE LSEEK** can move the file pointer either to an array boundary or to an arbitrary element (though not to an arbitrary bit or byte boundary). For the **OFFSET** argument, use the number of bytes in the array's element type times the number of elements to traverse.
- In parallel-ordered files, **CMF_FILE_LSEEK** and **CMF_FILE_LSEEK_FMS** move the file pointer *only* from one array to another within a file. You cannot move it to an arbitrary element. To compute the offset, you need not specify the size of the array(s), since this information is contained in the file geometry. You need specify only the size of an array's elements, using **CMF_SIZEOF_ARRAY_ELEMENT**.

For example, suppose that a parallel-ordered file associated with unit 29 was created with three successive writes, of array **A**, then array **B**, then array **C**. To position the file pointer to the beginning of array **B**, use:

```
CALL CMF_FILE_REWIND( 29, IOSTAT )
SIZEOF_A = CMF_SIZEOF_ARRAY_ELEMENT( A )
CALL CMF_FILE_LSEEK( 29, SIZEOF_A, IOSTAT )
```

To move the file pointer to the beginning of array **C**, add the return values of **CMF_SIZEOF_ARRAY_ELEMENT** for the two arrays to be traversed:

```
CMF_FILE_REWIND( 29, IOSTAT )
SIZEOF_A = CMF_SIZEOF_ARRAY_ELEMENT( A )
SIZEOF_B = CMF_SIZEOF_ARRAY_ELEMENT( B )
CMF_FILE_LSEEK( 29, SIZEOF_A + SIZEOF_B, IOSTAT )
```

To read arrays **A** and **C**:

```
CALL CMF_FILE_REWIND( 29, IOSTAT )
CALL CMF_CM_ARRAY_FROM_FILE( 29, DEST_A, IOSTAT )
CALL CMF_FILE_LSEEK( 29, SIZEOF_B, IOSTAT )
CALL CMF_CM_ARRAY_FROM_FILE( 29, DEST_C, IOSTAT )
```

If these arrays had been written with **CMF_CM_ARRAY_TO_FILE_FMS**, you would have used the FMS variant of the seek and read procedures.

## Changing the Size of a File

CMF_FILE_TRUNCATE increases or decreases the size of a CM file:

```
CALL CMF_FILE_TRUNCATE( UNIT, LENGTH, IOSTAT )

LENGTH = CMF_SIZEOF_ARRAY_ELEMENT( ARRAY )
```

This subroutine changes the size of the file specified by UNIT to LENGTH. If the file is smaller than LENGTH, it is extended to LENGTH. If the file is larger than LENGTH, it is truncated and the extra data is lost. The file must be open when CMF_FILE_TRUNCATE is called.

Like the seek procedures described above, CMF_FILE_TRUNCATE behaves slightly differently with serial-ordered and parallel-ordered files, and you compute the LENGTH argument differently for the two kinds of files.

- CMF_FILE_TRUNCATE can extend or truncate a serial-ordered file either by whole arrays or by an arbitrary number of array elements (though not by an arbitrary number of bits or bytes). For the LENGTH argument, supply the number of bytes in the array's element type times the number of elements desired.
- CMF_FILE_TRUNCATE can extend or truncate a parallel-ordered file only by whole arrays, not by an arbitrary number of array elements. To compute the LENGTH argument, you need not specify the size of the array(s), since this information is contained in the file geometry. Specify only the size of array elements, using CMF_SIZEOF_ARRAY_ELEMENT.

You compute the LENGTH argument for parallel-ordered files in the same way as the OFFSET argument for the seek procedures, shown above. And, as with the seek procedures, you can extend or truncate a file by more than one array by invoking CMF_SIZEOF_ARRAY_ELEMENT on several arrays in succession, adding the returned values, and supplying the result as the LENGTH argument.

### 4.1.4 Example of CM File Operations

The following program writes five arrays into a file and then reads the third one:

```
      PROGRAM READ_RECORD
      INTEGER FILE_UNIT, IOSTAT, RECORD
      REAL A(8192)
      DOUBLE PRECISION B(8192)
      COMPLEX C(8192), DEST(8192)
      DOUBLE COMPLEX D(8192)
      LOGICAL E(8192)
      INCLUDE '/usr/include/cm/CMF_defs.h'

C Initialize variables

      FILE_UNIT = 13
      IOSTAT    = 0
      A = [1:8192]
      B = [1:8192]
      C = CMPLX([1:8192],[1:8192])
      D = DCMPLX([1:8192],[1:8192])
      E = MOD([1:8192],2).EQ.0
      DEST = 0.0

C Open a file and write to it; add failure tests to each
C operation if desired.

      CALL CMF_FILE_OPEN(FILE_UNIT,'my-file',IOSTAT)
      IF (IOSTAT<0) PRINT *,"File open failed",IOSTAT

      CALL CMF_CM_ARRAY_TO_FILE(FILE_UNIT,A,IOSTAT)
      CALL CMF_CM_ARRAY_TO_FILE(FILE_UNIT,B,IOSTAT)
      CALL CMF_CM_ARRAY_TO_FILE(FILE_UNIT,C,IOSTAT)
      CALL CMF_CM_ARRAY_TO_FILE(FILE_UNIT,D,IOSTAT)
      CALL CMF_CM_ARRAY_TO_FILE(FILE_UNIT,E,IOSTAT)

C Rewind the file

      CALL CMF_FILE_REWIND(FILE_UNIT,IOSTAT)

C Compute the offset to the third record

      RECORD = CMF_SIZEOF_ARRAY_ELEMENT(A) +
     $     CMF_SIZEOF_ARRAY_ELEMENT(B)
```

```
C Seek to the third record

      CALL CMF_FILE_LSEEK(FILE_UNIT,RECORD,IOSTAT)

C Read the third record into array DEST

      CALL CMF_CM_ARRAY_FROM_FILE(FILE_UNIT,DEST,IOSTAT)

      STOP
      END
```

## 4.2 Parallel I/O via Devices and Sockets

The serial-order read and write utilities described above for the CM file system can also be used to transfer data via the CM-HIPPI or VME interfaces. In these cases the "file" is either a CM-HIPPI device or a CM socket, respectively. Operations on these devices require you to access the lower-level CM I/O library CMFS, as described in the CM I/O and CM-HIPPI documentation.

### Translating between File Descriptors and Unit Numbers

The CMFS procedures use file or socket descriptors to specify the desired "file." Two CM Fortran utility procedures translate between these descriptors and the unit numbers required by the CM Fortran utility I/O procedures.

One subroutine associates a CMFS file or socket descriptor of a previously opened "file" (or device) with a CM Fortran unit number.

```
CALL CMF_FILE_FDOPEN( CMFS_FD, UNIT, IOSTAT )
```

Both **CMFS_FD** and **UNIT** are input values; the procedure simply establishes an association between them. You can then call the CM Fortran utility read/write procedures, **CMF_CM_ARRAY_TO/FROM_FILE_SO**.

The other subroutine determines the CMFS file or socket descriptor that is already associated with a CM Fortran unit number:

```
CALL CMF_FILE_GET_FD( CMFS_FD, UNIT, IOSTAT )
```

`UNIT` is an input value; the value returned in `CMFS_FD` is the CMFS descriptor associated with it. This procedure is useful if you wish to use the descriptor in calls to the CMFS routines.

## I/O via Devices

To write or read data via devices, use the serial-order ("SO") I/O procedures.

Although the serial-order I/O procedures do not pad CM files, they *do* sometimes add extraneous data at the end of an array being written to a device. If you do not wish to deal with padding explicitly in the program, you can avoid it by observing certain restrictions on array size. These restrictions are reported in the documentation for CM-HIPPI.

# Appendix

Appendix

# Dictionary of Utility Procedures

This appendix provides reference information about the individual procedures in the CM Fortran Utility Library. The procedures are listed below by functional category. The dictionary entries that follow are alphabetical by procedure name.

**System Inquiry Functions**

```
ARCH = CMF_ARCHITECTURE ( )
NUM = CMF_NUMBER_OF_PROCESSORS ( )
BYTES = CMF_AVAILABLE_MEMORY ( )
```

**Array Inquiry Subroutine**

```
CMF_DESCRIBE_ARRAY ( ARRAY )
```

**Random Number Subroutines**

```
CMF_RANDOM ( DEST, LIMIT )
CMF_RANDOMIZE ( SEED )
```

**Dynamic Array Allocation Subroutines**

```
CMF_ALLOCATE_ARRAY
    ( ARRAY, EXTENTS, RANK, TYPE )
CMF_ALLOCATE_LAYOUT_ARRAY
    ( ARRAY, EXTENTS, RANK, TYPE, ORDERS, WEIGHTS )
CMF_ALLOCATE_DETAILED_ARRAY
    ( ARRAY, EXTENTS, RANK, TYPE, ORDERS, SUBGRIDS, PMASKS )
CMF_DEALLOCATE_ARRAY ( ARRAY )
```

**Array Transfer Subroutines**

CMF_FE_ARRAY_TO_CM ( DEST, SOURCE )

CMF_FE_ARRAY_FROM_CM ( DEST, SOURCE )

**Array Address Construction Procedures**

GEOMETRY = CMF_GET_GEOMETRY_ID ( ARRAY )

CMF_MAKE_SEND_ADDRESS ( ARRAY )

CMF_MY_SEND_ADDRESS ( ARRAY )

CMF_DEPOSIT_GRID_COORDINATE
( GEOMETRY, SEND_ADDRESS, AXIS, COORDINATE, MASK )

**Scatter-with-Combining Subroutine**

CMF_SEND_[ OVERWRITE | ADD | MAX | MIN | IOR | IAND | IEOR ]
( DEST, SEND_ADDRESS, SOURCE, MASK )

**Parallel Prefix Subroutine**

CMF_SCAN_[ COPY | ADD | MAX | MIN | IOR | IAND | IEOR ]
( DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
SEGMENT_MODE, MASK )

**Sorting Subroutines**

CMF_ORDER
( DEST, SOURCE, AXIS, MASK )

CMF_RANK
( DEST, SOURCE, SEGMENT, AXIS, DIRECTION,
SEGMENT_MODE, MASK )

CMF_SORT
( DEST, SOURCE, SEGMENT, AXIS, DIRECTION,
SEGMENT_MODE, MASK )

**Table Lookup Procedures**

TABLE = CMF_ALLOCATE_TABLE
( TYPE, ELEMENT_COUNT, INITIAL_VALUES )

CMF_DEALLOCATE_TABLE ( TABLE )

CMF_LOOKUP_IN_TABLE ( DEST, TABLE, INDEX, MASK )

**Gathers/Scatters on Serial Axes (Subroutines)**

CMF_AREF_1D ( DEST, ARRAY, INDEX, MASK )

CMF_ASET_1D ( ARRAY, SOURCE, INDEX, MASK )

## CM File Operations Procedures

CMF_FILE_OPEN ( UNIT, PATH, IOSTAT )

CMF_FILE_CLOSE ( UNIT, IOSTAT )

CMF_FILE_UNLINK ( PATH, IOSTAT )

CMF_CM_ARRAY_TO_FILE ( UNIT, SOURCE, IOSTAT )

CMF_CM_ARRAY_FROM_FILE ( UNIT, DEST, IOSTAT )

CMF_CM_ARRAY_TO_FILE_FMS ( UNIT, SOURCE, IOSTAT )

CMF_CM_ARRAY_FROM_FILE_FMS ( UNIT, DEST, IOSTAT )

CMF_CM_ARRAY_TO_FILE_SO ( UNIT, SOURCE, IOSTAT )

CMF_CM_ARRAY_FROM_FILE_SO ( UNIT, DEST, IOSTAT )

CMF_FILE_LSEEK ( UNIT, OFFSET, IOSTAT )

CMF_FILE_LSEEK_FMS ( UNIT, OFFSET, IOSTAT )

CMF_FILE_REWIND ( UNIT, IOSTAT )

CMF_FILE_TRUNCATE ( UNIT, LENGTH, IOSTAT )

SIZEOF = CMF_SIZEOF_ARRAY_ELEMENT ( ARRAY )

## CM I/O via Sockets or Devices (Subroutines)

CMF_FILE_FDOPEN ( CMFS_FD, UNIT, IOSTAT )

CMF_FILE_GET_FD ( UNIT, CMFS_FD, IOSTAT )

## NAME

**CMF_ALLOCATE_ARRAY** – Allocates a CM array dynamically.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_ALLOCATE_ARRAY(ARRAY, EXTENTS, RANK, TYPE)
```

## ARGUMENTS

*ARRAY* Front-end array of integers. This front-end array must have `CMF_SIZEOF_DESCRIPTOR` elements of type `INTEGER`. (`CMF_SIZEOF_DESCRIPTOR` is a predefined constant.) This argument will be modified to point to the allocated CM memory.

*EXTENTS*
Front-end array of at least *RANK* integers. This array contains the length of each axis of the array to be created. The first element specifies the length of axis 1, the second element specifies the length of the second axis, and so on. The axes will default to `CMF_NEWS_ORDER` ordering.

*RANK* Integer. The rank of the array to be created.

*TYPE* Integer. The type of the array to be created. This is one of the following integer values:

- `CMF_LOGICAL`
- `CMF_S_INTEGER`
- `CMF_FLOAT`
- `CMF_DOUBLE`
- `CMF_COMPLEX`
- `CMF_DOUBLE_COMPLEX`

## RETURNED VALUE

None.

## DESCRIPTION

The subroutine `CMF_ALLOCATE_ARRAY` allocates CM storage to hold an array of the shape specified by *RANK* and *EXTENTS*, and of the type specified by *TYPE*. *ARRAY* is modified to serve as a descriptor for the array.

To use the elements of the CM array created by `CMF_ALLOCATE_ARRAY` in CM Fortran operations, you must pass *ARRAY* to a program unit that explicitly declares it as a CM array. Since the program unit that calls `CMF_ALLOCATE_ARRAY` must declare *ARRAY* as a front-end array, *ARRAY* cannot be used in that program unit except to be passed to other program units. See the example given below.

### NOTE

Do not use the compiler switches `-safety=`*level* or `-argument_checking` when compiling programs that use dynamically allocated arrays.

## EXAMPLE

This example illustrates the standard method for using `CMF_ALLOCATE_ARRAY`. In the `ALLOCATE` subroutine, *NEW_ARRAY* is declared as a front-end array and modified by the call to `CMF_ALLOCATE_ARRAY` to point to the CM memory allocated for the array. *NEW_ARRAY* is then passed to the subroutine `PRINT_DIMS3D` which declares and uses it as a CM array.

```
      SUBROUTINE ALLOCATE()
      IMPLICIT NONE
      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER NEW_ARRAY(CMF_SIZEOF_DESCRIPTOR), EXTENTS(7), RANK, I
      PARAMETER (RANK=3)
C
      DO I=1, RANK
         EXTENTS(I) = I * 10
      END DO
C
      CALL CMF_ALLOCATE_ARRAY(NEW_ARRAY, EXTENTS, RANK, CMF_S_INTEGER)
C
      CALL PRINT_DIMS3D(NEW_ARRAY)
C
      CALL CMF_DEALLOCATE_ARRAY(NEW_ARRAY)
      END SUBROUTINE ALLOCATE
C
C---------------------------------------------------------------------
C
      SUBROUTINE PRINT_DIMS3D(IN)
      IMPLICIT NONE
      INTEGER IN(:,:,:)
C
      PRINT *,"Shape of DUMMY is (",DUBOUND(IN,1),
     &       ",",DUBOUND(IN,2),
     &       ",",DUBOUND(IN,3),")"
C
      END SUBROUTINE PRINT_DIMS3D
```

## SEE ALSO

```
CMF_ALLOCATE_DETAILED_ARRAY
CMF_ALLOCATE_LAYOUT_ARRAY
CMF_DEALLOCATE_ARRAY
```

## NAME

**CMF_ALLOCATE_DETAILED_ARRAY** – Allocates a CM array dynamically with a specified detailed layout.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_ALLOCATE_DETAILED_ARRAY(ARRAY, EXTENTS, RANK, TYPE, ORDERS,
                                 SUBGRIDS, PMASKS)
```

## ARGUMENTS

*ARRAY* Front-end array of integers. This front-end array must have `CMF_SIZEOF_DESCRIPTOR` elements of type `INTEGER`. (`CMF_SIZEOF_DESCRIPTOR` is a predefined constant.) This argument will be modified to point to allocated CM memory when the array is passed as an argument to a subprogram.

*EXTENTS*
Front-end array of at least *RANK* integers. This array contains the length of each axis of the array to be created. The first element specifies the length of axis 1, the second element specifies the length of axis 2, and so on.

*RANK* Integer. The rank of the array to be created.

*TYPE* Integer. The type of the array to be created. This is one of the following integer values:

- `CMF_LOGICAL`
- `CMF_S_INTEGER`
- `CMF_FLOAT`
- `CMF_DOUBLE`
- `CMF_COMPLEX`
- `CMF_DOUBLE_COMPLEX`

*ORDERS*
Front-end array of integers. This array specifies the ordering of each axis of the array to be created. Each element of this array must be one of the following integer values:

- `CMF_NEWS_ORDER`
- `CMF_SERIAL_ORDER`

The axes will default to `CM_NEWS_ORDER` ordering.

Use `CMF_NEWS_ORDER` for axes for which *SUBGRIDS* and *PMASKS* values are specified. Anywhere `CMF_SERIAL_ORDER` is used for an axis, the correspoinding *PMASKS* value must be 0, and the *SUBGRIDS* value must be the axis extent.

*SUBGRIDS*
: Front-end array of integers. This array indicates the desired subgrid length for each axis.

*PMASKS*
: Front-end array of integers. The integers in this array serve as bitmasks to indicate the desired processors. If the *ORDERS* argument contains the value `CMF_SERIAL_ORDER` for any axis, the *PMASKS* argument must contain 0 for that axis.

## RETURNED VALUE

None.

## DESCRIPTION

The subroutine `CMF_ALLOCATE_DETAILED_ARRAY` allocates the CM storage to hold an array of the shape specified by *RANK* and *EXTENTS*, the type specified by *TYPE*, and with `CMF_NEWS_ORDER` ordering.

To use the elements of CM array created by `CMF_ALLOCATE_DETAILED_ARRAY` in CM Fortran operations, you must pass *ARRAY* to a program unit that explicitly declares it as a CM array. Since the program unit that calls `CMF_ALLOCATE_DETAILED_ARRAY` must declare *ARRAY* as a front-end array, *ARRAY* cannot be used in that program unit except to be passed to other program units.

The *SUBGRIDS* and *PMASKS* arguments enable you to specify in detail how the CM array is laid out on the parallel processing elements (CM-5 vector units, CM-5 nodes, or CM-2/200 nodes, depending on the execution model). For each array axis, the value in the SUBGRIDS argument specifies the number of elements in the subgrid in each processing element. The value in PMASKS is a bit-mask that specifies which processing elements are used.

## NOTES

Do not use the compiler switch `-safety=`*level* or `-argument_checking` to compile programs that contain dynamically allocated arrays.

`CMF_ALLOCATE_DETAILED_ARRAY` cannot be used under the Paris execution model on a CM-2/200 system.

## EXAMPLE

The following program illustrates CMF_ALLOCATE_DETAILED_ARRAY. Notice the use of the assumed-layout directive when the new array is passed as an argument to a subprogram.

```
      IMPLICIT NONE
      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER NEWARRAY(CMF_SIZEOF_DESCRIPTOR)
      INTEGER EXTENTS(7),ORDERS(7),SUBGRIDS(7),PMASKS(7)
      INTEGER RANK,I
      INTEGER NPN,NPN_FRAC,FRAC,SG1,SG2
      REAL A(200)

      PARAMETER (RANK = 2)
      PARAMETER (FRAC = 4)
      PARAMETER (SG1 = 5, SG2 = 40)

      A = 1.0      ! initialize if CM-2 running in auto-attach mode

      NPN = CMF_NUMBER_OF_PROCESSORS()
      NPN_FRAC = NPN/FRAC

      PMASKS(1) = (NPN_FRAC - 1) * FRAC
      PMASKS(2) = FRAC - 1

      SUBGRIDS(1) = SG1
      SUBGRIDS(2) = SG2

      EXTENTS(1) = NPN_FRAC * SG1
      EXTENTS(2) = FRAC * SG2

      DO I = 1,RANK
        ORDERS(I) = CMF_NEWS_ORDER
      END DO

      CALL CMF_ALLOCATE_DETAILED_ARRAY
     & (NEWARRAY,EXTENTS,RANK,CMF_FLOAT,ORDERS,SUBGRIDS,PMASKS)

      CALL USE_NEWARRAY(NEWARRAY,EXTENTS)

      CALL CMF_DEALLOCATE_ARRAY(NEWARRAY)

      STOP
      END


      SUBROUTINE USE_NEWARRAY(A,EXT)
      INTEGER EXT(2)
      REAL A(EXT(1),EXT(2)), B(EXT(1),EXT(2))
CMF$  LAYOUT A(:,:)
```

```
CMF$  ALIGN  B(I,J) WITH A(I,J)

      B = CSHIFT(A,DIM=1,SHIFT=1)

C     Other operations on arrays A and B

      RETURN
      END
```

## SEE ALSO

CMF_ALLOCATE_ARRAY
CMF_DEALLOCATE_ARRAY
CMF_ALLOCATE_LAYOUT_ARRAY

## NAME

**CMF_ALLOCATE_LAYOUT_ARRAY** – Allocates a CM array with a specified layout.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'
```

CALL CMF_ALLOCATE_LAYOUT_ARRAY (*ARRAY*, *EXTENTS*, *RANK*, *TYPE*, *ORDERS*, *WEIGHTS*

## ARGUMENTS

*ARRAY* Front-end array of integers. This front-end array must have `CMF_SIZEOF_DESCRIPTOR` elements of type `INTEGER`. (`CMF_SIZEOF_DESCRIPTOR` is a predefined constant.) This argument will be modified to point to the allocated CM memory.

*EXTENTS*
Front-end array of at least *RANK* integers. This array contains the length of each axis of the array to be created. The first element specifies the length of axis 1, the second element specifies the length of axis 2, and so on. The axes will have the ordering specified by the *ORDERS* and *WEIGHTS* arguments.

*RANK* Integer. The rank of the array to be created.

*TYPE* Integer. The type of the array to be created. This is one of the following integer values:

- `CMF_LOGICAL`
- `CMF_S_INTEGER`
- `CMF_FLOAT`
- `CMF_DOUBLE`
- `CMF_COMPLEX`
- `CMF_DOUBLE_COMPLEX`

*ORDERS*
Front-end array of integers. This array specifies the ordering of each axis of the array to be created. Each element of this array must be one of the following integer values:

- `CMF_NEWS_ORDER`
- `CMF_SERIAL_ORDER`
- `CMF_SEND_ORDER`

*WEIGHTS*
Front-end array of non-negative integers. This array specifies the weight, or expected heaviness of use, of each axis of the array to be created. The *WEIGHTS* array should be initialized to all ones if no special weighting of axes is required.

## RETURNED VALUE

None.

## DESCRIPTION

The subroutine CMF_ALLOCATE_LAYOUT_ARRAY allocates the CM storage to hold an array of the shape specified by *RANK* and *EXTENTS*, the type specified by *TYPE*, and with ordering and weights specified for each axis by *ORDERS* and *WEIGHTS*.

To use the elements of CM array created by CMF_ALLOCATE_LAYOUT_ARRAY in CM Fortran operations, you must pass *ARRAY* to a program unit that explicitly declares it as a CM array. Since the program unit that calls CMF_ALLOCATE_LAYOUT_ ARRAY must declare *ARRAY* as a front-end array, *ARRAY* cannot be used in that program unit except to be passed to other program units.

### NOTE

Do not use the compiler switch `-safety=`*level* or `-argument_checking` to compile programs that contain dynamically allocated arrays.

## SEE ALSO

CMF_ALLOCATE_ARRAY
CMF_ALLOCATE_DETAILED_ARRAY
CMF_DEALLOCATE_ARRAY

## NAME

**CMF_ALLOCATE_TABLE** – Allocates a lookup table and returns a table identifier.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'
```

*TABLE_ID* = CMF_ALLOCATE_TABLE (*TYPE*, *ELEMENT_COUNT*, *INITIAL_VALUES*)

## ARGUMENTS

*TYPE* Integer. *TYPE* describes the type of the elements to be allocated for the table. Valid values are:

- CMF_LOGICAL
- CMF_S_INTEGER
- CMF_FLOAT
- CMF_DOUBLE
- CMF_COMPLEX
- CMF_DOUBLE_COMPLEX

*ELEMENT_COUNT*
An INTEGER specifying the number of elements in the lookup table.

*INITIAL_VALUES*
A front-end array of the same type as *TYPE* containing the values to be used to initialize the table.

**NOTE:** This routine assumes that the front-end array *INITIAL_VALUES* has a lower bound of 1. All other lower bound values are ignored.

## RETURNED VALUE

An INTEGER used as an identifier for the lookup table. This value must be passed to the other CM Fortran utility routines, CMF_LOOKUP_IN_TABLE and CMF_DEALLOCATE_TABLE, that operate on this table.

## DESCRIPTION

CMF_ALLOCATE_TABLE allocates a table as specified by *TYPE* and *ELEMENT_COUNT*, initializes it to the values specified in *INITIAL_VALUES*, and returns a table identifier. Values can be retrieved from this table using parallel array referencing by passing the table identifier to CMF_LOOKUP_IN_TABLE.

Using `CMF_ALLOCATE_TABLE` and `CMF_LOOKUP_IN_TABLE` to perform indirect indexing is significantly faster than using a conventionally allocated table when:

- The content of the table never or rarely changes.
- The table is relatively small. Specifically, it must use less memory than is available on a single processing element (vector unit, node, or processor, depending on the execution model).

For more detail on using these tables, see the man page for `CMF_LOOKUP_IN_TABLE`.

## SEE ALSO

```
CMF_LOOKUP_IN_TABLE
CMF_DEALLOCATE_TABLE
```

## NAME

**CMF_ARCHITECTURE** – Identifies current CM model and execution model.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

ARCH = CMF_ARCHITECTURE ( )
```

## ARGUMENTS

None.

## RETURNED VALUE

Returns an `INTEGER` constant. Valid values are one of the following:

- `CMF_CM5_SPARC`
- `CMF_CM5_VU`
- `CMF_CM200_SLICEWISE`
- `CMF_CM2_SLICEWISE`
- `CMF_CM200_PARIS`
- `CMF_CM2_PARIS`
- `CMF_SIM`

## DESCRIPTION

This function returns a constant that identifies the CM model (CM-2, CM-200, or CM-5) and the execution model under which a program is running. On the CM-2/200 the execution model can be Paris or slicewise. On the CM-5 the execution model can be `SPARC` indicating a CM-5 without vector units, or `VU` indicating a CM-5 that contains vector units in addition to the Sparc processors. Finally, `CMF_SIM` indicates that the program is running on a Sun computer under the CM Fortran simulator.

## SEE ALSO

`CMF_NUMBER_OF_PROCESSORS`

## NAME

**CMF_AREF_1D** – Extracts array-indexed values from the serial axis of a CM array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_AREF_1D(DEST, ARRAY, INDEX, MASK)
```

## ARGUMENTS

| | |
|---|---|
| *DEST* | A CM array of the same type as *ARRAY* and conforming to *INDEX* and *MASK*. Values referenced from *ARRAY* are stored in *DEST*. |
| *ARRAY* | The CM array to be referenced. *ARRAY* must be of the same type as *DEST* and have one more dimension than *DEST*, *INDEX*, and *MASK*. The first axis must have `:SERIAL` ordering and all axes after the first must have the same shape and layout as the other arguments. |
| *INDEX* | A CM `INTEGER` array conforming to *DEST* and *MASK*. These values are used as indices into the `:SERIAL` axis of *ARRAY*. |
| *MASK* | A CM `LOGICAL` array conforming to *DEST* and *INDEX*, or the scalar value `.TRUE.`. If *MASK* is the scalar value `.TRUE.`, all the elements of *DEST* are modified. If *MASK* is a `LOGICAL` array, only the elements of *DEST* corresponding to the elements of *MASK* that contain `.TRUE.` are modified. |

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine places into selected elements of *DEST* the value from the first axis of *ARRAY* referenced by the corresponding elements of *INDEX*. The elements selected are those that correspond to a `.TRUE.` element in *MASK*. Note that even though `CMF_AREF_1D` operates only on the first axis of *ARRAY*, *ARRAY* must have a rank greater than one.

`CMF_AREF_1D` uses indirect addressing hardware to perform this reference significantly faster than the equivalent CM Fortran code. (See the example provided below.)

### NOTES

This subroutine is significantly faster when

- *MASK* is the scalar value `.TRUE.`.

- The product of the dimensions of *INDEX* is an integer multiple of the number of nodes or processors available to the program. (The number of processing elements is returned by the function CMF_NUMBER_OF_PROCESSORS.)

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank that have the individual axes offset.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## EXAMPLE

The DO loop, the FORALL statement, and the call to CMF_AREF_1D given below are all equivalent, but the call to CMF_AREF_1D is significantly faster.

```
      INTEGER I, DEST(8192), ARRAY(10,8192), INDEX(8192)
CMF$LAYOUT ARRAY(:SERIAL, :NEWS)
      LOGICAL MASK(8192)
C     ...
      DO I=1, 8192
        IF (MASK(I)) DEST(I) = ARRAY(INDEX(I), I)
      END DO
C     ...
      FORALL(I=1:8192, MASK(I)) DEST(I) = ARRAY(INDEX(I),I)
C     ...
      CALL CMF_AREF_1D(DEST, ARRAY, INDEX, MASK)
```

## NAME

**CMF_ASET_1D** – Stores values into the serial axis of a CM array at array-indexed locations.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_ASET_1D(ARRAY, SOURCE, INDEX, MASK)
```

## ARGUMENTS

*ARRAY* A CM array of the same type as *SOURCE*. *ARRAY* must have one more dimension than *SOURCE*, *INDEX*, and *MASK*. The first axis must have `:SERIAL` ordering, and all axes after the first must have the same shape and layout as the other arguments. Values referenced from *SOURCE* are stored in *ARRAY*.

*SOURCE*
A CM array of the same type as *ARRAY* and the same shape and layout as *INDEX* and *MASK*.

*INDEX* A CM `INTEGER` array of the same shape and layout as *SOURCE* and *MASK*. These values are used as indices into the `:SERIAL` axis of *ARRAY*, specifying the location at which to store the corresponding value of *SOURCE*.

*MASK* A CM `LOGICAL` array of the same shape and layout as *SOURCE* and *INDEX*, or the scalar value `.TRUE.`. If *MASK* is the scalar value .TRUE., all the elements of *SOURCE* are stored. If *MASK* is a `LOGICAL` array, only the elements of *SOURCE* corresponding to the elements of *MASK* that contain .TRUE. are stored.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine stores selected elements of *SOURCE* into the first (serial) axis of *ARRAY* at the locations specified by the corresponding elements of *INDEX*. The elements selected are those that correspond to a `.TRUE.` element in *MASK*. Though `CMF_ASET_1D` operates on only a single axis, *ARRAY* must have a rank greater than one.

`CMF_ASET_1D` uses indirect addressing hardware to perform this reference significantly faster than the equivalent CM Fortran code.

## NOTES

This subroutine is significantly faster when

- *MASK* is the scalar value `.TRUE.`.
- The product of the dimensions of *INDEX* is an integer multiple of the number of nodes or processors available to the program. (The number of processing elements is returned by the function `CMF_NUMBER_OF_PROCESSORS`.)

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## NAME

**CMF_AVAILABLE_MEMORY** – Returns the number of bytes available in each node or processor.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

MEM = CMF_AVAILABLE_MEMORY ()
```

## RETURNED VALUE

An `INTEGER` specifying the number of bytes of memory available in each node or processor.

## DESCRIPTION

This function returns an integer reporting, in units of bytes, the amount of memory left in each processing element: node for CM-5 sparc model, or CM-2/200 slicewise; vector unit for CM-5 vu model; or processor for CM-2/200 Paris.

Note: This function returns incorrect results for the vector unit model in Version 2.0 Beta.

## NAME

**CMF_CM_ARRAY_FROM_FILE** – Reads an array from a CM file.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_CM_ARRAY_FROM_FILE(UNIT, DEST, IOSTAT)
```

## ARGUMENTS

| | |
|---|---|
| *UNIT* | An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This `UNIT` number has no relation to standard Fortran unit numbers used in front-end I/O. |
| *DEST* | A CM array of any type. The `DEST` array must be identical in shape and type to the array that is to be transferred from the file. |
| *IOSTAT* | An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success, a value of 0 indicates an end-of-file condition, and a negative value indicates failure. |

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine reads an array from the CM file specified by `UNIT` into the CM array specified by `DEST`. The file must have been written by `CMF_CM_ARRAY_TO_FILE`. `CMF_CM_ARRAY_TO_FILE` writes the array to the file in a parallel order that reflects the geometry of the array. This allows `CMF_CM_ARRAY_FROM_FILE` to transfer the array faster than `CMF_CM_ARRAY_FROM_FILE_SO`. However, the files written by `CMF_CM_ARRAY_TO_FILE` cannot be transferred outside the CM system and are subject to the following restrictions when being read back into the CM system:

- The machine used to read the file must be the same model (CM-2/200 or CM-5) that was used to write the file.
- The `DEST` array must be identical in shape and type to the array in the file.
- Files read by `CMF_CM_ARRAY_FROM_FILE` must have been written by `CMF_CM_ARRAY_TO_FILE`. The machine size, array layout, and execution model can be different between the write and read operations, with the following exceptions:

- On CM-2/200 only, an array written from one execution model (Paris or slicewise) and read into the other execution model must have canonical layout.

  A canonical array is one in which the axis ordering or weights have not been changed from the defaults by the `LAYOUT` directive. Within a program, a noncanonical array can be converted to a canonical array by an array assignment.

- On CM-2/200 only, an array written from one machine size and read into a different machine size must have at least as many elements as the number of bit-serial processors in the larger machine.

More specialized parallel order files can be written with `CMF_CM_ARRAY_TO_FILE_FMS` and read with `CMF_CM_ARRAY_FROM_FILE_FMS`. These subroutines write and read arrays to CM files more quickly than `CMF_CM_ARRAY_TO_FILE` and `CMF_CM_ARRAY_FROM_FILE`, but they are more restricted in their use. In particular, a file must be read on the same size machine as it was written from.

Serial order files are written with `CMF_CM_ARRAY_TO_FILE_SO` and read with `CMF_CM_ARRAY_FROM_FILE_SO`. Such files can be transferred between CM-2/200 and CM-5 sytems, outside the CM file system, or directly to an I/O device such as a HIPPI interface or a CM socket.

### NOTE

The CM Fortran Utility Library procedures do not operate correctly on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_CM_ARRAY_FROM_FILE_FMS
CMF_CM_ARRAY_FROM_FILE_SO
CMF_CM_ARRAY_TO_FILE
CMF_CM_ARRAY_TO_FILE_FMS
CMF_CM_ARRAY_TO_FILE_SO
```

## NAME

**CMF_CM_ARRAY_FROM_FILE_FMS** – Reads an array from a CM file to a CM array for a fixed machine size.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CMF_CM_ARRAY_FROM_FILE_FMS (UNIT, DEST, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This `UNIT` number has no relation to standard Fortran unit numbers used in front-end I/O.

*DEST* A CM array of any type. The `DEST` array must be identical in shape, type, and layout to the array that is to be transferred from the file.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success, a value of zero indicates an end-of-file condition, and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine reads the contents of an array from the file specified by `UNIT` and stores it in the `DEST` CM array. The file must have been written by `CMF_CM_ARRAY_TO_FILE_FMS`. `CMF_CM_ARRAY_TO_FILE_FMS` writes the file in a parallel order that reflects the geometry of the array, the array layout, and the size of the machine executing the program. This allows `CMF_CM_ARRAY_FROM_FILE_FMS` to transfer the array substantially faster than `CMF_CM_ARRAY_FROM_FILE` or `CMF_CM_ARRAY_FROM_FILE_SO`. However, parallel order files cannot be transferred outside the CM file system, and FMS files are subject to the following restrictions when being read back into the CM:

- The array must be read by `CMF_CM_ARRAY_FROM_FILE_FMS` into the same machine model (CM-2/200 or CM-5) that was used to write the file. Files written by `CMF_CM_ARRAY_TO_FILE_FMS` are not portable between CM-2/200 and CM-5 systems.
- The machine used to read the array must be a CM-2/200 section or a CM-5

partition with the same physical size as the one that was used to write the file. In addition, the same execution model (slicewise or Paris nodes or vector units) must be used when writing and reading.

- As mentioned in the description of the `DEST` argument above, the destination array on the CM and the array that is to be transferred from the file must be identical in shape, type, and layout.

More general parallel order files that have some of the performance advantages of FMS files but less severe restrictions can be written with `CMF_CM_ARRAY_TO_FILE` and read with `CMF_CM_ARRAY_FROM_FILE`.

Serial order files that can be transferred, between CM-2 and CM-5 systems, outside the CM file system, or directly to an I/O device such as a HIPPI interface or a CM socket, can be written with `CMF_CM_ARRAY_TO_FILE_SO` and read with `CMF_CM_ARRAY_FROM_FILE_SO`.

**NOTE**

The CM Fortran Utility Library procedures do not operate correctly on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_CM_ARRAY_FROM_FILE
CMF_CM_ARRAY_FROM_FILE_SO
CMF_CM_ARRAY_TO_FILE
CMF_CM_ARRAY_TO_FILE_FMS
CMF_CM_ARRAY_TO_FILE_SO
```

## NAME

**CMF_CM_ARRAY_FROM_FILE_SO** – Reads an array from a CM file to a CM array in serial order.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_CM_ARRAY_FROM_FILE_SO(UNIT, DEST, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This `UNIT` number has no relation to standard Fortran unit numbers used in front-end I/O.

*DEST* A CM array of any type, shape, or layout.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and specifies the number of bytes read from the file, a value of zero indicates an end-of-file condition, and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

`CMF_CM_ARRAY_FROM_FILE_SO` reads an array from the CM file specified by `UNIT` into the CM array `DEST`. `CMF_CM_ARRAY_FROM_FILE_SO` expects the array to be in normal Fortran (or "serial") order, that is, an array written with `CMF_CM_ARRAY_TO_FILE_SO` or with the Fortran 77 (and CM Fortran) `WRITE` statement.

### NOTE

The CM Fortran Utility Library procedures do not operate correctly on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_CM_ARRAY_FROM_FILE
CMF_CM_ARRAY_FROM_FILE_FMS
CMF_CM_ARRAY_TO_FILE
CMF_CM_ARRAY_TO_FILE_FMS
CMF_CM_ARRAY_TO_FILE_SO
```

## NAME

**CMF_CM_ARRAY_TO_FILE** – Writes the contents of an array to a CM file.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_CM_ARRAY_TO_FILE(UNIT, SOURCE, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This `UNIT` number has no relation to the standard Fortran unit numbers used in front-end I/O.

*SOURCE*
A CM array of any type.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success, and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine writes the contents of the CM array `SOURCE` to the CM file specified by `UNIT`. The array is written to the file in a parallel order that reflects the geometry of the array. This allows `CMF_CM_ARRAY_TO_FILE` to transfer the array substantially faster than `CMF_CM_ARRAY_TO_FILE_SO`. However, files written with `CMF_CM_ARRAY_TO_FILE` cannot be transferred outside the CM file system and are subject to the following restrictions when being used on the CM:

- All arrays written to the file must have the same shape as the first array written to the file.
- The array must be read by `CMF_CM_ARRAY_FROM_FILE` into the same machine model (CM-2/200 or CM-5) that was used to write the file. Files written by `CMF_CM_ARRAY_TO_FILE` are not portable between CM-2/200 and CM-5 systems.
- Files written by `CMF_CM_ARRAY_TO_FILE` are, in most cases, portable across machine sizes, array layouts, and execution models. The exceptions are:

- On CM-2/200 only, an array written from one execution model (Paris or slicewise) and read into the other execution model must have canonical layout.

  A canonical array is one in which the axis ordering or weights have not been changed from the defaults by the `LAYOUT` directive. Within a program, a noncanonical array can be converted to a canonical array by an array assignment.

- On CM-2/200 only, an array written from one machine size and read into a different machine size must have at least as many elements as the number of bit-serial processors in the larger machine.

More specialized parallel order files can be written with `CMF_CM_ARRAY_TO_FILE_FMS` and read with `CMF_CM_ARRAY_FROM_FILE_FMS`. These subroutines write and read arrays to CM files more quickly than `CMF_CM_ARRAY_TO_FILE` and `CMF_CM_ARRAY_FROM_FILE`, but they are more severly restricted in their use. In particular, a file must be read on the same size machine as it was written from.

Serial order files are written with `CMF_CM_ARRAY_TO_FILE_SO` and read with `CMF_CM_ARRAY_FROM_FILE_SO`. Such files can be transferred between CM-2/200 and CM-5 sytems, outside the CM file system, or directly to an I/O device such as a HIPPI interface or a CM socket.

## NOTE

The CM Fortran Utility Library procedures do not operate correctly on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_CM_ARRAY_FROM_FILE
CMF_CM_ARRAY_FROM_FILE_FMS
CMF_CM_ARRAY_FROM_FILE_SO
CMF_CM_ARRAY_TO_FILE_FMS
CMF_CM_ARRAY_TO_FILE_SO
```

## NAME

**CMF_CM_ARRAY_TO_FILE_FMS** – Writes the contents of an array to a CM file for a fixed machine size.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_CM_ARRAY_TO_FILE_FMS (UNIT, SOURCE, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This `UNIT` number has no relation to standard Fortran unit numbers used in front-end I/O.

*SOURCE*
A CM array of any type.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success, and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine writes the contents of the CM array `SOURCE` to the CM file specified by `UNIT`. The array is written to the file in a parallel order that reflects the geometry of the array, the array layout, and the size of the machine executing the program. This allows `CMF_CM_ARRAY_TO_FILE_FMS` to transfer the array substantially faster than `CMF_CM_ARRAY_TO_FILE` or `CMF_CM_ARRAY_TO_FILE_SO`. However, parallel order files cannot be transferred outside the CM file system, and FMS files are subject to the following restrictions when being used on the CM:

- All arrays written to the file must have the same shape and layout as the first array written.
- The array must be read by `CMF_CM_ARRAY_FROM_FILE_FMS` into the same machine model (CM-2/200 or CM-5) and the same execution model (slicewise or Paris, nodes or vector units) that were used to write the file.
- The array must be read into a CM-2/200 section or a CM-5 partition with the same physical size as the one that was used to write the file.

- The array from the file must be read into an array on the CM with the same shape, type, and layout.

More general parallel order files that have some of the performance advantages of FMS files but less severe restrictions can be written with `CMF_CM_ARRAY_TO_FILE` and read with `CMF_CM_ARRAY_FROM_FILE`.

Serial order files that can be transferred between CM-2/200 and CM-5 systems, outside the CM file system, or directly to an I/O device such as a HIPPI interface or a CM socket, can be written with `CMF_CM_ARRAY_TO_FILE_SO` and read with `CMF_CM_ARRAY_FROM_FILE_SO`.

### NOTE

The CM Fortran Utility Library procedures do not operate correctly on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_CM_ARRAY_FROM_FILE
CMF_CM_ARRAY_FROM_FILE_FMS
CMF_CM_ARRAY_FROM_FILE_SO
CMF_CM_ARRAY_TO_FILE
CMF_CM_ARRAY_TO_FILE_SO
```

## NAME

**CMF_CM_ARRAY_TO_FILE_SO** – Writes the contents of an array to a CM file in serial order.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_CM_ARRAY_TO_FILE_SO(UNIT, SOURCE, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the the CM file system. This `UNIT` number has no relation to standard Fortran unit numbers used in front-end I/0.

*SOURCE*
A CM array of any type, shape, or layout.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success, and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

`CMF_CM_ARRAY_TO_FILE_SO` writes the contents of the `SOURCE` array to a CM file specified by `UNIT` in normal Fortran (or "serial") order. These arrays must be read back into the CM system with `CMF_CM_ARRAY_FROM_FILE_SO`.

The array elements are stored in a serial order file in the same order as those written with the Fortran 77 (and CM Fortran) `WRITE` statement. For example, the array A(2,3) is stored in the following order:

A(1,1)
A(2,1)
A(1,2)
A(2,2)
A(1,3)
A(2,3)

Files containing arrays in serial order are portable to any CM configuration and may also be transferred outside the CM system to other file systems. However, arrays that are written by `CMF_CM_ARRAY_TO_FILE_SO` directly to a device (a HIPPI interface or a CM socket) may contain some "padding".

The padding consists of extra elements added to the array when it is allocated in CM memory or when the I/O system writes it out. The padding is handled transparently by the CM Fortran I/O utilties that read and write in parallel order (`CMF_CM_ARRAY_TO/FROM_FILE` and `CMF_CM_ARRAY_TO/FROM_FILE_FMS`), and the padding is stripped from the arrays when they are written to a file in serial order by `CMF_CM_ARRAY_TO_FILE_SO`. However, when `CMF_CM_ARRAY_TO_FILE_SO` is used to write an array to a device, extraneous data may be added to the end of the array. Padding is not present if the following restrictions are observed when writing to devices:

- From the CM-5:

  Write from arrays whose size (number of elements) is a power of 2 and an integer multiple of the size of the partition (number of nodes) executing the program.

- From the CM-2/200:

  Write from arrays whose size (number of elements) is a power of 2 and an integer mulitple of the size of the machine (number of *bit-serial* processors) executing the program. The I/O system considers the number of bit-serial processors to be the CM-2/200 machine size under either execution model, Paris or slicewise.

**NOTE**

The CM Fortran Utility Library procedures do not operate correctly on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_CM_ARRAY_FROM_FILE
CMF_CM_ARRAY_FROM_FILE_FMS
CMF_CM_ARRAY_FROM_FILE_SO
CMF_CM_ARRAY_TO_FILE
CMF_CM_ARRAY_TO_FILE_FMS
```

## NAME

**CMF_DEALLOCATE_ARRAY** – Deallocates a dynamically allocated CM array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_DEALLOCATE_ARRAY (ARRAY)
```

## ARGUMENTS

*ARRAY* Front-end array. The *ARRAY* argument modified by `CMF_ALLOCATE_ARRAY`, `CMF_ALLOCATE_DETAILED_ARRAY`, or `CMF_ALLOCATE_LAYOUT_ARRAY` as a descriptor for a dynamically allocated CM array. The CM array represented by this argument will be deallocated.

## RETURNED VALUE

None.

## DESCRIPTION

`CMF_DEALLOCATE_ARRAY` deallocates a CM array that has been allocated with `CMF_ALLOCATE_ARRAY`, `CMF_ALLOCATE_DETAILED_ARRAY`, or `CMF_ALLOCATE_LAYOUT_ARRAY`. Only *ARRAY* arguments modified by these three subroutines should be passed to this subroutine. The contents of the CM array represented by *ARRAY* cannot be accessed after a call to this subroutine.

## SEE ALSO

```
CMF_ALLOCATE_ARRAY
CMF_ALLOCATE_DETAILED_ARRAY
CMF_ALLOCATE_LAYOUT_ARRAY
```

## NAME

**CMF_DEALLOCATE_TABLE** – Deallocates all storage associated with a lookup table allocated by `CMF_ALLOCATE_TABLE`.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_DEALLOCATE_TABLE ( TABLE )
```

## ARGUMENTS

*TABLE* An `INTEGER`. The identifier, as returned by `CMF_ALLOCATE_TABLE`, for the table to be deallocated. Only tables allocated by the `CMF_ALLOCATE_TABLE` subroutine can be deallocated by this procedure.

## RETURNED VALUE

None.

## DESCRIPTION

`CMF_DEALLOCATE_TABLE` deallocates all storage associated with a lookup table allocated by `CMF_ALLOCATE_TABLE`. Under some circumstances, these tables allow significantly faster access for vector indirection on invariant arrays than conventionally allocated arrays.

See the man page for `CMF_LOOKUP_IN_TABLE` for more details.

## SEE ALSO

```
CMF_ALLOCATE_TABLE
CMF_LOOKUP_IN_TABLE
```

## NAME

**CMF_DEPOSIT_GRID_COORDINATE** – Modifies a send address to incorporate specific grid coordinates.

## SYNTAX

```
 INCLUDE '/usr/include/cm/CMF_defs.h'

 CALL CMF_DEPOSIT_GRID_COORDINATE
&      (GEOMETRY, SEND_ADDRESS, AXIS, COORDINATE, MASK)
```

## ARGUMENTS

*GEOMETRY*
An INTEGER geometry ID as returned by CMF_GET_GEOMETRY_ID. The send address is computed for the geometry specified by this argument.

*SEND_ADDRESS*
A CM array in which the send addresses are stored.

On any CM platform, this array may be declared as INTEGER to support 4-byte send addresses, or as DOUBLE PRECISION or REAL*8 to support 8-byte send addresses. We recommend using DOUBLE PRECISION or REAL*8. See **DESCRIPTION** below for details.

*AXIS* An INTEGER specifying the axis number of the coordinates being deposited into the send address.

*COORDINATE*
A CM INTEGER array of the same shape and layout as *SEND_ADDRESS* and *MASK*. This array contains the grid coordinates to be incorporated into *SEND_ADDRESS*. These coordinates should be one based and not larger than the length of the axis of the specified *GEOMETRY*.

*MASK* A CM LOGICAL array of the same shape and layout as *SEND_ADDRESS* and *COORDINATE*, or the scalar value .TRUE.. If *MASK* is the scalar value .TRUE., all the elements of *SEND_ADDRESS* are modified. If *MASK* is a LOGICAL array, only the elements of *SEND_ADDRESS* corresponding to the elements of *MASK* that contain .TRUE. are modified.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine modifies send addresses stored in selected elements of *SEND_ADDRESS*, along the axis specified by *AXIS*, to incorporate the grid coordinates stored in the corresponding elements of *COORDINATE*. The *MASK* argument controls which elements are selected for the computation. The *SEND_ADDRESS* array should be initialized by calling `CMF_MAKE_SEND_ADDRESS` before calling `CMF_DEPOSIT_GRID_COORDINATE`. You can call `CMF_DEPOSIT_GRID_COORDINATE` repeatedly for each axis of the geometry without disturbing coordinates already deposited in *SEND_ADDRESS*.

*SEND_ADDRESS* can be declared as an `INTEGER`, or as a `DOUBLE PRECISION` or `REAL*8` CM array. The CM-2/200 computes send addresses as 4-byte values; the CM-5 uses 8-byte send addresses. Each platform will accept either 4-byte (`INTEGER`) or 8-byte (`DOUBLE PRECISION` or `REAL*8`) send address arrays. However, there may be a performance penalty for using 4-byte addresses on the CM-5, as the system coerces the values to 8-byte length. There is a minimal performance penalty for using 8-byte send-address arrays on the CM-2 (one array copy). Therefore, for maximum portability, all CM Fortran programs that compute send addresses should declare them as `DOUBLE PRECISION` or `REAL*8` values. `INTEGER` send address arrays should only be used in programs to be run on the CM-2 in which the marginally greater memory use is an issue.

### NOTES

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## SEE ALSO

```
CMF_GET_GEOMETRY_ID
CMF_MAKE_SEND_ADDRESS
CMF_MY_SEND_ADDRESS
CMF_SEND
```

## NAME

**CMF_DESCRIBE_ARRAY** – Prints information about a CM array to `stdout`.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_DESCRIBE_ARRAY (ARRAY)
```

## ARGUMENTS

*ARRAY* A CM array of any type.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine prints descriptive information about a CM array to `stdout`. This information includes the shape and layout of the array.

## SEE ALSO

```
CMF_GET_GEOMETRY_ID
CMF_SIZEOF_ARRAY_ELEMENT
```

## NAME

**CMF_FE_ARRAY_FROM_CM** – Transfers the contents of a CM array to a front-end array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FE_ARRAY_FROM_CM(DEST, SOURCE)
```

## ARGUMENTS

*DEST* A front-end array of the same type and shape as *SOURCE*. This array is the destination of the transfer.

*SOURCE*
A CM array of any type.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine transfers the contents of a CM array to a front-end array as quickly as possible. The two arrays should be of the same shape and type.

### NOTES

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## SEE ALSO

```
CMF_FE_ARRAY_TO_CM
```

## NAME

**CMF_FE_ARRAY_TO_CM** – Transfers the contents of a front-end array to a CM array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FE_ARRAY_TO_CM(DEST, SOURCE)
```

## ARGUMENTS

*DEST* A CM array of the same type and shape as *SOURCE*. This array is the destination of the transfer.

*SOURCE*
A front-end array of any type.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine transfers the contents of a front-end array to a CM array as quickly as possible. The two arrays should be of the same shape and type.

### NOTES

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## SEE ALSO

CMF_FE_ARRAY_FROM_CM

## NAME

**CMF_FILE_CLOSE** – Closes a CM file.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_CLOSE(UNIT, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable containing a valid unit number [1:29]. It may be a parameter or literal constant. The unit number is assigned by the user to a file when it is created with `CMF_OPEN_FILE` and has no relation to standard Fortran unit numbers.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

Closes a file in the CM file system.

## SEE ALSO

```
CMF_FILE_FDOPEN
CMF_FILE_LSEEK
CMF_FILE_LSEEK_FMS
CMF_FILE_OPEN
CMF_FILE_REWIND
CMF_FILE_TRUNCATE
CMF_FILE_UNLINK
```

## NAME

**CMF_FILE_FDOPEN** – Associates CM file or socket descriptor with a CM Fortran unit number. Both the descriptor and the unit number are input values.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_FDOPEN(CMFS_FD, UNIT, IOSTAT)
```

## ARGUMENTS

*CMFS_FD*
: `INTEGER`. A CMFS (CM file system) file or socket descriptor.

*UNIT*
: An `INTEGER` variable containing a valid unit number [1:29]. It may be a parameter or literal constant. This is the CM Fortran unit number to be associated with *CMFS_FD*. This unit number has no relation to standard Fortran unit numbers.

*IOSTAT*
: `INTEGER`. An integer variable into which the status of the I/O operation will be placed. A positive value indicates that the operation has succeeded; a negative value indicates that the operation has failed.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine associates the descriptor, *CMFS_FD*, of an open CMFS file or a CM socket with the CM Fortran unit number *UNIT*. You can then use *UNIT* as an argument to `CMF_CM_ARRAY_TO_FILE` or `CMF_CM_ARRAY_FROM_FILE` on the CM-5, or to `CMF_CM_ARRAY_TO_FILE_SO` or `CMF_CM_ARRAY_FROM_FILE_SO` on the CM-2, to perform I/O to CM-HIPPI, VME, or CM sockets from within a CM Fortran program.

## SEE ALSO

`CMF_FILE_GET_FD`

For more information on using the CM file system, see your CM I/O system documentation.

For more information on using the CM-HIPPI interface, see your CM-HIPPI documentation.

## NAME

**CMF_FILE_GET_FD** – Determines the CMFS file or socket descriptor previously associated with a specified CM Fortran unit.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_GET_FD(UNIT, CMFS_FD, IOSTAT)
```

## ARGUMENTS

*CMFS_FD*
An `INTEGER` output argument. The CMFS file or socket descriptor is returned in this variable.

*UNIT* An `INTEGER` variable containing a valid unit number [1:29]. It may be a parameter or literal constant. The unit number is assigned by the user to a file when it is created with `CMF_OPEN_FILE` and has no relation to standard Fortran unit numbers.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine returns, in the argument *CMFS_FD*, the CMFS (CM file system) file or socket descriptor, associated with the CM Fortran unit, *UNIT*. This allows you to determine the file descriptor previously associated with *UNIT*, for example with `CMF_FILE_FDOPEN`, so that the file descriptor can be used in calls to the low-level routines of the CMFS (CM File System) library.

## SEE ALSO

`CMF_FILE_FDOPEN`

For more information on using the CM file system, see your CM I/O system documentation.

## NAME

**CMF_FILE_LSEEK** – Offsets the file pointer a specified distance within a CM file.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_LSEEK(UNIT, OFFSET, IOSTAT)
```

## ARGUMENTS

| | |
|---|---|
| *UNIT* | An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This *UNIT* number has no relation to the standard Fortran unit number used in front-end I/O. |
| *OFFSET* | `INTEGER` An offset from the current position in the specified file. This argument is specified differently for serial order and parallel order files. See the **DESCRIPTION** and **EXAMPLE** sections below for more information. |
| *IOSTAT* | An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure. |

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine moves the file pointer within a CM file. If you do not reset the file pointer with `CMF_FILE_REWIND`, the offset is added to the current position of the file pointer. Before performing a seek operation on any CM file, you must perform a read or write operation on the file. This establishes the geometry of the file.

### Seeking in Serial Order Files

For serial order files (those written with `CMF_CM_ARRAY_TO_FILE_SO`), the offset is given in bytes. To calculate the offset, multiply the number of bytes in the array's element type by the number of elements to traverse. This allows you to seek to an arbitrary element in the file.

### Seeking in Parallel Order Files

For parallel order files (those written with `CMF_CM_ARRAY_TO_FILE`), you can only seek to the beginning of whole arrays. To compute the offset, you need not specify the size of the array, since this information is contained in the file geometry. You need specify only the size of an array's elements using the function CMF_SIZEOF_ARRAY_ELEMENT. To seek over multiple arrays, call the utility function CMF_SIZEOF_ARRAY_ELEMENT on each array and add the results. (See the example given below.)

Note that on the CM-5 only the element size of any later file operation must be the same as the element size of the read or write operation that established the geometry of the file when it was first opened.

### NOTE

If the file was written with CMF_CM_ARRAY_TO_FILE_FMS, you must use CMF_FILE_LSEEK_FMS to perform a seek operation on it.

## EXAMPLE

These examples illustrate the use of CMF_SIZEOF_ARRAY_ELEMENT to seek over parallel order files. For these examples, assume that a file associated with unit 29 has had three arrays written to it: A, B, and then C. Assume also that we have determined SIZEOF_A and SIZEOF_B by calling CMF_SIZEOF_ARRAY_ELEMENT on each array. Then, to position the file pointer to the beginning of array A, call

```
CALL CMF_FILE_REWIND(29, IOSTAT)
```

To position the file pointer to the beginning of array B, use:

```
CALL CMF_FILE_REWIND(29, IOSTAT)
CALL CMF_FILE_LSEEK(29, SIZEOF_A, IOSTAT)
```

To position the file pointer to the beginning of array C, use:

```
CALL CMF_FILE_REWIND(29, IOSTAT)
CALL CMF_FILE_LSEEK(29, SIZEOF_A+SIZEOF_B, IOSTAT)
```

To read arrays A and C:

```
CALL CMF_FILE_REWIND(29, IOSTAT)
CALL CMF_CM_ARRAY_FROM_FILE(29, DEST_ARRAY, IOSTAT)
CALL CMF_FILE_LSEEK(29, SIZEOF_B, IOSTAT)
CALL CMF_CM_ARRAY_FROM_FILE(29, DEST_ARRAY, IOSTAT)
```

## SEE ALSO

CMF_FILE_CLOSE
CMF_FILE_LSEEK_FMS
CMF_FILE_OPEN
CMF_FILE_REWIND
CMF_SIZEOF_ARRAY_ELEMENT
CMF_FILE_TRUNCATE
CMF_FILE_UNLINK

## NAME

**CMF_FILE_LSEEK_FMS** – Moves the file pointer a specified distance in a file written by `CMF_CM_ARRAY_TO_FILE_FMS`.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_LSEEK_FMS (UNIT, OFFSET, IOSTAT)
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29 inclusive. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This *UNIT* number has no relation to the standard Fortran unit number used in front-end I/O.

*OFFSET*
`INTEGER` An offset from the current position in the specified file. See the **DESCRIPTION** and **EXAMPLE** sections below for information on how to specify this argument.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine moves the file pointer to array boundaries within a CM file that has been written by `CMF_CM_ARRAY_TO_FILE_FMS`. See the man page for `CMF_CM_ARRAY_TO_FILE_FMS` for details on FMS procedures.

Before performing a seek operation on any CM file, you must perform a read or write operation on the file. This establishes the geometry of the file. Note that on the CM-5 only the element size of any later file operation must be the same as the element size of the read or write operation that established the geometry of the file when it was first opened.

To compute the offset, you need not specify the size of the array, since this information is contained in the file geometry. You need specify only the size of an array's elements using the function `CMF_SIZEOF_ARRAY_ELEMENT`. To seek over multiple arrays, call the utility function `CMF_SIZEOF_ARRAY_ELEMENT` on each array and add the results. (See the example given below.) If you do not reset the file pointer with `CMF_FILE_REWIND`, the offset is added to the current position of the file pointer.

## NOTE

If the file was written with CMF_CM_ARRAY_TO_FILE or CMF_CM_ARRAY_TO_FILE, use CMF_FILE_LSEEK to perform a seek operation on it.

## EXAMPLE

These examples illustrate the use of CMF_SIZEOF_ARRAY_ELEMENT to seek over FMSparallel order files. For these examples, assume that CMF_CM_ARRAY_TO_FILE_FMS has been used to write three arrays (A, B, and then C) to the file associated with unit 29. Assume also that we have determined SIZEOF_A and SIZEOF_B by calling CMF_SIZEOF_ARRAY_ELEMENT on each array. Then, to position the file pointer to the beginning of array A, call

```
CALL CMF_FILE_REWIND(29, IOSTAT)
```

To position the file pointer to the beginning of array B, use:

```
CALL CMF_FILE_REWIND(29, IOSTAT)
CALL CMF_FILE_LSEEK_FMS(29, SIZEOF_A, IOSTAT)
```

To position the file pointer to the beginning of array C, use:

```
CALL CMF_FILE_REWIND(29, IOSTAT)
CALL CMF_FILE_LSEEK_FMS(29, SIZEOF_A+SIZEOF_B, IOSTAT)
```

To read arrays A and C:

```
CALL CMF_FILE_REWIND(29, IOSTAT)
CALL CMF_CM_ARRAY_FROM_FILE_FMS(29, DEST_ARRAY, IOSTAT)
CALL CMF_FILE_LSEEK_FMS(29, SIZEOF_B, IOSTAT)
CALL CMF_CM_ARRAY_FROM_FILE_FMS(29, DEST_ARRAY, IOSTAT)
```

## SEE ALSO

```
CMF_FILE_CLOSE
CMF_FILE_LSEEK_FMS
CMF_FILE_OPEN
CMF_FILE_REWIND
CMF_SIZEOF_ARRAY_ELEMENT
CMF_FILE_TRUNCATE
CMF_FILE_UNLINK
```

## NAME

**CMF_FILE_OPEN** – Opens a CM file and attaches the file to the `UNIT`.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_OPEN( UNIT, PATH, IOSTAT)
```

## ARGUMENTS

| | |
|---|---|
| *UNIT* | An `INTEGER` variable containing a valid unit number [1:29]. It may be a parameter or literal constant. |
| *PATH* | A `CHARACTER` string containing the pathname for the file to be opened. |
| *IOSTAT* | An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure. |

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine opens a CM file and attaches the file to the *UNIT*. You must supply this unit number to other CM file system procedures when you wish to operate on this file. Note that the CM file system unit numbers have no relation to standard Fortran unit numbers.

## SEE ALSO

```
CMF_FILE_FDOPEN
CMF_FILE_CLOSE
CMF_FILE_LSEEK
CMF_FILE_LSEEK_FMS
CMF_FILE_REWIND
CMF_FILE_TRUNCATE
CMF_FILE_UNLINK
```

## NAME

**CMF_FILE_REWIND** – Moves a file pointer to the beginning of a CM file.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_REWIND (UNIT, IOSTAT)
```

## ARGUMENTS

| | |
|---|---|
| *UNIT* | An INTEGER variable containing a valid unit number [1:29]. It may be a parameter or literal constant. The unit number is assigned by the user to a file when it is created with CMF_OPEN_FILE and has no relation to standard Fortran unit numbers. |
| *IOSTAT* | An INTEGER variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure. |

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine moves the file pointer for the CM file associated with the *UNIT* number to the beginning of that file. You can reset the file pointer before setting it to a specific location in the file with CMF_FILE_LSEEK or CMF_FILE_LSEEK_FMS. See the man page for CMF_FILE_LSEEK for more information.

### NOTE

Before calling CMF_FILE_REWIND on a newly opened file, you must first perform a read or write on the file.

## SEE ALSO

CMF_FILE_CLOSE
CMF_FILE_LSEEK
CMF_FILE_LSEEK_FMS
CMF_FILE_OPEN
CMF_FILE_TRUNCATE
CMF_FILE_UNLINK

## NAME

**CMF_FILE_TRUNCATE** – Change the size of a CM file.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_TRUNCATE ( UNIT, LENGTH, IOSTAT )
```

## ARGUMENTS

*UNIT* An `INTEGER` variable, parameter, or literal constant containing a valid unit number in the range 1 to 29. The unit number is specified in the call to `CMF_FILE_OPEN` that creates the file in the CM file system. This *UNIT* number has no relation to the standard Fortran unit number used in front-end I/O.

*LENGTH*
An `INTEGER` specifying the new length of the file. This argument is specified differently for serial order and parallel order files. See the **DESCRIPTION** section below for more information.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine changes the size of the specified CM file to *LENGTH*. The file is extended or shortened by the difference between the *LENGTH* argument and the file's current length. If the file was previously larger than *LENGTH*, the extra data is lost. If the file was previously smaller than *LENGTH*, the file is extended to *LENGTH*.

### Deriving *LENGTH* for Serial Order Files

For serial order files (those written with `CMF_CM_ARRAY_TO_FILE_SO`), the length is given in bytes. To calculate the length for the file, multiply the number of bytes in the array's element type by the number of array elements to be contained in the file.

### Deriving *LENGTH* for Parallel Order Files

For parallel order files (those written with `CMF_CM_ARRAY_TO_FILE` or `CMF_CM_ARRAY_TO_FILE_SO`), the file can only be reduced or enlarged by whole arrays. To compute the length of the array, you need not specify the size of the array, since this information is contained in the file geometry. You need specify only the size of an array's elements using the function `CMF_SIZEOF_ARRAY_ELEMENT`. You can increase or decrease the size of the file by more than one array by calling `CMF_SIZEOF_ARRAY_ELEMENT` on several arrays in succession, adding the returned values together, and supplying the cumulative result as the *LENGTH* argument.

### NOTES

`CMF_FILE_TRUNCATE` requires the file to be open for writing.

Before calling `CMF_FILE_TRUNCATE` on a newly opened file, you must first perform a read or write on the file.

## SEE ALSO

```
CMF_FILE_CLOSE
CMF_FILE_LSEEK
CMF_FILE_LSEEK_FMS
CMF_FILE_OPEN
CMF_FILE_REWIND
CMF_SIZEOF_ARRAY_ELEMENT
CMF_FILE_UNLINK
```

## NAME

**CMF_FILE_UNLINK** – Removes a file from a directory.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_UNLINK(PATH, IOSTAT)
```

## ARGUMENTS

*PATH* A `CHARACTER` string containing the pathname of the file to be removed.

*IOSTAT* An `INTEGER` variable into which the status of the I/O operation will be placed. A positive value indicates success and a negative value indicates failure.

## RETURNED VALUE

None.

## DESCRIPTION

`CMF_FILE_UNLINK` removes the entry for the file *PATH* from the file's directory. If this entry was the last link to the file and no process has the file open, then the file is deleted and all resources associated with the file are reclaimed. If, however, the file was open in any process, the actual resource reclamation is delayed until the file is closed, even though the directory entry has disappeared.

## SEE ALSO

```
CMF_FILE_CLOSE
CMF_FILE_LSEEK
CMF_FILE_LSEEK_FMS
CMF_FILE_OPEN
CMF_FILE_REWIND
CMF_FILE_TRUNCATE
```

## NAME

**CMF_GET_GEOMETRY_ID** – Returns a geometry identifier for a CM array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

GEOM_ID = CMF_GET_GEOMETRY_ID (ARRAY)
```

## ARGUMENTS

*ARRAY* A CM array of any type.

## RETURNED VALUE

An `INTEGER` identifying the geometry of *ARRAY*. This identifier should only be passed to other CM Fortran library procedures that accept geometry identifiers, such as `CMF_DEPOSIT_GRID_COORDINATE`.

## DESCRIPTION

This function returns an identifier for a geometry object that defines the shape and layout of *ARRAY* on the CM. This identifier is required by `CMF_DEPOSIT_GRID_COORDINATE`. Note that you cannot access the array information directly from this identifier. It can only be passed to other procedures. Information about an array can be displayed by calling `CMF_DESCRIBE_ARRAY`.

## SEE ALSO

```
CMF_DEPOSIT_GRID_COORDINATE.
CMF_DESCRIBE_ARRAY
CMF_SIZEOF_ARRAY_ELEMENT
```

## NAME

**CMF_LOOKUP_IN_TABLE** – Performs parallel array reference on a lookup table.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_LOOKUP_IN_TABLE(DEST, TABLE, INDEX, MASK)
```

## ARGUMENTS

*DEST* A CM array. The destination array. The values retrieved from the table are stored into this array.

*TABLE* Integer. The identifier for the lookup table to be referenced as returned by `CMF_ALLOCATE_TABLE`. Only lookup tables allocated by the `CMF_ALLOCATE_TABLE` subroutine can be referenced by this procedure.

*INDEX* An `INTEGER` CM array containing the indices to be used to reference *TABLE*. The indices must have a lower bound of 1.

*MASK* A CM `LOGICAL` array or the scalar value `.TRUE.`. If *MASK* is the scalar value `.TRUE.`, all the elements of *DEST* are modified. If *MASK* is a `LOGICAL` array, only the elements of *DEST* corresponding to the elements of *MASK* that contain `.TRUE.` are modified.

## RETURNED VALUE

None.

## DESCRIPTION

`CMF_LOOKUP_IN_TABLE` performs a parallel array reference on *TABLE*.

*DEST*, *INDEX*, and *MASK* (if an array) must be conformable parallel arrays. Each element of *INDEX* is used as an index into *TABLE*, and the value retrieved from that location in *TABLE* is stored into the corresponding element of *DEST*.

Using `CMF_ALLOCATE_TABLE` and `CMF_LOOKUP_IN_TABLE` to perform indirect indexing is significantly faster that using a conventionally allocated table when:

- The content of the table never or rarely changes.
- The table is relatively small. Specifically, it must consume less memory than is available on a processing element (vector unit, node, or processor, depending on the execution model). The function `CMF_AVAILABLE_MEMORY` returns the amount of memory left in each processing element in units of bytes.

When these constraints are met, `CMF_LOOKUP_IN_TABLE` stores a copy of the table into each processing element. This allows the subroutine to do local memory references into the local copy of the table using indirect addressing hardware.

## NOTES

`CMF_LOOKUP_IN_TABLE` is substantially faster when

- the *MASK* argument has a value of `.TRUE.`
- the product of the dimensions of *INDEX* is an integer multiple of the number of nodes or processors available to the program. The number of processing elements is returned by the function `CMF_NUMBER_OF_PROCESSORS`.

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## EXAMPLE

The code below using `CMF_LOOKUP_IN_TABLE` is significantly faster than the following code when the constraints on *TABLE* are met.

```
C Conventional Array Referencing
C
      REAL DEST(8192), TABLE(100)
      INTEGER INDEX(8192)
      DEST = TABLE(INDEX)
C
C Faster Array Referencing Using CMF_LOOKUP_IN_TABLE
C
      REAL DEST(8192), TABLE_VALUES(100)
      INTEGER TABLE
      INTEGER INDEX(8192)
C
      TABLE = CMF_ALLOCATE_TABLE(CMF_FLOAT, 100, TABLE_VALUES)
      CALL CMF_LOOKUP_IN_TABLE(DEST, TABLE, INDEX, .TRUE.)
C
      CALL CMF_DEALLOCATE_TABLE(TABLE)
```

## SEE ALSO

```
CMF_ALLOCATE_TABLE
CMF_AVAILABLE_MEMORY
CMF_DEALLOCATE_TABLE
```

## NAME

**CMF_MAKE_SEND_ADDRESS** – Initializes a send address.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_MAKE_SEND_ADDRESS (ARRAY)
```

## ARGUMENTS

*ARRAY* A CM array. On any CM platform, this array may be declared as `INTEGER` to support 4-byte send addresses, or as `DOUBLE PRECISION` or `REAL*8` to support 8-byte send addresses. We recommend using `DOUBLE PRECISION` or `REAL*8`. See **DESCRIPTION** below for details.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine initializes *ARRAY* with NULL send addresses. This should be done before calling `CMF_DEPOSIT_GRID_COORDINATE`. *ARRAY* can be an `INTEGER` CM array, or a `DOUBLE PRECISION` or `REAL*8` CM array.

The CM-2/200 computes send addresses as 4-byte values; the CM-5 uses 8-byte send addresses. Each platform will accept either 4-byte (`INTEGER`) or 8-byte (`DOUBLE PRECISION` or `REAL*8`) send address arrays. However, there may be a performance penalty for using 4-byte addresses on the CM-5, as the system coerces the values to 8-byte length. There is a minimal performance penalty for using 8-byte send-address arrays on the CM-2 (one array copy). Therefore, for maximum portability, all CM Fortran programs that compute send addresses should declare them as `DOUBLE PRECISION` or `REAL*8` values. `INTEGER` send address arrays should only be used in programs to be run on the CM-2 in which the marginally greater memory use is an issue.

### NOTES

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the array has a lower bound of 1. All other lower bound values are ignored.

**SEE ALSO**

```
CMF_MY_SEND_ADDRESS
CMF_DEPOSIT_GRID_COORDINATE
CMF_SEND
```

## NAME

**CMF_MY_SEND_ADDRESS** – Calculates the send address of each element in an array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_MY_SEND_ADDRESS(ARRAY)
```

## ARGUMENTS

*ARRAY* A CM array. Each element of this array is filled with its own send address.

On any CM platform, this array can be declared as `INTEGER` to support 4-byte send addresses, or as `DOUBLE PRECISION` or `REAL*8` to support 8-byte send addresses. We recommend using `DOUBLE PRECISION` or `REAL*8`. See **DESCRIPTION** below for details.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine calculates the send address for each element of *ARRAY* and fills each element with its own send address.

The CM-2/200 computes send addresses as 4-byte values; the CM-5 uses 8-byte send addresses. Each platform will accept either 4-byte (`INTEGER`) or 8-byte (`DOUBLE PRECISION` or `REAL*8`) send address arrays. However, there may be a performance penalty for using 4-byte addresses on the CM-5, as the system coerces the values to 8-byte length. There is a minimal performance penalty for using 8-byte send-address arrays on the CM-2 (one array copy). Therefore, for maximum portability, most CM Fortran programs that compute send addresses should declare them as `DOUBLE PRECISION` or `REAL*8` values. `INTEGER` send address arrays should only be used in programs to be run on the CM-2 in which the marginallly greater memory use is an issue.

### NOTE

The CM Fortran Utility Library procedures will not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_MAKE_SEND_ADDRESS
CMF_DEPOSIT_GRID_COORDINATE
CMF_SEND
```

## NAME

**CMF_NUMBER_OF_PROCESSORS** – Returns the number of vector units, nodes, or processors currently available to the program.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'
```

*NUM* = CMF_NUMBER_OF_PROCESSORS ( )

## ARGUMENTS

None.

## RETURNED VALUE

```
INTEGER
```

## DESCRIPTION

The meaning of the value returned by CMF_NUMBER_OF_PROCESSORS varies with the CM architecture and the exectution model under which the program is running. The machine and execution model can be determined with the CM Fortran utility CMF_ARCHITECTURE. The following table summarizes the meaning of the return value of CMF_NUMBER_OF_PROCESSORS for each value returned by CMF_ARCHITECTURE:

| CMF_ARCHITECTURE<br>Return Values | CMF_NUMBER_OF_PROCESSORS<br>Return Values |
|---|---|
| CMF_CM5_SPARC | number of processing nodes |
| CMF_CM5_VU | number of vector units |
| CMF_CM200_SLICEWISE | number of processing nodes |
| CMF_CM2_SLICEWISE | number of processing nodes |
| CMF_CM200_PARIS | number of bit-serial processors |
| CMF_CM2_PARIS | number of bit-serial processors |
| CMF_SIM (CM Fortran simulator) | 1 |

## SEE ALSO

CMF_ARCHITECTURE

## NAME

**CMF_ORDER** – places the numerical rank of each element of a source array in the corresponding element of the destination array.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_ORDER(DEST, SOURCE, AXIS, MASK)
```

## ARGUMENTS

*DEST* A CM `INTEGER` array.

*SOURCE*
A CM array of any type. The order of this array is stored in *DEST*.

*AXIS* Integer. The axis over which to do the ordering.

*MASK* A CM `LOGICAL` array conforming to *DEST*, or the scalar value `.TRUE.`. If *MASK* is the scalar value .TRUE., all the elements of *DEST* are modified. If *MASK* is a `LOGICAL` array, only the elements of *DEST* corresponding to the elements of *MASK* that contain `.TRUE.` are modified.

## RETURNED VALUE

None.

## DESCRIPTION

For each element of *SOURCE* with a *MASK* value of `.TRUE.`, `CMF_ORDER` places the numerical rank of that element in the corresponding element of *DEST*. Each row along the specified *AXIS* is treated as a separate set of values to be ordered. The rank values computed by this subroutine will always be 1 to N inclusive, where N is the number of items in each set of values to be ordered. This is true regardless of the lower bound of *SOURCE*.

### NOTES

The CM Fortran Utility Library procedures do not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## NAME

**CMF_RANDOM** – Places a different pseudo random number in each element of an array *DEST*.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_RANDOM(DEST, LIMIT)
```

## ARGUMENTS

*DEST* A CM array of one of the following types:

- INTEGER
- REAL
- DOUBLE PRECISION
- COMPLEX
- DOUBLE COMPLEX

*LIMIT* An INTEGER (*4 only) specifying the exclusive upper bound for the range of random numbers generated. For floating-point values this number should be 1.0. For INTEGERs only, to specify no upper bound, *LIMIT* should be 0.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine places a pseudo random number in each element of the *DEST* array.

The random number generator algorithm used by this procedure is Wolfram's Rule 30 Cellular Automaton. For INTEGER data the random numbers are generated by simply running the automaton for 32 generations. For REAL, DOUBLE PRECISION, COMPLEX, and DOUBLE COMPLEX types, the random numbers are generated by running the automaton for *s* generations (where *s* is the mantissa length), and setting the exponent bits and sign bit so that the result is uniformly distributed between 2.0 and 1.0. Then 1.0 is subtracted from the result to yield a number that is uniformly distributed between 0.0 and 1.0. This automaton is run on a finite string of bits, i=0,...,N-1, with periodic boundary conditions (so that site N is equivalent to site 0). In the CM implementation N = 59.

The primary reference for the Rule 30 Cellular Automaton is Stephen Wolfram, "Random Sequence Generation by Cellular Automata," *Advances in Applied Mathematics* 7, pp. 123-69 (1986). This paper may be more readily available as a reprint in Stephen Wolfram, *Theory and Application of Cellular Automata* (including selected papers 1983-1986), World Scientific (1986).

**NOTE**

The CM Fortran Utility Library procedures do not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

## SEE ALSO

```
CMF_RANDOMIZE
```

## NAME

**CMF_RANDOMIZE** – Initializes the random number generator with a seed.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_RANDOMIZE(SEED)
```

## ARGUMENTS

*SEED* An `INTEGER` scalar specifying the seed value with which to initialize the random number generator.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine uses *SEED* to initialize the random number generator used when `CMF_RANDOM` is called.

The random number generator algorithm used by `CMF_RANDOM` is Wolfram's Rule 30 Cellular Automaton. For more information see the man page for `CMF_RANDOM`.

## SEE ALSO

`CMF_RANDOM`

## NAME

**CMF_RANK** – Places the numerical rank of each selected element along an array axis, or axis segment, into the corresponding element of the destination array.

## SYNTAX

```
 INCLUDE '/usr/include/cm/CMF_defs.h'

 CALL CMF_RANK(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, SEGMENT_MODE,
&              MASK)
```

## ARGUMENTS

*DEST* A CM INTEGER array. The destination array. The *DEST* and *SOURCE* arrays must be of the same shape and layout.

*SOURCE*
A CM array of any type. The source array. The *SOURCE* and *DEST* arrays must be of the same shape and layout.

*SEGMENT*
A LOGICAL CM array of the same shape and layout as *DEST*, *SOURCE*, and *MASK*. .TRUE. values in the *SEGMENT* array are used as segment delimiters for the corresponding elements of the *SOURCE* array.

If *SEGMENT_MODE* has a value of CMF_NONE, then this argument is ignored and may be CMF_NULL. Otherwise, each segment is ranked independently. The arguments *SEGMENT_MODE*, *DIRECTION*, and *MASK* control the way the ranking proceeds over the segments. See the **DESCRIPTION** section below for details.

*AXIS* An integer. The axis of *SOURCE* to be ranked.

*DIRECTION*
An integer. The value can be CMF_UPWARD or CMF_DOWNWARD. If the value is CMF_UPWARD, the values are ranked from the smallest value to the largest; rank 1 is assigned to the smallest value. If the value is CMF_DOWNWARD, the values are ranked from largest value to the smallest; rank 1 is assigned to the largest value.

*SEGMENT_MODE*
An INTEGER. One of the following integer values: CMF_NONE, CMF_SEGMENT_BIT or CMF_START_BIT. This argument controls how the segments of *SOURCE* defined by the *SEGMENT* array are interpreted. See **DESCRIPTION** below for more information.

*MASK* A CM LOGICAL array, or the scalar value .TRUE..

If the value of *MASK* is a scalar .TRUE., all the values of *SOURCE* will be included in the ranking.

If *MASK* is a logical array, it must be of the same shape and layout as *DEST*, *SOURCE*, and *SEGMENT*. The values in *SOURCE* corresponding to values of `.FALSE.` in *MASK* are not included in the ranking.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine determines the numerical ranking of the values stored in the selected elements along the specified axis of the *SOURCE* array and places the rank of each element in the corresponding element of the destination array *DEST*. Selected elements are those that correspond to a `.TRUE.` element in the *MASK* array. The rankings computed by this subroutine are always 1 to *N* inclusive, where *N* is the number of elements in the set of values to be ordered. Array elements that correspond to a `.FALSE.` value in the *MASK* argument are not included in the ranking and the corresponding element of *DEST* is not changed.

The rank is always stable; for each pair of elements that contain equal values, the element with the lower grid coordinate along the ranking axis is assigned the lower numbered rank, regardless of the direction of the ranking.

In addition, the array elements along the axis may be partitioned into distinct sets, called *segments*, through the use of the *SEGMENT* and *SEGMENT_MODE* arguments. Each segment along the specified *AXIS* is treated as a separate set of values to be ordered. Each element of *SEGMENT* that contains `.TRUE.`, marks the corresponding element of *SOURCE* as a segment boundary (the start or end of a segment). Segments begin (or end) with each element in which the value of *SEGMENT* is `.TRUE.`, and continue up (or down) the axis through all elements where the value of *SEGMENT* is `.FALSE.`. The effect of these boundaries depends on the value of *SEGMENT_MODE*.

If *SEGMENT_MODE* is `CMF_NONE`, the elements are ranked along the entire length of the array axis and the values in *SEGMENT* have no effect.

If *SEGMENT_MODE* is `CMF_SEGMENT_BIT`, then:

- The *MASK* argument does not affect the use of the *SEGMENT* array. That is, elements containing `.TRUE.` in the *SEGMENT* array create a segment boundary even if the corresponding value of *MASK* is `.FALSE.` (The *MASK* array still selects the elements of *SOURCE* to be included as described above.)
- A *SEGMENT* value of `.TRUE.` indicates the start of a segment for both upward and downward ranks.

If the value is `CMF_START_BIT`, then:

- The *MASK* argument applies to the *SEGMENT* array as well as to the *SOURCE* array. That is, elements containing `.TRUE.` in *SEGMENT* array create a segment boundary only if the corresponding element of *MASK* is also `.TRUE.`.

- A *SEGMENT* value of `.TRUE.` indicates the *start* of a segment for upward ranks, but the *end* of a segment for downward ranks. That is, the *SOURCE* element corresponding to a `.TRUE.` *SEGMENT* element is the first element in a segment for an upward rank, but the last element in a segment for a downward rank. In downward ranks, the new segment begins with the first unmasked element following the segment boundary.

**NOTES**

The CM Fortran Utility Library procedures do not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## EXAMPLES

**Upward Rank**

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0],
and SEGMENT   = [T,   F,   F,   F  ],
then DEST     = [1,   4,   3,   2  ].
```

**Downward Rank**

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   F,   F  ],
then DEST     = [4,   1,   2,   3  ].
```

**Upward Rank With Mask**

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   F,   F  ],
and MASK      = [T,   T,   F,   T  ],
then DEST     = [1,   3,   X,   2  ].
```

**Segmented Upward Rank**

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   T,   F  ],
and MASK      = [T,   T,   T,   T  ],
then DEST     = [1,   2,   4,   3  ].
```

**Segmented Upward Rank With Context**

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   T,   F  ],
and MASK      = [F,   T,   T,   T  ],
then DEST     = [X,   1,   3,   2  ].
```

Note that, while the ranking is determined within each segment, the rank indices are numbered continuously across the entire axis. In this example, the ranking stored in `DEST` is `[X,  1,  3,  2]` as illustrated, not `[X,  1,  2,  1]`. That is, the ranking starts anew in each segment, but the numbering of the indices associated with each element is not restarted. Each element receives a unique ranking index.

## NAME

**CMF_SCAN_[ADD,_MAX,_MIN,_COPY,_IOR,_IAND,_IEOR]** – Performs a scan along an axis on the selected elements of the source array, optionally within segments.

## SYNTAX

```
 INCLUDE '/usr/include/cm/CMF_defs.h'

 CALL CMF_SCAN_ADD(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                  SEGMENT_MODE, MASK)

 CALL CMF_SCAN_MAX(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                  SEGMENT_MODE, MASK)

 CALL CMF_SCAN_MIN(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                  SEGMENT_MODE, MASK)

  CALL CMF_SCAN_COPY(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                   SEGMENT_MODE, MASK)

 CALL CMF_SCAN_IOR(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                  SEGMENT_MODE, MASK)

 CALL CMF_SCAN_IAND(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                   SEGMENT_MODE, MASK)

 CALL CMF_SCAN_IEOR(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, INCLUSION,
&                   SEGMENT_MODE, MASK)
```

## ARGUMENTS

*DEST* A CM array. The destination array. The *DEST* and *SOURCE* arrays must be of the same type, shape, and layout.

*SOURCE*

A CM array. The source array. The *SOURCE* and *DEST* arrays must be of the same type, shape, and layout.

*SEGMENT*

A LOGICAL CM array of the same shape and layout as *DEST*, *SOURCE*, and *MASK*. .TRUE. values in the *SEGMENT* array are used as segment delimiters for the corresponding elements of the *SOURCE* array.

If *SEGMENT_MODE* has a value of CMF_NONE, then this argument is ignored and may be CMF_NULL. Otherwise, the scan operation is performed independently for each segment of *SOURCE* defined by *SEGMENT*. The arguments *SEGMENT_MODE*, *DIRECTION*, *INCLUSION*, and *MASK* control the way the scan proceeds over the segments. See the **DESCRIPTION** section below for details.

*AXIS* An integer. The axis of *SOURCE* along which the scan is performed.

*DIRECTION*
An integer. The value can be `CMF_UPWARD` or `CMF_DOWNWARD`. If the value is `CMF_UPWARD`, the values are combined from the lower numbered elements toward the higher. If the value is `CMF_DOWNWARD`, the values are combined from higher numbered elements toward the lower.

*INCLUSION*
An integer. The value can be `CMF_EXCLUSIVE` or `CMF_INCLUSIVE`. If the value is `CMF_EXCLUSIVE` the first element in each *SOURCE* segment (as defined by the `.TRUE.` elements of *SEGMENT*) is not included in the computation. If the value is `CMF_INCLUSIVE`, the first value in each segment is included.

*SEGMENT_MODE*
An `INTEGER`. One of the following integer values: `CMF_NONE`, `CMF_SEGMENT_BIT` or `CMF_START_BIT`. This argument controls how the segments of *SOURCE* defined by the *SEGMENT* array are interpreted for the scan operation. See **DESCRIPTION** below for more information.

*MASK* A CM `LOGICAL` array, or the scalar value `.TRUE.`.

If the value of *MASK* is a scalar `.TRUE.`, all the values of *SOURCE* will be included in the computation.

If *MASK* is a logical array, it must be of the same shape and layout as *DEST*, *SOURCE*, and *SEGMENT*. The values in *SOURCE* corresponding to values of `.FALSE.` in *MASK* are not included in the computation.

## RETURNED VALUE

None.

## DESCRIPTION

Each subroutine in this group performs a scan operation along an axis of the source array on the selected elements and puts the results in the destination array. Optionally, you may specify scan segments for the source array so that the scan operation is performed independently on distinct sections of the array axis.

Each of these subroutines cumulatively applies a binary operator over the selected elements of one axis of the source array *SOURCE*. Selected elements are those that correspond to a `.TRUE.` element in the *MASK* array. The scan operation combines each selected element of the array with the cumulative result from all the selected elements that precede it. The result for each of these elements is stored in the corresponding element of the destination array *DEST*. Array elements that correspond to a `.FALSE.` value in the *MASK* argument are excluded from the computation and the corresponding element of *DEST* is not changed.

In addition, the array elements along the axis may be partitioned into distinct sets, called *segments*, through the use of the *SEGMENT* and *SEGMENT_MODE* arguments. Each element of *SEGMENT* that contains `.TRUE.`, marks the corresponding element of *SOURCE* as a segment boundary (the start or end of segment). Segments begin (or end) with each element in which the value of *SEGMENT* is `.TRUE.`, and continue up (or down) the axis through all elements where the value of *SEGMENT* is `.FALSE.`. The effect of these boundaries depends on the value of *SEGMENT_MODE.*

If *SEGMENT_MODE* is `CMF_NONE`, the operation specified by the subroutine proceeds along the entire length of the array axis and the values in *SEGMENT* have no effect.

If *SEGMENT_MODE* is `CMF_SEGMENT_BIT`, then:

- The *MASK* argument does not affect the use of the *SEGMENT* array. That is, elements containing `.TRUE.` in the *SEGMENT* array create a segment boundary even if the corresponding value of *MASK* is `.FALSE.` (The *MASK* array still selects the elements of *SOURCE* to be included as described above.)
- A *SEGMENT* value of `.TRUE.` indicates the start of a segment for both upward and downward scans.
- When the *INCLUSION* argument is `CMF_EXCLUSIVE`, the first *DEST* element in each segment, is set to zero. (There is no scan result value for this element because in exclusive mode the first element of each segment of *SOURCE* is excluded from the scan).

If *SEGMENT_MODE* is `CMF_START_BIT`, then:

- The *MASK* argument applies to the *SEGMENT* array as well as to the *SOURCE* array. That is, elements containing `.TRUE.` in *SEGMENT* array create a segment boundary only if the corresponding element of *MASK* is also `.TRUE.`.
- A *SEGMENT* value of `.TRUE.` indicates the *start* of a segment for upward scans, but the *end* of a segment for downward scans. That is, the *SOURCE* element corresponding to a `.TRUE.` *SEGMENT* element is the first element in a segment for an upward scan, but the last element in a segment for a downward scan. In downward scans, the new segment begins with the first unmasked element following the segment boundary.
- When the *INCLUSION* argument is `CMF_EXCLUSIVE`, the first *DEST* element in each segment (which is set to zero in `CMF_SEGMENT_BIT` scans) is used to store the final scan result of the preceding segment. Note that this result value does not contribute to the scan result for the segment in which it is stored.

See the example below for an illustration of how these arguments interact. Information on each of the individual scan routines follows.

### CMF_SCAN_ADD

The subroutine CMF_SCAN_ADD can operate on numbers of the following types:

- INTEGER
- REAL
- DOUBLE PRECISION (real)
- COMPLEX
- DOUBLE COMPLEX (double-precision complex)

### CMF_SCAN_MAX, CMF_SCAN_MIN

The subroutines CMF_SCAN_MAX and CMF_SCAN_MAX can operate on numbers of the following types:

- INTEGER
- REAL
- DOUBLE PRECISION (real)

### CMF_SCAN_IOR, CMF_SCAN_IAND, CMF_SCAN_IEOR

The subroutines CMF_SCAN_IOR, CMF_SCAN_IAND, and CMF_SCAN_IEOR, can operate on the following types:

- LOGICAL
- INTEGER

The operations IOR, IAND, and IEOR, correspond to logical inclusive OR, logical AND, and logical exclusive OR, respectively.

For INTEGERs, these subroutines do the operation on a bitwise basis.

### CMF_SCAN_COPY

The subroutine CMF_SCAN_COPY operates on all types. The binary operator used by this routine always returns its first argument. This subroutine is usually used to copy the first element in a segment to all the other elements of that segment.

Here is an example for CMF_SCAN_COPY:

```
 CMF_SCAN_COPYDEST, SOURCE, SEGMENT, 1, CMF_UPWARD, CMF_INCLUSIVE,
&             CMF_SEGMENT_BIT, .TRUE.)
```

```
If SOURCE      = [1,2,3,4,5,6,7,8,9],
and SEGMENT    = [T,F,F,F,T,F,F,F,F],
then DEST      = [1,1,1,1,5,5,5,5,5].
```

### NOTE

The CM Fortran Utility Library procedures do not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## EXAMPLE

The table below shows the results for a single row along the axis being "scanned" by the subroutine `CMF_SCAN_ADD`. The *SOURCE* argument is an integer array filled with the value 1. The *MASK* and *SEGMENT* arguments are logical arrays with the values indicated at the top of the table (where T stands for `.TRUE.` and F stands for `.FALSE.`). The argument *DIRECTION* can be `CMF_UPWARD` or `CMF_DOWNWARD`. The argument *INCLUSION* can be `CMF_EXCLUSIVE` or `CMF_INCLUSIVE`. The argument *SEGMENT_MODE* can be `CMF_NONE`, `CMF_SEGMENT_BIT`, or `CMF_START_BIT`. *DEST* elements that are masked (the elements marked with dots " . " in the table) are not changed by this operation.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MASK | T | T | T | T | F | F | F | F | T | T | F | F | T | T | T | F |
| SEGMENT | F | F | T | F | F | F | T | F | F | F | F | F | F | T | F | F |
| SOURCE | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| DIRECTION | INCLUSION | SEGMENT-MODE | DEST |
|---|---|---|---|
| upward | exclusive | none | 0 1 2 3 . . . . 4 5 . . 6 7 8 . |
| downward | exclusive | none | 8 7 6 5 . . . . 4 3 . . 2 1 0 . |
| upward | inclusive | none | 1 2 3 4 . . . . 5 6 . . 7 8 9 . |
| downward | inclusive | none | 9 8 7 6 . . . . 5 4 . . 3 2 1 . |
| upward | exclusive | segment | 0 1 0 1 . . . . 0 1 . . 2 0 1 . |
| downward | exclusive | segment | 1 0 1 0 . . . . 2 1 . . 0 1 0 . |
| upward | inclusive | segment | 1 2 1 2 . . . . 1 2 . . 3 1 2 . |
| downward | inclusive | segment | 2 1 2 1 . . . . 3 2 . . 1 2 1 . |
| upward | exclusive | start | 0 1 2 1 . . . . 2 3 . . 4 5 1 . |
| downward | exclusive | start | 2 1 5 4 . . . . 3 2 . . 1 1 0 . |
| upward | inclusive | start | 1 2 1 2 . . . . 3 4 . . 5 1 2 . |
| downward | inclusive | start | 3 2 1 5 . . . . 4 3 . . 2 1 1 . |

## NAME

**CMF_SEND_[OVERWRITE,_MAX,_MIN,_ADD,_IOR,_IAND,_IEOR]** - Sends elements from *SOURCE* to *DEST* according to the addresses in *SEND_ADDRESS.* Combines multiple values sent to the same *DEST* element using the operation specified in the name of the send function.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_SEND_OVERWRITE(DEST,SEND_ADDRESS, SOURCE, MASK)

CALL CMF_SEND_MAX(DEST,SEND_ADDRESS, SOURCE, MASK)

CALL CMF_SEND_MIN(DEST, SEND_ADDRESS, SOURCE, MASK)

CALL CMF_SEND_ADD(DEST, SEND_ADDRESS, SOURCE, MASK)

CALL CMF_SEND_IOR(DEST, SEND_ADDRESS, SOURCE, MASK)

CALL CMF_SEND_IAND(DEST, SEND_ADDRESS, SOURCE, MASK)

CALL CMF_SEND_IEOR(DEST, SEND_ADDRESS, SOURCE, MASK)
```

## ARGUMENTS

*DEST* A CM array. The destination array. The data types allowed for each type of combiner are listed below.

*SEND_ADDRESS*
CM array. The send addresses used to determine where in *DEST* each element of *SOURCE* is sent. Send addresses are constructed using the CM Fortran Utility Library procedures CMF_MAKE_SEND_ADDRESS, CMF_MY_SEND_ADDRESS, and CMF_DEPOSIT_GRID_COORDINATES.

On any CM platform, this array may be declared as INTEGER to support 4-byte send addresses, or as DOUBLE PRECISION or REAL*8 to support 8-byte send addresses. We recommend using DOUBLE PRECISION or REAL*8. See **DESCRIPTION** below for details.

*SOURCE*
A CM array. The source array. This array must have same shape and layout as *SEND_ADDRESS.* The data types allowed for each type of combiner are listed below.

*MASK* A CM LOGICAL array or the scalar value .TRUE.. If *MASK* is a logical array, it must have the same shape and layout as *SEND_ADDRESS* and only those elements of *SOURCE* that correspond to .TRUE. values in *MASK* are sent to *DEST.* If *MASK* is the scalar value .TRUE., all elements of *SOURCE* are sent.

## RETURNED VALUE

None.

## DESCRIPTION

Each selected element of *SOURCE* is sent to the element of *DEST* specified by the send-address in the corresponding element of *SEND_ADDRESS*. If multiple elements of *SEND_ADDRESS* have the same value, the corresponding elements of *SOURCE* are combined together. The *MASK* argument controls which elements of *SOURCE* are selected for the computation.

The *SEND_ADDRESS* array may be declared as `INTEGER`, or as `DOUBLE PRECISION` or `REAL*8`. The CM-2/200 computes send addresses as 4-byte (`INTEGER`)values; the CM-5 uses 8-byte (`DOUBLE PRECISION` or `REAL*8`) send addresses. Each platform will accept either 4-byte or 8-byte send address arrays. However, there may be a performance penalty for using 4-byte addresses on the CM-5, as the system coerces the values to 8-byte length. There is a minimal performance penalty for using 8-byte send-address arrays on the CM-2 (one array copy). Therefore, for maximum portability, most CM Fortran programs that compute send addresses should declare them as `DOUBLE PRECISION` or `REAL*8` values. `INTEGER` send address arrays should only be used in programs to be run on the CM-2 in which the marginally greater memory use is an issue.

### CMF_SEND_ADD

The subroutine `CMF_SEND_ADD` can operate on numbers of the following types:

- `INTEGER`
- `REAL`
- `DOUBLE PRECISION`
- `COMPLEX`
- `DOUBLE COMPLEX`

### CMF_SEND_MAX, CMF_SEND_MIN

The subroutines `CMF_SEND_MAX` and `CMF_SEND_MIN` can operate on numbers of the following types:

- `INTEGER`
- `REAL`
- `DOUBLE PRECISION`

### CMF_SEND_IOR, CMF_SEND_IAND, CMF_SEND_IEOR

The subroutines `CMF_SEND_IOR`, `CMF_SEND_IAND`, and `CMF_SEND_IEOR` can operate on numbers of the following types:

- `INTEGER`
- `LOGICAL`

These operations correspond to logical inclusive OR, logical AND, and logical exclusive OR, respectively. For `INTEGERs`, these subroutines do the operation on a bitwise basis.

### CMF_SEND_OVERWRITE

The subroutine `CMF_SEND_OVERWRITE` operates on all the element types. The combining function used by this subroutine arbitrarily chooses one of the values being combined as the output. That is, if there are multiple elements of *INDEX* with the same index value, one of the corresponding values of *SOURCE* is arbitrarily chosen and written into *DEST*.

## SEE ALSO

```
CMF_MAKE_SEND_ADDRESS
CMF_MY_SEND_ADDRESS
CMF_DEPOSIT_GRID_COORDINATE
```

### NOTES

The CM Fortran Utility Library procedures do not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.

## NAME

**CMF_SIZEOF_ARRAY_ELEMENT** - Returns the size of an array for use with `CMF_FILE_LSEEK`, `CMF_FILE_LSEEK_FMS`, and `CMF_FILE_TRUNCATE`.

## SYNTAX

```
INCLUDE '/usr/include/cm/CMF_defs.h'
```

*LENGTH* = `CMF_SIZEOF_ARRAY_ELEMENT` (*ARRAY*)

## ARGUMENTS

*ARRAY* A CM array of any type.

## RETURNED VALUE

```
INTEGER
```

## DESCRIPTION

This function returns the size of a single array element of *ARRAY*.

This return value can also be passed to `CMF_FILE_LSEEK`, `CMF_FILE_LSEEK_FMS`, and `CMF_FILE_TRUNCATE` to specify the length of an array.`CMF_SIZEOF_ARRAY_ELEMENT` can be called for multiple arrays stored in a file and the return values added together to compute file positions for these routines.

See the man pages for `CMF_FILE_LSEEK`, `CMF_FILE_LSEEK_FMS`, and `CMF_FILE_TRUNCATE` for more information.

## SEE ALSO

`CMF_DESCRIBE_ARRAY`

## NAME

**CMF_SORT** – Sorts the elements along an array axis (or axis segment) by numerical ranking and places the values in order in the destination array.

## SYNTAX

```
 INCLUDE '/usr/include/cm/CMF_defs.h'

 CALL CMF_SORT(DEST, SOURCE, SEGMENT, AXIS, DIRECTION, SEGMENT_MODE,
&               MASK)
```

## ARGUMENTS

*DEST* A CM array of the same type as *SOURCE*. The destination array. The *DEST* and *SOURCE* arrays must be of the same shape and layout.

*SOURCE*
A CM array of any type. The *SOURCE* and *DEST* arrays must be of the same shape and layout.

*SEGMENT*
A `LOGICAL` CM array of the same shape and layout as *DEST*, *SOURCE*, and *MASK*. `.TRUE.` values in the *SEGMENT* array are used as segment delimiters for the corresponding elements of the *SOURCE* array.

If *SEGMENT_MODE* has a value of `CMF_NONE`, then this argument is ignored and may be `CMF_NULL`. Otherwise, each segment is sorted independently. The arguments *SEGMENT_MODE*, *DIRECTION*, and *MASK* control the way the sorting proceeds over the segments. See the **DESCRIPTION** section below for details.

*AXIS* An integer. The axis of *SOURCE* to be sorted.

*DIRECTION*
An integer. The value can be `CMF_UPWARD` or `CMF_DOWNWARD`. If the value is `CMF_UPWARD`, the values are sorted from the smallest value to the largest; the smallest value is assigned to the first element of the corresponding axis of *DEST*. If the value is `CMF_DOWNWARD`, the values are sorted from the largest value to the smallest; the largest value is assigned to the first element of the corresponding axis of *DEST*.

*SEGMENT_MODE*
An `INTEGER`. One of the following integer values: `CMF_NONE`, `CMF_SEGMENT_BIT` or `CMF_START_BIT`. This argument controls how the segments of *SOURCE* defined by the *SEGMENT* array are interpreted. See **DESCRIPTION** below for more information.

*MASK* A CM `LOGICAL` array, or the scalar value `.TRUE.`.

If the value of *MASK* is a scalar `.TRUE.`, all the values of *SOURCE* are sorted.

If *MASK* is a logical array, it must be of the same shape and layout as *DEST*, *SOURCE*, and *SEGMENT*. The values in *SOURCE* corresponding to values of `.FALSE.` in *MASK* are not sorted.

## RETURNED VALUE

None.

## DESCRIPTION

This subroutine sorts the values stored in the selected elements along one axis of the *SOURCE* array by numerical ranking, and stores the values in order into the *DEST* array. Selected elements are those that correspond to a `.TRUE.` element in the *MASK* array. Array elements that correspond to a `.FALSE.` value in the *MASK* argument are not sorted and the corresponding element of *DEST* is not changed.

In addition, the array elements along the axis may be partitioned into distinct sets, called *segments*, through the use of the *SEGMENT* and *SEGMENT_MODE* arguments. Each segment along the specified *AXIS* is treated as a separate set of values to be sorted. Each element of *SEGMENT* that contains `.TRUE.`, marks the corresponding element of *SOURCE* as a segment boundary (the start or end of a segment). Segments begin (or end) with each element in which the value of *SEGMENT* is `.TRUE.`, and continue up (or down) the axis through all elements where the value of *SEGMENT* is `.FALSE.`. The effect of these boundaries depends on the value of *SEGMENT_MODE*.

If *SEGMENT_MODE* is `CMF_NONE`, the elements are sorted along the entire length of the array axis and the values in *SEGMENT* have no effect.

If *SEGMENT_MODE* is `CMF_SEGMENT_BIT`, then:

- The *MASK* argument does not affect the use of the *SEGMENT* array. That is, elements containing `.TRUE.` in the *SEGMENT* array create a segment boundary even if the corresponding value of *MASK* is `.FALSE.`. (The *MASK* array still selects the elements of *SOURCE* to be included as described above.)
- A *SEGMENT* value of `.TRUE.` indicates the start of a segment for both upward and downward sorts.

If the value is `CMF_START_BIT`, then:

- The *MASK* argument applies to the *SEGMENT* array as well as to the *SOURCE* array. That is, elements containing `.TRUE.` in *SEGMENT* array create a segment boundary only if the corresponding element of *MASK* is also `.TRUE.`.
- A *SEGMENT* value of `.TRUE.` indicates the *start* of a segment for upward sorts, but the *end* of a segment for downward sorts. That is, the *SOURCE* element

corresponding to a `.TRUE.` *SEGMENT* element is the first element in a segment for an upward sort, but the last element in a segment for a downward sort. In downward sorts, the new segment begins with the first unmasked element following the segment boundary.

## EXAMPLES

### Upward Sort

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0],
and SEGMENT   = [T,   F,   F,   F  ],
then DEST     = [1.0, 2.0, 3.0, 7.0  ].
```

### Downward Sort

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   F,   F  ],
then DEST     = [7.0, 3.0, 2.0, 1.0].
```

### Upward Sort With Mask

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   F,   F  ],
and MASK      = [T,   T,   F,   T  ],
then DEST     = [1.0, 2.0, 7.0, X  ].
```

### Segmented Upward Sort

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   T,   F  ],
and MASK      = [T,   T,   T,   T  ],
then DEST     = [1.0, 7.0, 2.0, 3.0].
```

### Segmented Upward Sort With Mask

```
If SOURCE     = [1.0, 7.0, 3.0, 2.0]
and SEGMENT   = [T,   F,   T,   F  ],
and MASK      = [F,   T,   T,   T  ],
then DEST     = [7.0, 2.0, 3.0, X  ].
```

Note that, while each segment is sorted independently, the values are stored into the destination without regard to segments. As illustrated in this example, the selected values are packed into *DEST* in sorted order without preserving the segment boundaries: (`[7.0, 2.0, 3.0, X]`, not `[7.0, X, 2.0, 3.0]`).

## NOTES

The CM Fortran Utility Library procedures do not operate on arrays that have been aligned with other arrays of greater rank or with other arrays of the same rank but with offsets for the individual axes.

This routine assumes that the arrays have a lower bound of 1. All other lower bound values are ignored.