The
Connection Machine
System

CM Fortran Release Notes

Version 1.2
August 1992

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, August 1992

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

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

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

CM Fortran Release Notes
Version 1.2

1 About Version 1.2

CM Fortran Version 1.2 is a maintenance release, emphasizing improvements in reliability. This release also makes available to CM-2 and CM-200 users some new features that were released previously for the CM-5 only.

These release notes *replace* all previous release notes for CM Fortran, including Versions 1.0 and 1.1 (all CM models) and 1.1.3 (CM-5 only). New features and long-term restrictions described in the earlier release notes are repeated here.

1.1 Highlights of Version 1.2

The following features, new since Version 1.1, are now available for all CM models. (Many of these were released for CM-5 only in Version 1.1.3.)

- Numerous bugs have been fixed—about 60 in all. The major ones are noted in Section 9; known bugs still outstanding are listed in Section 10.
- The global optimizer (the `-O` switch) is much more robust.
- The **`cmf`** switches **`-cmprofile`**, **`-g`**, and **`-cmdebug`** enable a program to run under the Prism development environment. The older debugger **`cmdbx`** is now deprecated (Section 3.1).

- The **cmf** switches **-cm5**, **-cm2**, and **-cm200**, and the environmental variable **CMF_DEFAULT_MACHINE**, enable you to specify the hardware platform for which a program should be compiled (Section 3.2).
- Send-address arrays used by Utility Library procedures can now be declared either integers or double-precision reals for any CM hardware platform (Section 4.1). We recommend declaring send-address arrays as double-precision real for maximum portability with little or no performance penalty.
- Two new Utility Library procedures, **CMF_ARCHITECTURE** and **CMF_NUMBER_OF_PROCESSORS**, provide information about the CM system that is executing a program (Section 4.2).
- Two new Utility Library procedures, **CMF_RANK** and **CMF_SORT**, provide enhanced sorting capabilities of numerical array values. **CMF_RANK** is similar to **CMF_ORDER**, except that it permits ranking values within segments of an array axis. **CMF_SORT** writes the sorted values themselves to a destination array. (Section 4.3)
- Two new Utility Library procedures, **CMF_FILE_FDOPEN** and **CMF_FILE_GET_FD**, enable the utility I/O procedures to work on devices as well as files by translating between CM Fortran unit numbers and CMFS (CM File System) file descriptors (Section 4.4).
- Two new Utility Library procedures, **CMF_CM_ARRAY_TO_FILE_SO** and **CMF_CM_ARRAY_FROM_FILE_SO**, enable you to read and write CM files from the CM-2/200 in normal Fortran order ("serial order"), which is the order compatible with the CM-5 and CM-HIPPI (Section 4.5). CM-5 users, please note that the behavior of these procedures has changed since V1.1.3 (Sections 4.5 and 4.6).

New users of CM Fortran should note the descriptions here of compiler switches (Section 3.3) and the C language preprocessor (Section 6), features added in Version 1.1. These features and the ongoing language restrictions (Section 7) have not been incorporated into the *CM Fortran User's Guide* for the CM-2/200.

Two restrictions are newly documented with this release (see Section 7):

- Certain main programs may require you to initialize the parallel processing unit, even if they contain no parallel operations.
- You may need to work around restrictions on passing noncanonical array expressions as arguments (by providing an interface block, for example).

1.2 Porting from Version 1.1

CM Fortran programs developed under Version 1.1 should be recompiled and relinked to execute under Version 1.2.

A code change is required in CM-5 programs that used the serial-order (SO) I/O utility under V1.1.3 to write CM files. These files should be read under V1.2 with the "no suffix" I/O procedure, not with the SO procedure. See Section 4.5.

1.3 Current Documentation

The complete documentation set for CM Fortran V1.2 is as follows. The information in these manuals applies to all CM hardware platforms, except where noted.

CM Fortran Release Notes, V1.2, August 1992 [this document]

Getting Started in CM Fortran, November 1991
CM Fortran Reference Manual, V1.0 and 1.1, July 1991
CM Fortran Programming Guide, V1.0, January 1991

CM Fortran User's Guide, V1.0 and 1.1, July 1991 [CM-2/200 only]
CM Fortran User's Guide, V1.1.3, January 1992 [CM-5 only]

CM Fortran Optimization Notes: Slicewise Model, V1.0, March 1991
CM Fortran Optimization Notes: Paris Model, V1.0, February 1991 [CM-2/200 only]
CM Fortran Master Index, V1.0 May 1991

On-line man pages for the **cmf** compiler command, all CM Fortran intrinsic functions, and all procedures in the CM Fortran Utility Library. To view these man pages, use the command **man** on CM-5 or the command **cmman** on CM-2/200. Enter the comand or function name in all upper case.

NOTE: The descriptions of the Utility Library procedures in the two user's guides are incomplete and slightly outdated. Please refer to the on-line man pages for up-to-date information on the utility procedures.

NOTE: The master index dates from Version 1.0, and thus covers only the material of that vintage.

2 Software Compatibility

Version 1.2 of CM Fortran is implemented for the CM-2, the CM-200, and the CM-5. This section notes the versions of the CM system software and layered products that are compatible with this release.

2.1 CM System Software

On the CM-5, Version 1.2 requires CMOST Version 7.1 or later.

On the CM-2 and CM-200, Version 1.2 requires CM System Software Version 6.1 or later.

2.2 Sun FORTRAN Not Required

Previous releases required that CM Fortran programs be linked on a system with Sun's FORTRAN 77 libraries installed. This restriction is lifted with Version 1.2.

For the convenience of users whose programs call functions in the Sun library **`libF77.a`**, CM Fortran now provides these functions in **`libcmf77.a`**. If Sun FORTRAN 77 is installed, the CM Fortran compiler driver links with both the **`libcmf77.a`** and **`libF77.a`** libraries. Thus, object files compiled with Sun's **`f77`** command will link successfully.

The **`libcmf77.a`** library functions, listed in the table below, all have on-line man pages. View them with the command **`cmman`** on CM-2/200 and the command **`man`** on CM-5. To avoid name conflicts with the Sun library **`man`** pages, specify the function name in all upper case.

`access`	`getgid`	`lstat`
`alarm`	`getlog`	`ltime`
`chdir`	`getpid`	`perror`
`chmod`	`getuid`	`qsort`
`ctime`	`gmtime`	`rand`
`drand`	`hostnm`	`rename`
`dtime`	`iargc`	`rindex`
`etime`	`idate`	`signal`
`exit`	`ierrno`	`sleep`
`fdate`	`irand`	`stat`
`fork`	`itime`	`symlnk`
`free`	`kill`	`system`
`gerror`	`link`	`time`
`getarg`	`lnblnk`	`unlink`
`getcwd`	`loc`	`wait`
`getenv`		

There are a few differences between this list of functions and the contents of Sun's **`libf77.a`**.

- Sun's library includes the functions **`INDEX`** and **`LEN`**. These are intrinsic functions in CM Fortran (not library functions in **`libcmf77.a`**). They are described in the *CM Fortran Reference Manual*, and their on-line man pages are accessible with the command **`man`** on CM-5 or **`cmman`** on CM-2/200 (specify function name in all upper case).
- The CM Fortran library **`libcmf77.a`** does include the functions **`RINDEX`** and **`LNBLNK`**, and CM Fortran provides separate man pages for them.

2.3 Layered Products on CM-2/200 Systems

On CM-2/200 systems, CM Fortran Version 1.2 is compatible with the following versions of CM layered software products:

System Component	Link with
Prism, 1.1	-
CMSSL for CMF, 3.0	`-lcmssl[-s]`
CM Graphics, 2.0	
*Render for CMF	`-lcmsr`
Generic Display for CMF	`-lcmsr`
Image File Interface for CMF	`-ltiff`
Display Operations for Fortran, 2.0	-
CM File System for Fortran, 6.1	`-lcmfs`

Some notes on this table:

- Notice that the CMSSL library is not automatically linked. Use the suffix **-s** to link CMSSL for the slicewise execution model; omit **-s** for the Paris model.
- The three libraries in CM Graphics 2.0 provide a CM Fortran interface. The Display Operations ("Framebuffer") library provides a Fortran/Paris interface. The Display Operations library is part of **libparisfort.a**, although its version numbering has been changed to conform to the rest of the graphics software.
- The CM Fortran Utility Library provides utilities that call certain procedures in the CM File System library. The utilities open, close, read, write, truncate, and "lseek" CM files. Some other CMFS procedures can be called directly, via their Fortran interface.
- This release supports the practice of calling the C interface to any CMFS procedure from CM Fortran. New Utility Library procedures enable this practice by translating between CM Fortran unit numbers and CMFS file descriptors (see Section 4.4).

2.4 Layered Products on CM-5 Systems

On CM-5 systems, CM Fortran Version 1.2 is compatible with the following versions of CM layered software products:

System Component	Link with
Prism, 1.1	—
CMX11 Graphics, 1.3	See *CMX11 Reference Manual*
CMSSL Scientific, 3.0 Beta	See *CMSSL Release Notes*
CM File System for C, 7.1	See *Using the CM-5 I/O System* and *Using CM-HIPPI on the CM-5*

Some notes on this table:

- The CMX11 Graphics library is supported only on the CM-5.
- A version of the Scientific Software library for the CM-5 is currently in beta test.
- The CMMD Message Passing library, supported on CM-5, is not currently compatible with CM Fortran.
- The CM-2 Graphics and Paris libraries are not supported on the CM-5.
- Please use the CM Fortran Utility Library for parallel I/O. To call lower-level software (for example, to use CM-HIPPI), use the C interface to the subset of CMFS procedures that are supported on the CM-5.

2.5 CM Fortran Execution Models: CM-2/200 Only

Version 1.2 may be installed on CM-2 and CM-200 systems with either the slicewise or Paris execution model as the default. Use the compiler switch **`-paris`** or **`-slicewise`** to choose the non-default option. CM-5 systems support only one execution model, comparable to the slicewise model.

On the CM-2/200 any CM Fortran source code can be compiled for either execution model, but the models are not object-code compatible. A program unit compiled for one execution model cannot be linked with a program unit compiled for the other model.

3 New and Enhanced Compiler Switches

CM Fortran compiler switches have been enhanced to permit the use of the Prism development environment and to enable you to select the CM hardware platform for which to compile a program. (Some of these enhancements have been previously released for the CM-5 only.)

3.1 Switches for Using Prism

The following switch is now available on all CM platforms.

`-[no]cmprofile` default: **`-nocmprofile`**

Produce information needed for performance analysis under the Prism development environment. If used, this switch should be used during both compilation and linking.

By default, the **`-cmprofile`** switch activates the **`-cmdebug`** switch: Prism performs performance analysis on a block-by-block basis (with source code lines fused together). To analyze a program on a line-by-line basis, relating performance to source code lines, specify the **`-g`** switch. (By suppressing certain optimizations, the **`-g`** switch causes the program to execute artificially slowly.)

In addition, the switches **`-g`** and **`-cmdebug`**, which produce information needed for program debugging, now enable a program to run under the Prism development environment. For remote users or those without X Windows, Prism provides a shell interface that resembles **`cmdbx`**:

```
% prism -C
```

NOTE

The **cmdbx** debugger is deprecated now that the superior Prism development environment is available on all CM platforms. Please use the Prism Version 1.1 debugger instead of **cmdbx**.

3.2 Switches for Specifying CM Model

The CM hardware platform that the compiler targets by default is determined for a site at installation time. You can change the default by means of the user environmental variable **CMF_DEFAULT_MACHINE**. Possible values are **CM5**, **CM2**, and **CM200** (case is not significant).

The following **cmf** switches are now available on the CM-2/200, as well as on the CM-5.

-cm5

Compile for a CM-5 system.

-cm2

Compile for a CM-2 system.

-cm200

Compile for a CM-200 system.

The CM hardware platforms are not object-code-compatible. That is, the **.o** files generated under any one of these switches cannot be mixed with **.o** files generated under the other switches. Also, the system signals an error at run time if a load module prepared for one platform is executed on another platform.

3.3 Switches Released in Version 1.1

The following compiler switches were added to CM Fortran in Version 1.1. They appear in the new CM-5 *CM Fortran User's Guide* (January 1992), but have not yet been added to the user's guide for the CM-2/200.

`-safety=`*`level`*

Enables run-time safety checking at a level specified by an integer value. This switch can be used on the CM-5 and with the slicewise execution model on the CM-2/200. It is not supported with the Paris model (with **`-paris`**, use the **`-argument-checking`** switch instead.)

The key safety levels are:

`0`	No safety checking
`1`	Provides the same safety checks as the **`-argument_checking`** switch, which include checking the validity of send addresses and the number and homes of CM array arguments. Any argument from `1` through `9` provides these checks.
`10`	Provides the checks above plus NaN checking for CM arrays of type real or complex, under the slicewise execution model. This level of checking also causes program memory to be initialized to a known value (currently, the value -1), which may help to detect the use of uninitialized real or complex variables. Any argument of `10` or greater provides these checks.

You cannot use this switch with calls to the Utility Library routines that allocate arrays dynamically (see restriction in Section 7.8 below). Run-time safety checking mistakes the homes of dynamically allocated arrays and signals an error.

`-list`

The listing file produced by this switch now identifies the communication routines generated and the source code line numbers at which each reference occurs. For example, the source lines of a (somewhat contrived) program **`xref.fcm`** would appear in the listing file **`xref.lis`** as:

```
Source Listing File: /users/user-name/xref.fcm
    1      1  1        program xref
    2      2  1        parameter (m = 10)
    3      3  1        real a(m), b(m)
    4      4  1        integer v(m)
    5      5  1        a = [1:m]*17.0
    6      6  1        v = [1,4,3,2,7,6,9,8,10,5]
    7      7  1        a(v) = a*3
    8      8  1        print 10, a
    9      9  1     10 format( ¢ A:¢, 10F9.3 )
   10     10  1        loop: do 100 i=1,m
   11     11  1          b(i) = log(real(i*i*i))
   12     12  1          a(i) = a(i)*b(v(i))
   13     13  1          if (i==9) exit loop
   14     14  1    100   continue
   15     15  1        print 10, a
   16     16  1    200 end
```

The listing file reports the communication routine references as:

```
COMMUNICATION ROUTINES
  Name                      Line Number (number of times)
  READ VALUE FROM PROCESSOR  12(2)
  VECTOR SEND                7
  FE TO CM ARRAY TRANSFER    6
```

The example code generates references to three different communication routines: `READ VALUE FROM PROCESSOR` on line 12, `VECTOR SEND` on line 7, and `FE TO CM ARRAY TRANSFER` on line 6. (`VECTOR SEND` is a general communication routine to handle vector-valued subscripting.) If more than one reference to a communication routine appears on a single line, that number is indicated in parentheses following the line number.

Many of the communication routines support the intrinsic functions directly, and references to them use the name of the intrinsic function itself (possibly qualified), such as `CSHIFT`, `MAXLOC`, `SUM` (into scalar), and `SUM` (into vector). Others refer to common CM communication patterns: `SEND`, `GET`, `VPMOVE`, `NEWS`, and `NEWS (power of two)`. Still others refer to data transfers between the CM and the front end: READ VALUE FROM PROCESSOR, FE TO CM ARRAY TRANSFER, and so on. The listing also reports uses of SUBROUTINE ARGUMENT COPYOUT.

-[no]cross_reference

When used together with the **-list** switch, this switch causes the listing file to include information that relates line labels and names (symbols) to source code lines. (Add the switch **-show_include** if you want the contents of include files to be listed also.) The **-cross_reference** switch is ignored if the **-list** switch is not specified. The default is **-nocross_reference**. The symbol and label cross reference listings generated for the program listed in the previous bullet are shown below.

```
Symbol Cross Reference   File: /users/usr-name/xref.fcm
Symbol       Line Number(s)


A            3    5    7    7    8    12   12   15
B            3    11   12
I            10   11   11   11   11   12   12   12   13
LOG          11
LOOP         10   13
M            2    3    3    4    5    10
REAL         11
V            4    6    7    12
XREF         1

Label Cross Reference   File: /users/usr-name/xref.fcm
Label         Defined References(s)
10            9    8    15
100           14   10
200           16
```

-D*name* | **-D***name*=*def*

Defines the symbol *name* for use by the C language preprocessor **cpp**. The first form sets the value of *name* equal to 1; the second form sets its value equal to *def*. The switch has the same effect as a `#define` preprocessor directive (see Section 6 below).

-pecode

This switch has been modified to behave like the **-s** switch. That is, the program is compiled and linked, but the intermediate file containing the PE assembler code is retained (the intermediate file has the extension **_peac.peac**). (This switch is supported only on the CM-2 and CM-200.)

4 Enhancements to the Utility Library

Several new Utility Library procedures (for system information and for I/O) are available to CM-2/200 users with CM Fortran Version 1.2.

The new procedures were previously released in Version 1.1.3 for the CM-5 only. They are documented in the *CM Fortran User's Guide for the CM-5*, January 1992.

In addition, two new sorting utilities have been added, and the send-address procedures have been enhanced to permit easy porting between CM models.

4.1 Enhancement of Send-Address Types

The procedures that take send-address arrays as arguments have been enhanced to facilitate porting programs between CM platforms. These procedures can now take a double-precision real array or an integer array as the send-address argument on any CM hardware platform.

The procedures affected are:

CMF_MAKE_SEND_ADDRESS
CMF_MY_SEND_ADDRESS
CMF_DEPOSIT_GRID_COORDINATE
CMF_SEND_*combiner*

The CM-2/200 computes send addresses as integers (4-byte values), whereas send addresses on the CM-5 are 8-byte integers. Since CM Fortran does not support an 8-byte integer type, CM Fortran programmers writing for the CM-5 declare send-address arrays as **DOUBLE PRECISION** or **REAL*8**.

We recommend, for maximum portability among CM platforms, that all CM Fortran programs declare send addresses as double-precision values. There is a performance penalty for using integer send-address arrays on the CM-5, as the system coerces the values to the proper length. In addition, addresses for arrays larger than 2^{32} cannot be represented in 4 bytes. In contrast, there is only a marginal performance penalty for using double-precision send-address arrays on the CM-2/200 under the slicewise execution model (one array copy operation), along with the slightly greater use of memory.

4.2 System Information Procedures

The following procedures enable a program to determine at run time what CM platform, what execution model (on CM-2/200), and what number of parallel processors are executing the program.

CMF_ARCHITECTURE()

Returns an integer constant that identifies the CM model and execution model under which a program is running. The returned value is one of:

```
CMF_CM5_SPARC
CMF_CM200_SLICEWISE
CMF_CM200_PARIS
CMF_CM2_SLICEWISE
CMF_CM2_PARIS
```

CMF_NUMBER_OF_PROCESSORS()

On the CM-5, returns as an integer the number of nodes in the partition executing the program. On the CM-2/200, returns the number of nodes (slicewise model) or the number of bit-serial processors (Paris model) executing the program.

4.3 Numerical Ranking and Sorting Procedures

Two procedures have been added to the CM Fortran Utility Library to enhance its sorting capabilities. Man pages are available on line for these procedures.

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

Determines the numerical rank of each element along an array axis and stores the rank of that element into the corresponding element of a destination array. This is the same operation performed by the previously released utility procedure **CMF_ORDER**, but **CMF_RANK** enables you to control the direction of the ranking and to partition the array axis into segments that are ranked independently.

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

Ranks the elements along an array axis and places the sorted values in order into a destination array. This procedure enables you to specify the direction of the sort and to define axis segments.

4.4 Procedures for I/O via Devices

New CM Fortran utility procedures enable you to perform I/O via CM-HIPPI, VME, or CM sockets. To do so, you need to access lower-level parallel I/O procedures, as described in the documentation for CM I/O and CM-HIPPI. These lower-level procedures use file or socket descriptors, rather than CM Fortran unit numbers. The new CM Fortran utility procedures associate such descriptors with the unit numbers required by the CM Fortran utility I/O procedures.

The procedure **CMF_FILE_FDOPEN** associates the file or socket descriptor of a previously opened "file" (or device) with a CM Fortran unit number. You can then use the unit number in a call to **CMF_CM_ARRAY_TO/FROM_FILE_SO** (always use this "serial order" variant of the read/write utilities). The procedure **CMF_FILE_GET_FD** translates between unit numbers and CMFS file descriptors, which enables you to call the low-level routines of the CMFS (CM File System) library from a CM Fortran program.

CMF_FILE_FDOPEN (CMFS_FD, UNIT, IOSTAT)

Associates the descriptor of an open CM file system file or a CM socket with a CM Fortran unit number. Both values are input values; the procedure establishes an association between them.

CMFS_FD	Integer; a CMFS file or socket descriptor.
UNIT	An integer variable containing a valid unit number [1:29].
IOSTAT	An integer variable into which the status of the I/O operation will be placed. A positive value indicates success; a negative value indicates failure.

CMF_FILE_GET_FD (CMFS_FD, UNIT, IOSTAT)

Given a CM Fortran unit previously initialized by a call to either **CMF_FILE_OPEN** or **CMF_FILE_FDOPEN**, associates that unit with the descriptor of a CM file system file or a CM socket. It returns the descriptor in the argument variable **CMFS_FD**.

CMFS_FD	Integer; a CMFS file or socket descriptor.
UNIT	An integer variable containing a valid unit number [1:29].
IOSTAT	An integer variable into which the status of the I/O operation will be placed. A positive value indicates success; a negative value indicates failure.

4.5 Procedures for Parallel I/O in Serial Order

Two new Utility Library procedures perform I/O directly from the parallel processing unit, but they read and write files in normal Fortran order (or "serial order"), rather than in a parallel order reflecting array geometry and machine size. These new utilities give you the option of reading and writing data in the order that is portable across CM models (CM-2/200 and CM-5) and compatible with CM-HIPPI.

CMF_CM_ARRAY_TO_FILE_SO (UNIT, SOURCE, IOSTAT)

Writes the contents of a CM array to a CM file in serial order.

UNIT	An integer variable containing a valid unit number [1:29].
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; a negative value indicates failure.

CMF_CM_ARRAY_FROM_FILE_SO (UNIT, DEST, IOSTAT)

Reads an array from a CM file in serial order.

UNIT	An integer variable containing a valid unit number [1:29].
DEST	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; a negative value indicates failure; a zero value indicates an end-of-file condition.

Serial-Order Files

File data written with the SO utility is stored in the same order as data written with the 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)
```

Unlike the other parallel read/write utilities, the SO utilities do not "pad" files. Because they read and write only the array elements, not any extraneous data, these utilities operate independently of the array geometry and of machine size and model. Serial-order files are completely portable across the range of CM configurations.

CM-5 users, please note that this behavior is a change from Version 1.1.3. The SO utilities did pad files under some circumstances in that release. To read files previously written from the CM-5 with the SO write procedure, please use the generic read procedure, **CMF_CM_ARRAY_FROM_FILE**. The generic utilities under Version 1.2 on CM-5 behave the same way the SO utilities behaved under V1.1.3; the SO utilities now behave differently in that they never pad files.

Writing to Devices

When writing to devices, use the serial-order I/O utilities. In this situation, the "file" is either a CM-HIPPI device or a CM socket. You need to open it with the utility described earlier, **`CMF_FILE_FDOPEN`**, and then use **`CMF_FILE_GET_FD`** to relate its file or socket descriptor to a Fortran unit number. You use the unit number in a call to **`CMF_CM_ARRAY_TO/FROM_FILE_SO`**.

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 the following restrictions when writing to devices:

- From 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 CM-2/200:

  Write from arrays whose size (number of elements) is a power of 2 and an integer multiple 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.

See the documentation on the CM I/O system and the CM-HIPPI (for CM-2 and CM-5, respectively) for more information on this form of I/O programming.

4.6 Parallel File I/O: Behavior and Restrictions

The CM Fortran Utility Library now provides three variants of the procedures that read or write CM arrays in parallel, that is, in multiple streams directly between the memory of CM processors and the storage device. This section describes the behavior of the three variants and the restrictions that apply to each.

Arguments

The three variants are distinguished by suffix (or lack of): no-suffix or generic, FMS, or SO. They take the same arguments.

CMF_CM_ARRAY_TO_FILE [, / *_FMS*, / *_SO*] **(UNIT, ARRAY, IOSTAT)**
CMF_CM_ARRAY_FROM_FILE [, / *_FMS*, / *_SO*] **(UNIT, ARRAY, IOSTAT)**

UNIT The unit number can be a variable, parameter, or literal constant in the range 1:29 (inclusive). It is associated with a file by using it first in a call to **CMF_FILE_OPEN**. Such "unit numbers" have no relation to the CM Fortran unit numbers that are used in front-end I/O (described in the *CM Fortran Reference Manual.)*

ARRAY The array is a CM array of any type that is the source or destination of the I/O operation. Like all arrays used with CM Fortran utility procedures, it cannot be aligned with another array of higher rank or aligned with an array of the same rank but with dimensions offset with respect to each other. However, unlike other utility procedures, the I/O procedures can operate on arrays whose lower dimension bounds are not necessarily one.

IOSTAT This argument is an integer variable into which the status of the operation is placed. For all the I/O procedures, a positive value indicates success and a negative value indicates failure. In addition, for all the parallel read utilities, a zero value indicates an end-of-file condition. Other than sign or zero, there is no significance to any of the particular values returned.

Behavior

The three sets of the I/O 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. The generic (no-suffix) routines are a compromise between the two for general-purpose use. Always read a file with the same variant that was used to write it.

Variants of **CMF_CM_ARRAY_TO/FROM_FILE**.

	FMS	Generic	SO
CM-2/200			
File order	parallel	parallel	serial
Padding, if any	scattered	scattered	none
Portability	CM-2/200 only same machine size same exec. model same array shape same array layout	CM-2/200 only any machine size any exec. model same array shape canonical array only	any CM or device any machine or partition size any exec. model any array shape canonical array only
CM-5			
File order	parallel	parallel	serial
Padding, if any	scattered	scattered	none
Portability	CM-5 only same partition size same array shape same array layout	CM-5 only any partition size same array shape canonical array only	any CM or device any partition or machine size any array shape canonical array only

The FMS and generic procedures write to a file in a *parallel* order that reflects the geometry of the array and, in the case of FMS, the array layout and the size of the machine executing the program. The FMS routines require that a later read operation be to an identical array in a program running on the same size machine under the same execution model. With the generic routines, which handle only canonical arrays, the array shape must be the same but the machine size or execution model can be different.

In addition, all arrays written to a parallel-order file must have the same shape. The geometry of a new parallel file is established by the shape of the first array written to it, and subsequent writes to the file must be of arrays with identical shape. Arrays written with the FMS routines must also have the same layout as the first array written.

The FMS and generic routines may write extraneous data (padding) to scattered locations within the file. As long as you observe the restrictions noted in the table, the padding is handled transparently when the file is read.

The SO procedures store data in files in serial order (array-element order) with no padding. Although the arrays read to or written must have canonical layout, there are no other restrictions on the portability of serial-order files, and arrays of any shape can be written to the same file.

When using the generic and SO routines, note that a noncanonical array can be changed to a canonical layout by means of an array assignment.

File Operations

All seek, rewind, and truncate operations on CM files must be preceded by a read or write operation. It is necessary first to establish the geometry of a newly opened file, even a serial-order file, by performing a read or write of the file. An additional restriction on the CM-5 only is that 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.

The procedures **CMF_FILE_LSEEK** and **CMF_FILE_TRUNCATE** operate on both parallel-order and serial-order files (use **CMF_FILE_LSEEK_FMS** for parallel files written with the FMS utility). However, there is a difference in how you calculate the offset (for seek) or length (for truncate) argument.

- For serial-order files (those created with the SO utility), you can seek or truncate either to an array or to an arbitrary element. For the offset or length argument, use the number of bytes in the array's element type times the number of elements to traverse.

- For parallel-order files, you can 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**.

As an example, suppose that a file associated with unit 29 was created with three successive writes, of array **A**, then array **B**, then array **C**. Assume that **SIZEOF_***X* is the value returned by **CMF_SIZEOF_ARRAY_ELEMENT(X)**.

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 pointer to the beginning of array **C**, use:

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

4.7 Note on Front-End I/O: Appending to a File

CM Fortran provides no automatic method to append a record to a file written from the front end or partition manager by means of the **WRITE** statement. The language does not support the VAX **APPEND** mode in the **OPEN** statement, and has not yet implemented the Fortran 90 **POSITION** keyword by which you could position the file pointer at the end of the last record.

To append to a file, read to end-of-file and then **BACKSPACE**. If all the records are the same length, you can use direct-access I/O.

5 Note on Floating-Point Exceptions

The CM-2/200 and the CM-5 handle floating-point exceptions in an IEEE standard manner. Overflow, division by zero, inexact, and invalid operands are masked in CM arrays of floating-point types. Overflow turns into a properly signed infinity, and division by zero (except 0/0) turns into a properly signed infinity. Underflow turns into zero instead of a denormalized number.

Floating-point exceptions are handled differently by Sun front-ends (CM-2/200) and control processor (CM-5), although still conforming to IEEE standards. In scalars and front-end arrays of floating-point types, the system executes traps for overflow, divide by zero, and invalid operand; underflow turns to zero; and inexact is masked.

Users should note that if a CM Fortran program has a **STOP** statement in it and if the program is linked with the Sun FORTRAN 77 library, Version SC1.0, the following message appears :

```
Note: Following IEEE floating-point traps are en-
able; see ieee_handler(3M): Overflow; Division by
Zero; Invalid Operand;
Sun's implementation of IEEE arithmetic is dis-
cussed in the Numerical Computation Guide.
```

This message is informational only. You can prevent its appearing by setting the following environmental variable to null:

```
% setenv CMF_SUN_FORTRAN_DIR
```

6 The C Language Preprocessor (From Version 1.1)

Beginning with Version 1.1, the CM Fortran compiler accepts files designated for preprocessing with the C language preprocessor. This feature is described in the CM-5 *CM Fortran User's Guide*; the description is included here since it has not yet been incorporated in the user's guide for the CM-2/200.

The **cmf** command driver accepts files with (uppercase) extensions of **.FCM**, **.F**, and **.FOR**, and invokes the C language preprocessor **cpp** on each file before passing it on to the appropriate compiler. (These extensions correspond to their lowercase counterparts used for CM Fortran, Sun FORTRAN 77, and VAX FORTRAN source files, respectively).

The C preprocessor can provide a useful conditional compilation facility for CM Fortran source code when used with the **cmf** command line switch **-D** described in the section on new compiler switches.

For example, the following program contains preprocessor control lines that conditionally define a parameter **N**, which is used in the declaration of a matrix **A**.

```
      PROGRAM CPP
#if ASIZE > 0 && ASIZE < 10
      PARAMETER (N = ASIZE)
#else
      PARAMETER (N = 9)
#endif
      CHARACTER*10 FMT
      INTEGER A(N,N)
      A = 0
      FORALL (I = 1:N, J=1:N) A(I,J) = I*10 + J
      WRITE (FMT, 10 ) N
10    FORMAT( ¢(1X,¢, I2.2, ¢I3)¢ )
      PRINT FMT, TRANSPOSE(A)
      END
```

The preprocessor control lines (those beginning with the character #) test whether the value of the symbol **ASIZE** is in the range 1 to 9 and, if so, select the first **PARAMETER** statement for compilation, otherwise the second. (The control lines themselves are filtered from the file actually passed to the compiler, along with the unselected **PARAMETER** statement.) The value for **ASIZE** is substituted for all occurrences of the symbol **ASIZE** in the program; the value of symbol **ASIZE** can be defined in the source code, on the command line, or it can be left undefined (in which case it assumes the value zero). If the program is in the file **cpp.FCM**, then the command line

```
% cmf -DASIZE=7 -P cpp.FCM
```

causes the matrix **A** to be declared as a 7x7 array. (The **-P** switch is passed on to the **cpp** program, and should be specified to circumvent a rather obscure compiler problem.)

The manual page for **cpp** describes the program switches and preprocessor command lines in detail, including a facility for defining macros with arguments.

7 New and Ongoing Restrictions

This section notes two newly documented restrictions in CM Fortran (Sections 7.1 and 7.2). It also restates several long-term restrictions, previously reported in CM Fortran release notes (Version 1.0 or 1.1).

7.1 May Need to Declare a CM Array in Main

A main program that performs no parallel operations does not initialize the parallel processing unit. If the main program has no parallel operations, the parallel processing unit is initialized by the first subprogram that does perform some parallel operation.

This arrangement usually causes no inconvenience. However, if such a main program should include calls to CM timer routines or to certain utility routines, such as **CMF_ALLOCATE_TABLE**, the program fails with a run-time error. The timer and certain utility routines fail unless the parallel processing unit has been initialized.

To work around this restriction, simply declare a CM array in a main program. (Forcing the array onto the parallel unit requires a **LAYOUT** directive as well as a specification statement.)

```
      INTEGER WORKAROUND(10)
CMF$  LAYOUT WORKAROUND(:NEWS)
```

The parallel processing unit is initialized when a CM array is declared, even if it is not used in an executable statement.

7.2 Noncanonical Array Expressions as Arguments

If an array expression involving an array with noncanonical layout is passed as an argument, the wrong values are passed for some elements even when the layouts of the caller and the called routines agree. No error is reported.

The problem arises because the array expression is evaluated in a temporary, and the temporary is passed as the argument. However, in the absence of an interface block, the temporary is laid out in canonical order; its element-order thus does not match the order expected by the dummy argument.

To have this operation execute correctly, you can either:

- Do the expression assignment to the actual argument before the call

  ```
  CALL SUB(A/1.5)          ! error

  A = A/1.5                ! correct
  CALL SUB(A)
  ```

  Notice that a noncanonical whole array is passed correctly.

- Provide an interface block in the caller that specifies the noncanonical layout of the array expression:

  ```
        INTERFACE
        SUBROUTINE SUB(X)
        INTEGER, PARAMETER :: GS = 5
        REAL, ARRAY(1:3, 1:GS, 1:GS) :: X
  CMF$  LAYOUT X(:SERIAL, :NEWS, :NEWS)
        END INTERFACE
  ```

7.3 FORALL Statement Limitations

In Version 1.2, the following forms of the **FORALL** statement generate code that executes serially. The compiler issues appropriate warning messages for these forms.

- A reference to an external function anywhere in a **FORALL** statement, such as,

  ```
  FORALL ( I = 1:F(3) ) A(G(I)) = H(I)
  ```

  Any of the references to functions **F**, **G**, and **H** is sufficient to cause the **FORALL** statement to be executed serially.

- The use of a **FORALL** index name in any of the following contexts (assume that **I** is an index name associated with a **FORALL** statement):
    - in an array constructor, such as **[I]** or **[1:I]**
    - as a subscript in an array element designator specifying an element of a front-end array, such as **FE(I)**
    - as an argument to a statement function, such as **FUN(I)**
    - in a triplet subscript, such as **1:I or I:I+5:2**
    - as an argument to a transformational intrinsic function (with the exception of **PROJECT** and **SPREAD**), such as **CSHIFT(X,I,2)**

  Note that use of a **FORALL** index name in any of the reduction intrinsics (**ALL**, **ANY**, **COUNT**, **MAXVAL**, **MINVAL**, **PRODUCT**, and **SUM**) does not inhibit parallelism.

The compiler still has difficulties compiling some forms of the **FORALL** statement into the most efficient CM operations. The CM Fortran Utility Library provides fast procedures that serve as replacements for those forms of **FORALL**. See the *CM Fortran User's Guide.*

7.4 Restrictions on Array-Valued User Functions

Array-valued user functions are supported in this release, but only for result arrays whose size can be determined at compile time and whose layout is canonical.

An example of an array-valued function is the following. Function **BINARY** converts an integer value to its binary representation, returning the result as a 32-element integer array.

```
FUNCTION BINARY( N )
INTEGER BINARY(0:31), N
BINARY = 0  ! FORCE RESULT TO CM MEMORY
BINARY=IBITS( [32[N]], [0:31], [32[1]] )
END
```

The function can be called by the program:

```
        INTERFACE
          FUNCTION BINARY( N )
          INTEGER BINARY(0:31)
        END INTERFACE
        PRINT 10, BINARY(1024+7)
    10  FORMAT( 1X,32I1 )
        END
```

Note the presence of the interface block. An interface block *must* be provided in any program unit that references an array-valued function. The interface block must describe the type and shape of each of the function's arguments and its result. If any of the arguments or the result has non-canonical layout, the interface block must indicate the layout of those arguments and the result. However, no **LAYOUT** directives are currently permitted on function result variables, so functions returning non-canonical result arrays cannot be defined or referenced.

A function declaring an array-valued result is flagged as an error if the size of the array result is not known at compile time, as in the following case:

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

7.5 Array Restrictions

This section restates the continuing restriction that the homes and shapes of actual array arguments must match the corresponding dummy arguments. There is also a limitation on the total size of serial dimensions of a CM array under the Paris execution model on the CM-2.

Array Argument Home

As specified in the *CM Fortran Reference Manual*, the *home* of an actual array must match the home of a dummy array argument to which it is passed. Consequently, a failure occurs if a CM actual array is passed to a front-end dummy array, or if a front-end actual array is passed to a CM dummy array. The programmer must ensure that the actual and dummy array homes match. An example illustrates this restriction:

```
        PROGRAM ERRONEOUS
        PARAMETER ( NA = 1000, NB = 500 )
        REAL A(NA), B(NB)
        A = [1:NA]              ! A is a CM array
        DO I = 1, NB
          B(I) = NB             ! B is an FE array
        END DO
        CALL SQUARE( A, NA )! OKAY: CM actual to CM dummy
        CALL SQUARE( B, NB )! ERROR: FE actual to CM
    dummy
        ...
        END

        SUBROUTINE SQUARE( X, N )
        REAL X( N )
        X = X*X                 ! Dummy X is a CM array
        END
```

The program above fails at the second call to subroutine **SQUARE** because of an array home mismatch. This will occur even if the main program and subprogram are compiled as part of the same file, since CM Fortran compiles program units completely independently of one another.

The compiler can detect this problem if interface blocks are used. Alternatively, compile with the switch **-safety=1** (or higher value) to catch this kind of error at run time.

Array Argument Shape

CM Fortran requires that the *shape* of an actual argument match the shape of the corresponding dummy argument in two cases:

- in a reference to a procedure with an explicit interface (i.e., with an interface block present)
- if the actual argument and the dummy argument are CM arrays

The compiler enforces the first case in most situations. The programmer must enforce the second case.

A procedure reference fails at run time if a CM array actual argument is passed to a CM array dummy argument of a different shape. For example, the subroutine

```
SUBROUTINE SUB( A, N )
REAL A(N)
A = A - 1.0          ! A is a CM array
END
```

should be called passing an array section, as with

```
CALL SUB( X(1:J), J )
```

The subroutine can be called with the statement

```
CALL SUB( X, J )
```

only if array **X** actually has the shape **[J]**, which would be the case if **X** were declared using the statement

```
REAL X(J)
```

Compile with the switch **-safety=1** (or higher value) to catch mismatches in *rank* at run time. The compiler currently does not generate code to check for shape mismatches.

Size Limit of Serial Dimensions [CM-2/200]

The following restriction applies only to programs running on the CM-2/200 that use the Paris library at any level. Such programs include those compiled under the Paris execution model, any programs that use Paris-based CM-2 libraries (CMFS or Graphics), and any programs that use the CM Fortran I/O utility procedures.

This restriction does not apply to programs executed on the CM-5.

There is a restriction on the total size occupied by the serial dimensions of a CM array as specified with the **LAYOUT** directive): the product of the size (rounded to a power of two) of all serial dimensions must not exceed 65536 bits. This restriction follows from a limitation of the Paris field addressing mechanism that uses 16 bit offsets with respect to a field ID.

In CM Fortran terms, this implies the following total size limitations, in elements, for the serial dimensions of an array of each possible type, assuming VP ratio of one:

65536	for **LOGICAL** arrays
2048	for **REAL** or **INTEGER** arrays
1024	for **DOUBLE PRECISION** real or **COMPLEX** arrays
512	for **DOUBLE COMPLEX** arrays

For a VP ratio **N** greater than one, divide the numbers above by **N**.

The compiler does not enforce this restriction on the length of serial dimensions, and a program that exceeds the size limit will fail. The problem manifests itself only on machines with more than 2^{16} bits of memory per processor.

7.6 ENTRY Statement Limitation

The **ENTRY** statement does not work correctly in many cases and should not be used in the current version.

7.7 Restrictions on Directives

ALIGN Directive Restrictions

The **ALIGN** directive is supported, except that an *index-value* with a leading minus sign is not permitted.

Order of Directives

In CM Fortran, **COMMON** directives must precede **ALIGN** and **LAYOUT** directives. A **COMMON** directive establishing a default home for arrays of common block **BLK** may not be followed by an **ALIGN** or **LAYOUT** directive establishing a different home for one of the arrays in **BLK**, or a compiler error message is issued.

Order of Directives and Executable Statements

In CM Fortran, all compiler directives must appear in the specification part of the program, before the first executable statement. Directives that appear after any executable statement are ignored.

7.8 Dynamic Array Allocation and Argument Checking

It is not possible to use run-time safety or argument checking when executing programs that call the subroutines **`CMF_ALLOCATE_ARRAY`** or **`CMF_ALLOCATE_LAYOUT_ARRAY`**. This is because the allocation subroutines declare the CM array as a front-end array descriptor, and when they are passed a CM array argument, the argument-checking code signals an error. When compiling such programs, do not use the **`-argument_checking`** switch or the **`-safety`** switch.

7.9 Data Segment Size Limitation [VAX front end to CM-2 only]

The VAX **`lk`** linker is limited in the size of object files it can handle. The problem can occur when linking for execution on the CM-2 under either the Paris or the slicewise model, but it is more likely under slicewise. There are several possible ways of eliminating this problem.

- If possible, build and execute the program using a Sun front end.
- Compile and link the program with the **`-pecode`** switch. This may work if the data segments are fairly evenly distributed between the object file (extension **`.o`**) and the PE code file (extension **`_peac.peac`**).
- If this fails, reduce the size of the larger procedures of your application by splitting them into smaller procedures.

8 Deprecation of Paris Calls (From Version 1.0)

Explicit calls to Paris are not recommended in CM Fortran programs: they are generally slow under the slicewise model on the CM-2/200 and are not supported at all on the CM-5. The CM Fortran Utility Library is intended to make Paris calls unnecessary. This section repeats earlier advice on upgrading existing code that uses explicit Paris calls.

8.1 Paris and the Two Execution Models (CM-2/200 Only)

Explicit Paris calls perform as expected when compiled for the Paris model on the CM-2/200. We recommend that you upgrade your Paris code to use the CM Fortran Utility Library (or CM Fortran itself) at your convenience.

If it is not convenient to upgrade your Paris code immediately, you must make some changes (described below) in the use of the array descriptor access functions before you can run your program under the slicewise model on the CM-2 or CM-200.

These changes will *not* enable your program to execute on the CM-5; for this purpose, you must remove all Paris calls and replace them with calls to the utility library procedures.

8.2 Paris and the Utility Library

The Utility Library gives you access (under both Paris and slicewise models) to the highly efficient CM operations that the CM Fortran compiler does not yet generate. Conversion from Paris to these utility procedures is straightforward. (See the *CM Fortran User's Guide* for complete information on the Utility Library.)

For example, compare the CM Fortran *(left)* and Paris *(right)* scan procedures:

```
CMF_SCAN_op(                        CM_scan_with_{s,u,f}_op(
   DEST_CM_ARRAY,                      dest,
   SOURCE_CM_ARRAY,                    source,
   SEGMENT_CM_ARRAY,                   axis,
   AXIS,                               len,
   DIRECTION,                          direction,
   INCLUSION,                          inclusion,
   SEGMENT_MODE,                       smode,
   MASK_CM_ARRAY )                     sbit )

where op is one of:                 where op is one of:
   ADD, MAX, MIN, COPY,                add, max, min, copy,
   OPR, IAND, IEOR                     logior, logand, logxor
```

Similarly, compare the CM Fortran and Paris procedures for array send with combining:

```
CMF_SEND_op(                        CM_send_with_{s,u,f}_op(
   DEST_CM_ARRAY,                      dest,
   SEND_ADDRESS,                       send_address,
   SOURCE_CM_ARRAY,                    source,
   MASK_CM_ARRAY)                      len,
                                       notify)
where op is one of:
   MAX, MIN, ADD,                   where op is one of:
   OVERWRITE,                          max, min, add,
   IOR, IAND, IEOR                     overwrite,
                                       ior, iand, ieor
```

8.3 Field ID Access Function

The array descriptor function **CMF_GET_FIELD_ID** is obsolete under the slicewise model and on the CM-5. If your code references this function, and if it is not convenient to update your code to call the utility library, you can use the technique described here as an interim measure on the CM-2/200 only.

CM Fortran provides a "wrapper" interface to lower-level software on the CM-2/200. The wrappers, named for Paris procedures, take array names (and thus, descriptors) rather than field ID's as arguments. The wrappers then generate either Paris or the slicewise run-time routines, as appropriate.

For example, to operate on array **A**, replace this:

```
ID = CMF_GET_FIELD_ID( A )
CALL CM_F_ADD_2_1L( ID, ID, 23, 8 )
```

with this:

```
CALL CM_F_ADD_2_1L( A, A, 23, 8 )
```

9 Bugs Fixed in Version 1.2

This section describes implementation errors that have been corrected in Version 1.2, listed in order by reference number. The list is divided into those that have not been previously reported (Section 9.1) and those that have been previously reported (Section 9.2).

All bugs pertain to all CM platforms and both execution models, except where otherwise noted.

9.1 Bugs Discovered and Fixed Since Version 1.1

In general, the optimizer (the **-o** switch) is now much more robust on all platforms and execution models.

#925 Hollerith constant as argument caused a compile-time error

#928 Compiler generated incorrect code for integer constants greater than 13 bits (2^{12})

#931 Repeated use of a scalar complex constant variable caused the compiler to generate erroneous code

#936 A logical `IF` followed by an arithmetic `IF` caused a segmentation violation

#951 The compiler generated incorrect code for the power operator (**) when used with complex values

#956 Nested array constructors failed [Paris]

#964 Vector-valued subscripts could fail at run time [Slice and CM-5]

#969 `DO WHILE` failed when the global reduction function `ANY` occurred as part of control expression [Sun and CM-5]

#970, 1013, 1015 The compiler generated incorrect optimized code [-O, Slice, and CM-5]

#980 The compiler generated erroneous calls to `CMRT_cross_geometry_move` [Slice and CM-5]

#981 The `END=` and `ERR=` constructs produced segmentation errors [CM-5, -Slice on Sun]

#983 Assumed-size character arrays could not be passed to a function

#1001 Array assignments failed with different layouts [Paris]

#1014 A double-complex value could be lost for a common subexpression when there was a function call to an intrinsic that used that value

#1020 Programs compiled for profiling (-pg switch) failed at run time [VAX]

9.2 Bugs Previously Reported and Now Fixed

In general, the optimizer (the **-O** switch) is now much more robust on all platforms and execution models.

#312 Concatenation and character function results [Slice and CM-5]

A user-defined function involving assignment of a concatenation expression to a character-valued function result could cause a compiler failure.

#467 Effect of `INQUIRE` statement could be optimized away

The effect of the `INQUIRE` statement was optimized away if the variables in which the results were returned had been assigned earlier in the program.

#530 Array constructors of the form `[R,I:J]` failed

Array constructors of the form `[R,I:J]` where `I` and `J` were integers and `R` was real or complex caused an internal compiler error or assembler error.

#531 `FORALL` generated incorrect code [Slice and CM-5, –O]

Optimized code compiled from `FORALL` for the slicewise model or the CM-5 may sometimes failed.

#622 Substring operations on character function results failed

A substring operation on a character function result could cause an internal compiler error.

#623 Substring operations on character function results could fail [–O]

This bug caused a segmentation fault at run time with some `PRINT` statements; it also interfered with the operation of the I/O utility routines.

#675 Overlapping string copy not as documented [Sun and CM-5]

Assignment of overlapping substrings produced unexpected results.

#683 `DLBOUND` and `DUBOUND` could cause compiler failure

`DLBOUND` or `DUBOUND` could cause an internal compiler error when applied to array sections or array constructors.

#773 Assignment of array-valued function reference to section [Paris]

Assignment of an array-valued function reference to a section of an array with non-canonical layout could fail at runtime with a field access error.

#821 Optimization problems [-o]

Use of the **-o** switch during compilation caused several compiler problems.

#827 Assumed-size character arrays caused compiler failure

Assumed-size character array arguments caused an internal compiler error.

#828 Character comparison could cause compiler failure [VAX]

A relational expression comparing two character string literals sometimes caused a compiler error.

#832 TRANSPOSE used near MATMUL failed [Paris]

The **TRANSPOSE** intrinsic used in close proximity to the **MATMUL** intrinsic sometimes returned all zeros.

#836 FORALL computed wrong answer [Slice and CM-5]

The second **FORALL** statement miscalculated the elements of the plane **T(:,:,1)**. The problem occurred for N = 32 or any greater power of 2.

```
PARAMETER ( N = 32 )
INTEGER, ARRAY(N,N,N) :: A, T
A = 0
T = 0
FORALL (J=1:N, K=1:N, L=1:N) A(J,K,L) = L
FORALL (J=1:N, K=1:N) T(J,K,1) = A(J,K,1)+A(J,K,N) !ERR
PRINT *, T(:,:,1) == N+1
END
```

#862 **TRANSPOSE of a non-canonical array section failed**

Assigning the result of a **TRANSPOSE** of a non-canonical array section to itself caused a compiler-time error.

#892 **ALIGN directive could cause compiler failure**

An attempt to **ALIGN** an array **C** with an array **B** that has itself already been aligned with an array **A** caused a compiler DTBRTS error if array **A** had a **:SERIAL** dimension.

#916 **MATMUL failed on non-canonical arrays**

MATMUL failed for array arguments with non-canonical layout. Under Paris, failure occurred if the array had non-power-of-two size in at least one dimension; under slicewise and on the CM-5, the failure always occurred.

#918 **Use of DO loop index in an IF statement failed [slice and CM-5, –O]**

A **DO** loop index variable that was used as an array subscript in the condition expression of a logical **IF** statement could cause a compiler DTBRTS error if the index variable was also used in the action statement of the **IF** statement.

10 Bugs Outstanding in Version 1.2

This section describes known remaining implementation errors in Version 1.2, listed in order by reference number. The list is divided into those that have not been previously reported (Section 10.1) and those that have been previously reported (Section 10.2).

All bugs pertain to all CM platforms and both execution models, except where noted.

10.1 Bugs Discovered Since Version 1.1

This section reports outstanding bugs that have not been reported previously.

Doc Correction: Incorrect formula for calculating peak FLOPS on CM-2 slicewise

The formula given in Appendix A (page 39) of the *CM Fortran Optimization Notes: Slicewise Model*, Version 1.0, for calculating peak FLOPS rate is incorrect. The factor `vector-length` is extraneous and should be removed from the formula. This factor is already included in the figures reported in the **`.peac`** file.

#938 **`FORALL`** fails with array section assignment

A **`FORALL`** statement causes a compile-time error if there is an array section (a colon) on the left-hand side of the **`FORALL`** assignment. For example,

```
FORALL (J = 1:M) X(:,J) = A0(J, BASIS(J))
```

will not work. The workaround is to introduce a new index to cover the same range, as in:

```
FORALL (I = 1:M, J = 1:M) X(I,J) = A0(J, BASIS(J))
```

#946 The compiler fails to flag mismatched arguments when enabling run-time safety [**`-argument_checking, -safety`**]

The compiler fails to check for mismatched arguments. For example, the following user code compiles, even though a subroutine that expects a 3-dimensional array is called with a 1-dimensional array. When such code is executed, a segmentation fault results.

```
PROGRAM SEGFAULT
INTEGER FOO(3)
CALL FAULTSEG(FOO)
END

SUBROUTINE FAULTSEG(BAR)
INTEGER BAR(:,:,:)
RETURN
END
```

#950 Data transfer from front end to serial axis can fail [CM-2]

When reading data off the front end into a serial axis of a CM array, data can become corrupted.

To work around this, read into a temporary **:news** array. Then use an assignment statement to copy into the array with the serial dimension.

```
      DOUBLE PRECISION, ARRAY (NY,NX) :: SERIAL
CMF$LAYOUT SERIAL (:SERIAL,:NEWS)

      READ(18) (TEMP(I,:), I = 1, NY)
      SERIAL = TEMP
```

#957 The MATMUL intrinsic fails for large arrays [Slice and CM-5]

When given an array with a large on-chip subgrid, the **MATMUL** intrinsic fails with an "Integer division by zero" error.

The workaround is to make the array smaller or link with **-lcmssl-s** to use the library version of matrix multiplication.

#959 Compiler generates invalid Paris code for FORALL that results in a safety error during execution [Paris]

#961 NaN checking (for uninitialized variables) is improperly implemented for array sections, even if the array section (but not the whole array) has been initialized [Slice and CM-5]

The workaround is to initialize the whole array first.

#962 Paris logicals represented differently from scalar logicals [Paris]

When written to a file with **DIRECT** access and then read back, Paris logical values are represented differently from front-end logical values. Because of this, an equivalence test between Paris and front-end logical values fails.

#963 A compile-time error occurs when FORALL attempts to spread a 1D array to be used with a 3D array

#978 Compiling with `-list` with more than 19 include files causes a compile-time error

The workaround is to merge headers or sources so as to include fewer than 19 files.

#984 A compile-time error occurs on a substring expression with upper bound unspecified

The compiler takes the incorrect expression to be an array element reference. For example, an incorrect reference such as **CMSGIN(1)**, implies that **CMSGIN** is an array of characters instead of a character string. The character string should be referenced:

```
        ITYP = CMSGIN(1:1)
```

#989 A READ from a file into an array section corrupts data elsewhere in the array or in the other memory locations [Slice and CM-5]

The workaround is to use implied **DO** loop on the **READ** instead of an array section:

```
        READ (1) ((X(I,J),I=1,7),J=1,7)
```

#992 **FORMAT** statement omits data if it encounters an embedded end-of-record [Sun and CM-5]

#997 Multiple definitions of blank common can cause run-time error [Sun and CM-5]

If there are two definitions of blank common, the latter is taken as a redefinition. This can lead to a segmentation fault if the second definition is smaller than the first.

#1000 Compile-time error occurs from using the **REAL** intrinsic within an expression on a **DOUBLE COMPLEX** data type that is being assigned to a **DOUBLE PRECISION** type [Slice and CM-5]

The compiler fails to promote the **REAL** output to double-precision real unless a double-precision **MOLD** argument is supplied.

The workaround is to use **MOLD** argument in the **REAL** intrinsic to cast the result of the real to a double-precision type.

#1003 **SPREAD** intrinsic fails with array constructor argument

Use of **SPREAD** intrinsic with an array constructor as an argument causes a compile-time error.

The workaround is to assign the result of the array constructor to a temp. Use the temp as an argument to **SPREAD**.

#1005 **UNPACK** fails for serial array section [Slice and CM-5]

The **UNPACK** intrinsic fails at run time with an array section that indexes into a serial dimension.

The workaround is to copy the array section into a temp and then perform the unpack on the temp.

#1016 The **PACK** intrinsic doesn't work with dummy array arguments [Slice and CM-5]

PACK returns the correct answer if the array arguments are copied into local arrays first and then these local arrays are passed to the **PACK** intrinsic.

#1017 Passing character strings between Sun f77 and CM Fortran [Sun and CM-5]

Sun's **f77** compiler and CM Fortran have incompatible calling conventions for functions returning character strings. Sun **f77** passes/expects the address of the character string to be returned, the size of that character string, and finally the explicit arguments. CM Fortran, however, passes/expects only the address of the character string followed by the explicit arguments. In other words, CM Fortran does not pass the size of the character string.

A workaround exists if the caller is compiled with CM Fortran and the callee is compiled with Sun **f77**. Just pass an additional argument specifying the length of the character string to be returned. For example, in the main program below, change the function call to **TMP = I21C (5,I)**.

```
PROGRAM TEST000
CHARACTER LABEL *30,I21C*5,TMP*5
I=23
TMP = I21C(I)
LABEL='COOR.'//TMP
PRINT*,LABEL
END
```

#1018 –o switch restricted for assumed-size character strings [Sun and CM-5]

Extra garbage characters are printed for assumed-size character strings when the **–o** switch is specified.

The workaround is to print substrings instead.

#1021 Array constructors restricted for DATA attribute

If an array type declaration has a **DATA** attribute and if that array is initialized by an array constructor that uses a negated lower bound, then the array is initialized with the wrong data. For example,

```
 PARAMETER (IVELOCITY_MAX=2)
 INTEGER, ARRAY(-IVELOCITY_MAX:IVELOCITY_MAX), DATA ::
$    VELOCITY = [-IVELOCITY_MAX:IVELOCITY_MAX]
```

produces incorrect results. The workaround is to avoid negated lower bounds.

10.2 Bugs Previously Reported (Version 1.1)

This section reports bugs that were described in the release notes for Version 1.1 and are still outstanding.

#392 OPEN statement has wrong default for BLANK= specifier [VAX]

The **BLANK=** specifier of the **OPEN** statement should default to a value of **¢NULL¢** if the specifier is omitted; it incorrectly defaults to **¢ZERO¢**.

#527 Functions returning adjustable arrays not yet supported

Array-valued functions whose size cannot be determined at compile-time are currently flagged as erroneous by the compiler. The following valid example fails to compile.

```
FUNCTION R(A,B,N)
REAL A(N), B(N), R(N)
IF (SUM(A) > SUM(B)) THEN
   R = A
ELSE
   R = B
END IF
END
```

#541 PACK without VECTOR argument, passed to an intrinsic

The compiler may generate an error message for a reference to **PACK** with no **VECTOR** argument. This happens only if the reference appears as an argument to an array-valued intrinsic function whose result size is dependent on its input argument. In the example below, the first and second references to **PACK** work, but the third causes a compiler error message.

```
PARAMETER ( N = 30 )
INTEGER A(N)
A = [1:N]
PRINT *, PACK( A, MOD(A,2)==0 )                 ! okay
PRINT *, SUM( PACK( A, MOD(A,2)==0 ) )          ! okay
PRINT *, CSHIFT( PACK( A, MOD(A,2)==0 ), 1, 1 ) ! ERROR
END
```

#558 **FORALL with mask expression and variable indexes fails [–O]**

A masked **FORALL** statement with an index expression involving variables can fail to produce the correct answer when compiled with optimization.

#561 **Concatenation of character substrings may fail at run time**

Concatenation of a character substring with a string may cause a segmentation fault at run time if the starting or ending point of the substring range is not known at compile time. The example below fails if **STR1** and **STR2** are character strings and **N** is an integer variable whose value is not known at compile time.

```
      CALL FOO( STR1(1:N) // STR2 )
```

The example works if **N** is a literal or named integer constant.

#599 **No LAYOUT directive permitted for array-valued function results**

The compiler currently does not allow **LAYOUT** directives to affect the layout of array-valued function result variables. Such a function must be declared using an interface block. The compiler fails to compile the following example, complaining that **FOO** is an unknown array name.

```
      INTERFACE
        FUNCTION FOO( ARG )
        REAL FOO(4)
CMF$    LAYOUT FOO()      ! currently not supported
      END INTERFACE

      REAL A(4)
CMF$  LAYOUT A()
      A = FOO(42.0)
      END
```

#636 **Incorrect array home assumed for array-valued function result**

The result variable of an array-valued function should be allocated in CM memory, since all array-valued function results are CM arrays. The compiler fails make this happen unless the function result is used in an array operation that causes it to be allocated in CM memory. For example, the function below requires the assignment to **SQUARE** to force the result to CM memory.

```
FUNCTION SQUARE( IGNORED )
INTEGER SQUARE(4,4)
SQUARE = 0    ! force result to CM memory
FORALL ( I=1:4, J=1:4 ) SQUARE(I,J) = I*10 + J
END
```

The function above can be referenced from a program unit that declares the interface block:

```
INTERFACE
  FUNCTION SQUARE(IGGY)
  INTEGER SQUARE(4,4)
END INTERFACE
```

#668 **FORALL statement assigning a [m:n] array constructor fails**

The compiler fails with an internal error when attempting to compile the following program.

```
INTEGER B(3,4)
FORALL (J = 1:4) B(:,J) = [1:3]    ! compiler fails
PRINT *, B
END
```

Replacing `[1:3]` with `[1,2,3]` eliminates the problem.

#680 **FORALL with MERGE may compute wrong answer [Paris]**

A **FORALL** assignment in which the **MERGE** intrinsic is referenced may produce the wrong answer. The program below illustrates the problem: the results differ in the last element.

```
      INTEGER A(32), B(32), C(32)
CMF$  LAYOUT A(), B(), C()
      A = -99
      B = -99
      FORALL (I=1:32) C(I) = I - 16
      DO I=1,32
        A(I) = MERGE( C(I+1), -99, I+1 < 30 )   ! WORKS
      END DO
      FORALL (I=1:32)
     $  B(I) = MERGE( C(I+1), -99, I+1 < 30) ! FAILS
      PRINT *, A(32), B(32)
      END
```

#718 DO loop with real index may fail

A **DO** loop with a real index variable may not increment properly. The example below fails as is, but works if the third loop control expression **(PI/2)** is replaced by **PI**.

```
DO X = 0., PI*2, PI/2
  PRINT *, X
END DO

END
```

#772 Complex PRODUCT with mask gives incorrect answers [Paris]

The **PRODUCT** intrinsic sometimes fails on complex array arguments. The program below prints incorrect answers regardless of whether the **DIM** argument is 1, 2, or 3.

```
COMPLEX A(2,3,2)
A = RESHAPE ( [2,3,2], [2 [ 3 [(0,0),(2,2)]]] )
PRINT *, PRODUCT(A, DIM=1, MASK = A .NE. 0)
END
```

#817 DO WHILE loop with a .NOT. in control expression [Paris]

A **DO WHILE** loop whose control expression begins with the operator **.NOT.** may exit prematurely. The **DO WHILE** loop in the program below terminates (incorrectly) after one iteration; it should loop indefinitely.

```
LOGICAL DONE
DONE = .FALSE.
DO WHILE ( .NOT. DONE )
  CALL DOIT( DONE )
ENDDO
END

SUBROUTINE DOIT (DONE)
LOGICAL DONE
REAL TEMP(8)
TEMP = 42.0
DONE = .NOT. ANY (TEMP > 0)
END
```

A workaround is to rewrite the loop condition without the **.NOT.** operator, as in

```
NOTDONE = .TRUE.
DO WHILE ( NOTDONE )
  CALL DOIT( .NOT. NOTDONE )
ENDDO
END
```

#875 **MAXLOC/MINLOC may choose non-first max element [Slice and CM-5]**

MAXLOC fails to return the indexes of the first maximum element of an array if the array has more than one maximum. **MINLOC** suffers from a similar problem.

```
PRINT *, MAXLOC( [4,7,3,7,5,-2,3,-2] )
PRINT *, MINLOC( [4,7,3,7,5,-2,3,-2] )
END
```

The example above prints the values 4 and 8; the correct answer is 2 and 6.

#878 **FORALL with variable offset on index [Paris]**

The following program incorrectly prints zeros.

```
      PARAMETER (N=64, M=28)
      REAL A(N,N,N), T(M)
CMF$  LAYOUT A(,,), T()
      FORALL (I = 1:M) T(I) = EXP(-0.008*I**2/M)
      A = 1.
      FORALL (I=1:M, J=1:N, K=1:N) A(M-I+1,J,K) =
     $   A(M-I+1,J,K)*T(I)
      PRINT ƒ(I10, 2X, E14.6)ƒ, ( I, A(I,N,N), I=1,M )
      END
```

A workaround is to introduce a temporary CM array to allow the index expression **M-I+1** to be moved into another **FORALL** statement. The second **FORALL** statement then becomes:

```
FORALL (I=1:M) TMP(I) = T(M-I+1)
FORALL (I=1:M,J=1:N,K=1:N) A(I,J,K) = A(I,J,K)*TMP(I)
```

#884 SPREAD gives RTS error on subgrids larger than 64K words [Slice and CM-5]

The **SPREAD** function fails if the total size of the subgrid in which its **SOURCE** argument is allocated exceeds 64K words. The run-time system prints a diagnostic message if this error occurs.

```
        CALL TEST(2048) !Runs only on large-memory system
        END

        SUBROUTINE TEST(N)
        REAL A(N,N,4), REAL B(N,N)
        B = 17.
        A = SPREAD( B, DIM=3, NCOPIES=4 )
        END
```

A workaround is to align **C** with the original array **A** rather than the aligned array **B**, as in

```
CMF$    ALIGN C(I) WITH A(1,1,I)
```

#888 FORALL statement can cause RTS warning [Slice and CM-5]

The FORALL statement below generates a warning from the run-time system indicating that an invalid send/get address exists at specified array coordinates.

```
        INTEGER A(8), T(8)
        K = 4
        T = [1:8]
        A = -99
        FORALL (I=1:8, I<=4) A(I) = T(I + K)
        PRINT *, A
        END
```

#909 NaN checking of single-precision complex may fail [Slice and CM-5]

NaN safety checking for CM arrays of type single-precision complex (invoked using the **-safety=10** switch) may incorrectly warn of invalid values even when the values are legitimate.

```
        COMPLEX E(16)
        E = (2,3)
        PRINT *, E
        END
```

Safety checking works correctly for double-precision complex arrays.

#920 List-directed input of 80+ character records [Sun and CM-5]

List-directed input of a record with 80 or more characters fails with an end-of-file error.

```
LOGICAL A(50)
OPEN( UNIT=10, FILE=¢TEST1¢, STATUS=¢UNKNOWN¢ )
A = .FALSE.
WRITE( 10, * ) A
CLOSE( 10 )
OPEN( UNIT=10, FILE=¢TEST1¢, STATUS=¢UNKNOWN¢ )
READ(10,*) A   ! this statement fails
END
```

One possible workaround is to use formatted I/O to read and write files.