The
Connection Machine
System

# Paris Release Notes

**Version 5.1**
**June 1989**

**Thinking Machines Corporation**
**Cambridge, Massachusetts**

First printing, June 1989



Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1214
(617) 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.

To contact Thinking Machines Customer Support:

| | |
|---|---|
| **U.S. Mail:** | Thinking Machines Corporation<br>Customer Support<br>245 First Street<br>Cambridge, Massachusetts 02142-1214 |
| **Internet Electronic Mail:** | customer-support@think.com |
| **Usenet Electronic Mail:** | harvard!think!customer-support |
| **Telephone:** | (617) 876-1111 |

## For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl-M to create a report. In the mail window that appears, the To: field should be addressed as follows:

```
To:  bug-connection-machine@think.com
```

Please supplement the automatic report with any further pertinent information.

# About Version 5.1 Paris Documentation

## Intended Audience and Objectives

The Paris language and its documentation are intended for experienced developers of Connection Machine system software and applications. Version 5.1 Paris documentation is published to inform Paris programmers about new and modified Paris features.

## Revision Information

The version 5.1 Paris release notes and supplementary documentation supersede all previous Paris release notes and all past editions of the Paris sections of *In Parallel* software bulletin. Release notes distributed with versions 5.0, 5.1A Field Test, and with any earlier releases are now obsolete and should be removed from the *Programming in Paris* binder. *In Parallel* editions published before June 1989 should also be discarded.

## Organization of Version 5.1 Paris Documentation

**Paris Release Notes, Version 5.1**

The release notes broadly describe new and changed Paris features and detail language restrictions. Outstanding implementation and documentation errors are also reported.

**Change Pages to Paris Dictionary, Version 5.1**

The change pages document Paris instruction that have been changed for Version 5.1 and should be used to replace Version 5.0 dictionary pages with descriptions accurate for Version 5.1.

**Paris Reference Manual Supplement, Version 5.1**

The supplement documents Paris features new with Version 5.1. It includes an instruction overview section, organized by functionality, and a dictionary section, arranged alphabetically.

**Scientific Subroutines**

This short section describes two types of operations that mark the beginning of a Scientific Subroutines Library for the Connection Machine: Fast Fourier Transform routines and Matrix Multiplication routines are included.

## Related Manuals

- *Paris Reference Manual* Version 5.0, printed February 1989
- *Introduction to Programming in C/Paris* Version 5.0, printed June 1989
- *In Parallel*, Software Bulletin

  Published as necessary between releases of Connection Machine System Software versions, *In Parallel* provides up-to-date information including bug reports and programming hints. See the Paris section in each *In Parallel* issue published since the release of Version 5.1.

## Typeface Conventions

The table below displays the typeface conventions used in the Paris documentation.

| Convention | Meaning |
|---|---|
| **boldface** | Language elements, such as keywords and instruction names, when they appear embedded in text. |
| *italics* | Operand names and placeholders in instruction formats and when they appear embedded in text. |
| `typewriter` | Code examples and code fragments. |

## New Notation Convention

All Paris Version 5.1 documentation follows the conventions for alphabetizing, syntax, and pseudocode established at the beginning of Chapter 9 of the *Paris Reference Manual* Version 5.0. One new convention has been introduced.

In the Formats portion of dictionary entries, brackets, [ and ], enclose arguments that are either not provided, optional, or keywords in the Lisp/Paris interface. Wherever this notation is used, the Operands list states explicitly whether the brackets enclose unprovided, optional, or keyword arguments. For example, in the format line

**Formats** result ← **CM:intern-geometry** *dimension-array, [rank]*

the *rank* operand is not provided when calling Paris from Lisp.

# 1. About Paris Version 5.1

**The Paris Language** is the Connection Machine assembly language. Paris is currently the lowest-level instruction set available for programming the Connection Machine. It provides a large number of operations similar to the machine-level instruction set of a serial computer. Paris is intended primarily as the basis for higher-level Connection Machine languages such as *Lisp, C*, and CM Fortran. It may nonetheless be called directly from standard Lisp, C, or Fortran or from *Lisp, C*, or CM Fortran code.

**Paris Version 5.1** provides an expanded parallel instruction set and corrects a number of implementation errors present in Version 5.0.

## 1.1. Summary of New Features

These new features distinguish Paris Version 5.1 from earlier versions:

- *CM Fortran now generates Paris 5.1.* CM Fortran no longer needs to be run in back-compatibility mode.

- *Scientific subroutines.* Two operations from the Connection Machine Scientific Subroutines Library are included in this release.These are Fast Fourier Transformation (FFT) of complex numbers and matrix multiplication of either floating-point or complex numbers.

  A separate supplement to the Paris reference manual documents these scientific subroutines. As more numerical and scientific routines become available, they will be released as a library. At that time, these routines will no longer be incorporated into Paris proper.

- *Complex floating-point numbers.* Most Paris operations may now be performed on complex numbers, with the real and imaginary parts represented as either single- or double-precision floating-point numbers.

- *Moves across VP sets.* The new **CM:cross-vp-move** instruction copies data between VP sets.

- *Geometry and VP set interning.* A set of new instructions create geometry and VP set descriptors that may be reused, thus reducing CM memory management overhead. The names of these instruction all begin with **CM:intern**.

- *Field aliases.* New instructions introduce a mechanism for referencing the same CM field from within different VP sets.

- *NEWS instructions with floating-point operands.* A variety of new instructions perform calculations in which all operands are floating-point fields and one of these fields is taken from a NEWS neighbor. The names of these instruction all begin with **CM:f-news**.

- *Power of 2 NEWS.* With the new **CM:get-from-power-of-two** instructions, each processor retrieves data from another processor, where the distance in the NEWS grid between the source and destination processors is a power of two.

- *floating-point format conversion.* Two new operations, **CM:f-ieee-to-vax-1L** and **CM:f-vax-to-ieee-1L** convert floating-point numbers between the IEEE and VAX formats.

- *Heap compression.* A new memory management instruction, named **CM:compress-heap**, removes heap fragmentation.

Paris features new with Version 5.1 are documented in the *Paris Reference Manual Supplement* Version 5.1.

## 1.2. Summary of Changed Features

These Paris features existed in Version 5.0 and have been modified in Version 5.1:

- *Revised geometry creation.* The **CM:create-geometry** and **CM:create-detailed-geometry** instructions have been rewritten, as has their documentation.

- *Revised bit block transfers.* The **CM:read-from-news-array** and **CM:write-to-news-array** family of operations, which copy data between the CM and the front end, have been improved and expanded. These now support transfers of complex numbers, packed arrays, arrays of structures, and portions of multi-dimensional arrays.

Version 5.0 Paris features modified in Version 5.1 are documented by the packet entitled *Change Pages to Paris Dictionary* Version 5.1.

## 1.3. Status of Layered Products

- The CM Fortran compiler now generates calls to Version 5.1 Paris instructions and fully supports all Paris features, including the virtual processor architecture, *n*-dimensional NEWS, and instructions new with Version 5.1. CM Fortran programs no longer need to be executed in back-compatibility mode.
- *Lisp fully supports all Paris features, including the virtual processor architecture, *n*-dimensional NEWS, and instructions new with Version 5.1.
- The C* compiler generates code for Version 4.*x* Paris instructions only. C* programs must run in back-compatibility mode as C* does not take advantage of either the new virtual processor architecture or *n*-dimensional NEWS.
- The DataVault mass storage system uses the new Paris features. Programs that use the DataVault may be run either under Version 5.1 or in back-compatibility mode. (The lowest-level interface to the DataVault depends only on processor cube addresses.)
- The CM graphic display system uses Paris features. Whether programs that use display instructions run under Version 5.1 or in back-compatibility mode depends on the language from which the display instructions are called.

## 1.4. C/Paris Interface

### 1.4.1. Standard UNIX Math Library

The standard UNIX math library is no longer automatically linked with Paris. When linking 5.1 C/Paris code, it is therefore necessary to use the **-lm** switch. For example, the following **cc** command will link a module named **test.c** with a main program named **main.c**.

```
% cc main.c test.c -lparis -lm
```

### 1.4.2. C/Paris Header File

C/Paris typing information, contained in the **paris.h** header file has been updated and corrected for Version 5.1. C/Paris globals, constants, and functions are declared in this file. C/Paris types are declared in the file **cmtypes.h**, which is included in **paris.h**.

## 1.5. Back Compatibility

Version 5.1 supports all documented instructions provided in Versions 4.*x* and 5.0.

### Back-Compatibility Mode

Any existing programs that call Paris 4.*x* instructions must be recompiled and relinked with the new Paris object library and must be run in back-compatibility mode. Back-compatibility mode implements the 4.*x* stack discipline by allocating the stack in field zero and making stack addresses offsets into this field. See the *Front-End Systems Release Notes* Version 5.1, for information on executing programs in back-compatibility mode.

# 2. Implementation Restrictions

## 2.1. Maximum Message Length

The constant **CM:*maximum-message-length*** has been defined as 128. This constant is an upper bound on the number of bits transferred between processors by certain router instructions (**send**s and **get**s).

- The maximum message length restriction also applies to the following Version 5*x* router instructions:

  **CM:send-with-f-max-1L**
  **CM:send-with-f-min-1L**
  **CM:send-with-f-add-1L**
  **CM:send-aset32-overwrite-1L**
  **CM:send-aset32-u-add-1L**
  **CM:send-aset32-logior-1L**
  **CM:get-aref32**

- The following Version 5*x* router instructions have *no* message length restriction; their message size is limited only by available memory:

  **CM:get-1L**
  **CM:send-1L**
  **CM:send-with-overwrite-1L**
  **CM:send-with-logxor-1L**
  **CM:send-with-logior-1L**
  **CM:send-with-logand-1L**
  **CM:send-with-u-min-1L**
  **CM:send-with-u-max-1L**
  **CM:send-with-s-min-1L**
  **CM:send-with-s-max-1L**
  **CM:send-with-u-add-1L**
  **CM:send-with-s-add-1L**

- The limit on message length applies to the following Version 4*x* router instructions:

  **CM:send**
  **CM:send-with-overwrite**
  **CM:send-with-logior**
  **CM:send-with-logxor**

**CM:send-with-logand**
**CM:send-with-add**
**CM:send-with-max**
**CM:send-with-min**
**CM:send-with-unsigned-max**
**CM:send-with-unsigned-min**

## 2.2. Incomplete Support for IEEE Floating-Point

Support for IEEE floating-point instructions and flags is incomplete in Version 5.1. In particular:

- the five IEEE floating-point flags are not supported
- denormalized numbers are not supported
- **Infinity** and **NaN** values are only partially supported

Also, all Version 5.1 floating-point instructions:

- set the integer *test-flag* and the integer *overflow-flag* if division by zero occurs
- set the integer *overflow-flag* if floating-point overflow occurs
- set the integer *test-flag* in response to an invalid operation
- produce a zero result on underflow, with no other indication

When overflow occurs, the value stored in the destination field varies depending on the floating-point hardware present. The result may be 0.0, it may be a quiet **NaN**, or it may be the biased adjusted result specified by IEEE. Similarly, using a **NaN** as an operand to a floating-point instruction yields indefinite results.

# 3. Implementation Errors

Most of the Paris implementation errors reported in the *In Parallel* software bulletin issues for January, February, March, and April of 1989 are corrected in Version 5.1. The outstanding bugs are reported again in these release notes. All past issues of Programming in Paris *In Parallel* may therefore be discarded.

## 3.1. Corrected Errors

The following Version 5.0 implementation errors, reported in *In Parallel* Number 1, January 1989, are fixed in Paris Version 5.1.

**aref32**
**bitblt-cross-seq**
**cross-vp-send-f-add**
**deposit-news-coordinate**
**fortran-lib**
**illegal-psect**
**lib-not-profiled**
**libparis-pg**
**lvnp**
**mult-const-sub**
**prototypes**
**send-to-news**
**subfrom-const-always**
**u-to-grey-code**

The following Version 5.0 implementation errors, reported in *In Parallel* Number 2, February 1989, are fixed in Paris Version 5.1.

**aref32-index-bug**
**copy-scan-no-segments**
**exp-with-vps**
**negative-field-length**
**sincosatan-inaccurate**

The following Version 5.0 implementation errors, reported in *In Parallel* Number 3, March 1989, are fixed in Paris Version 5.1.

**f-s-power**
**f-u-power**
**signed-exponentiation-error**

No Paris implementation errors were reported in *In Parallel* Number 4, April 1989.

The error reported below has not been previously reported.

**ID** **create-detailed-geometry-bug**

This is corrected in Paris Version 5.1.

**Environment**

Paris, Versions 5.0; any front-end/CM configuration

**Description**

The axis weighting mechanism available with **CM:create-detailed-geometry** did not work properly at VP ratios higher than 1. Instead of favoring communication along axes of lesser weight, it favored axes that had been assigned greated weights.

---

## 3.2. Outstanding Errors

Version 5.1 of Paris has some known implementation errors, most of which have been previously reported. They are reported here in alphabetical order by bug report ID.

**ID** **c-star-simulator**

This was originally reported in *In Parallel* Number 1, January 1989.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1; any front-end/CM configuration

**Synopsis**

C* does not work with the Paris simulator.

---

### ID cm-get-1l-runs-out-of-mem

This was originally reported in *In Parallel* Number 2, February 1989.

#### Environment

Paris, Versions 5.0, 5.0.1, and 5.1; any front-end/CM configuration.

#### Description

Calls to **CM:get-1L** may cause the CM to run out of heap memory because **CM:get-1L** performs backward routing, a communication process that stores router trace information in order to speed interprocessor data transmission. The amount of memory required depends on the pattern being run. The following message indicates the executing program has run out of memory:

```
Forward sprint-send-with-trace has exceeded its allowed
space for saving out trace data.

CM Microcode Function: CMI::SAVE-OUT-PETIT-CYCLE-TRACE
```

#### Workaround

Use **CM:get**, the older version of **CM:get-1l**. This instruction is slower, but it uses far less memory than does **CM:get-1l**.

---

### ID cm-time-overflows

This was originally reported in *In Parallel* Number 2, February 1989.

#### Environment

Paris, Versions 5.0, 5.0.1, and 5.1; any front-end/CM configuration.

#### Description

The result returned by **CM:time** can become too large to fit into the 32 bits that are allocated to accumulate and store the total time. When this happens in

Lisp/Paris, control is transferred to the Lisp debugger; in C/Paris, **CM_time** returns an incorrect result.

---

**ID** **deposit-news-constant**

This was originally reported in *In Parallel* Number 1, January 1989.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1; any front-end/CM configuration.

**Synopsis**

**CM:deposit_news_coordinate_1L** and **CM:deposit_news_constant_1L** are documented to execute conditionally but, in the current implementation, they execute *un*conditionally.

---

**ID** **lintlib**

This was originally reported in *In Parallel* Number 1, January 1989.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1; any front-end/CM configuration.

**Synopsis**

The **lint** version of the Paris library does not work on the VAX front end.

**Description**

There is an ULTRIX bug that prevents our **lint** library from working.

---

**ID** **negative-field-length**

This was originally reported in *In Parallel* Number 2, February 1989.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1; any front-end/CM configuration.

**Description**

The field allocation routines, **CM:allocate-stack-field** and **CM:allocate-heap-field**, successfully return when passed negative lengths as arguments—even if safety is on. The negative lengths can later cause a CM exception.

---

**ID** **no-psim-on-sun4** and **psim-back-only**

This was originally reported under both ID's in *In Parallel* Number 1, January 1989.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1.

**Synopsis**

The Paris simulator only works in back-compatibility mode. Therefore, since the Sun front end is supported only by CM System Software versions 5.0 and higher, there is no working C/Paris simulator for the Sun front end.

---

**ID** **no-segment-bits-for-scans**

This was originally reported in *In Parallel* Number 3, March, 1989.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1; any front end with any CM configuration.

**Description**

None of the Paris **scan** instructions accept the **:segment-bit** (**CM_segment_bit**) value for the *smode* operand.

---

**ID** **send-to-news-wrong-context**

This has not been previously reported.

**Environment**

Paris, Versions 5.0, 5.0.1, and 5.1, any front end with any CM configuration.

**Synopsis**

For the Paris **send-to-news** operation, both the documentation and the implementation are in error. Execution of the conditional version of this operation *should* depend on the context of the sending processors; it instead depends on the context of the receiving processors.

**Description**

For **CM:send-to-news-1L**, the context bit of the source processors should determine which processors send messages to their neighbors. Instead, in the current implementation, the context bit of the destination processors is used to determine which processors receive messages. The Context portion of the dictionary entry should read as follows:

**Context** The non-**always** operation is conditional. The source value is set only by processors whose *context-flag* is 1.
The **always** operation is unconditional. The source value is sent regardless of the value of the *context-flag*.

The implementation should be changed to reflect this.

# 4. Documentation Errors

## 4.1. Corrected Errors

The instructions listed below were reported in *Paris Release Notes*, Version 5.0, as documented but not implemented. They are all now implemented and the documentation for them is correct.

| | |
|---|---|
| **CM:u-add-carry-3-1L** | **CM:u-add-carry-3-3L** |
| **CM:aref-2L** | **CM:aset-2L** |
| **CM:u-isqrt-1-1L** | **CM:{u,s}-move-const-always-1L** |
| **CM:s-s-power-3-3L** | **CM:{f,u,s}-rank-2L** |
| **CM:s-f-signum-2-2L** | **CM:s-s-signum-1-1L** |

The instructions listed below were reported in *Paris Release Notes*, Version 5.0, as documented under one name and implemented under another. The documented names are implemented in *Paris* Version 5.1. The names under which these instructions were originally implemented continue to exist to allow back-compatibility. Programmers are, however, cautioned against using the undocumented names, which may be removed in the future.

| Undocumented Name | Documented Name |
|---|---|
| CM:my-send-address-1L | CM:my-send-address |
| CM:swap-2-1L | CM:swap-1L |
| CM:send-aset32-logior-1L | CM:send-aset32-logior-2L |
| CM:send-aset32-overwrite-1L | CM:send-aset32-overwrite-2L |
| CM:send-aset32-u-add-1L | CM:send-aset32-u-add-2L |
| CM:float-move-decoded-constant | CM:f-move-decoded-constant-1L |

## 4.2. Outstanding Errors

A number of documentation errors in the *Paris Reference Manual*, Version 5.0, remain outstanding. A corrected edition of the manual will be published in the future. Meanwhile, Paris programmers are strongly urged to add the corrections suggested here to their manuals by hand.

### 4.2.1. Instruction Set Overview

#### Omissions

The charts in Chapter 5, "Instruction Set Overview," do not include the following operation names. However, these operations are implemented and they are documented in the dictionary.

**CM:extract-multi-coordinate**
**CM:field-vp-set**
**CM:move-decoded-constant**
**CM:{s,u,f}-rank-2L**

#### Inaccuracies

The charts in Chapter 5, "Instruction Set Overview," include the following operation names. However, these operations are neither included in the dictionary, nor are they implemented.

**CM:invert-bit**
**CM:{s,u}-round**
**CM:deposit-multi-coordinate**

### 4.2.2. Dictionary: General Problems

This section describes general problems with the Paris reference documentation. These are errors that occur in many instruction definitions.

#### C/Paris Types

The C/Paris Interface chapter is quite vague about the types of various Paris operands. In previous releases the header files **cmtypes.h** and **paris.h** were not entirely accurate either. In the future, the C/Paris type information will be more explicitly described in

the *Paris Reference Manual*. Meanwhile, the **cmtypes.h** and **paris.h** header files have been corrected for the release of Version 5.1. While we apologize for the inconvenience, C/Paris users are encouraged to use these header files as their definitive source of information about C/Paris operand and return value types.

### Field ID Type

The dictionary section of the *Paris Reference Manual*, Version 5.0, defines a field-id as an unsigned integer. Although field-id's are currently implemented as unsigned integers, this may not be true in future Connection Machine System Software versions.

This error occurs throughout Version 5.0 of the *Paris Reference Manual*. For instance, definitions for all the field allocation instructions should define the return values as field-id's rather than as the field-id's of unsigned integer fields. Similarly the *dest* and *send-address* arguments to instructions such as **CM:deposit-news-coordinate** should be defined simply as field-id's—not necessarily as field-id's of unsigned integer fields.

User code should not depend on the type of a field-id. C/Paris and Fortran/Paris code should conform to the language-specific field-id types given in the "C/Paris Interface" and the "Fortran/Paris Interface" chapters. Lisp/Paris code may rely on automatic coercion.

### Zero Length Operands

In Version 5.0 of the *Paris Reference Manual*, all Paris operations on unsigned integers are documented to permit *length* operands of value zero. However, as implemented, some do support zero *length* operands and some do not. Giving an unsigned instruction a *length* operand of value zero will cause obvious errors in some cases, will cause subtle errors in other cases, and will work correctly in still other cases. It is therefore inadvisable to pass zero *length* operands to operations on unsigned integers.

Zero *length* operands are generally not useful and therefore this inconsistency should not prove troublesome. If a workaround is needed, provide a one-bit field containing zero in each processor.

It is uncertain whether this restriction will persist in the future.

### Integer Immediate Operands

For all Paris instructions that take signed and unsigned integer immediate operands, which become constant operands once broadcast to the CM processors, the constant *must* be representable in the number of bits specified by the *len* argument.

The statement "The constant need not be representable in the number of bits specified by *len*." is, in the current implementation, false. This discrepancy between the documentation and the implementation applies to all binary arithmetic and integer constant operations such as, for example,

**CM:{s,u}-add**
**CM:{s,u}-max**
**CM:{s,u}-min**
**CM:{s,u}-mod**
**CM:{s,u}-multiply**
**CM:{s,u}-subtract**
**cm:{s u}-{lt, le, eq, ne, ge, gt}-constant-1L**

### Integer Division

Division on signed or unsigned integers is accomplished with the truncation operations, **CM:s-truncate**, **CM:s-f-truncate**, and **CM:u-truncate**. Chapter 5, "Instruction Set Overview," does not make this clear.

### CM Floating Point

The CM System Software currently does not fully support the IEEE standard for floating point operations. For every Paris floating-point instructions, the flags section of the dictionary entry should read:

Flags — *test-flag* is set if division by zero occurs; otherwise it is unaffected. *overflow-flag* is set if floating-point overflow (including division by zero) occurs; otherwise it is unaffected.
Underflow sets the result field to all zeros.

## 4.2.3. Dictionary: Specific Problems

This section describes specific problems in Version 5.0 of the Paris reference documentation. These are errors that affect only individual instruction definitions. They are listed here alphabetically by instruction name.

### CM:f-abs

If the source operand is a **NaN**, then the output is also a **NaN**. The dictionary entry erroneously claims that a **NaN** source is copied unchanged. The entry should read as follows:

> For floating-point numbers, absolute value is calculated by changing the sign bit to a 0 (positive). All other bits in the number are unchanged. As a result, the absolute values of negative infinities, denormalized numbers, and **NaN**s are their positive counterparts.

### CM:allocate-stack-field-vp-set and CM:allocate-heap-field-vp-set

The order in which operands to **CM:allocate-stack-field-vp-set** and **CM:allocate-heap-field-vp-set** are to be specified is documented as *vp-set-id, len*. However, as implemented, these instructions expect their arguments in the opposite order.

### CM:aref32-shared-2L and CM:aset32-shared-2L

For **CM:aset32-shared-2L** and **CM:aref32-shared-2L** (including the **-always** version), the *array* operand is not completely documented. The *array* field operand must be contiguous in CM memory. Therefore, it must be allocated all at once with a single call to **CM:allocate-stack-field**. Alternatively, the array may be allocated within a **with-stack-fields** form—but only if no other field is allocated within the same form.

### CM:deposit-news-coordinate-1L

The *coordinate* operand definition is misleading. To emphasize that this is a field, it should read:

*coordinate* The NEWS coordinate field. This specifies the position along the corresponding axis of the processor whose send address is to be calculated.

### CM:extract-news-coordinate and CM:extract-multi-coordinate

The *send-address* operand definition is wrong for both operations. It should read:

*send-address* The send address field. Within each processor, this identifies the send address of some other processor.

### CM:get-1L and CM:get-aref32-2L

In both initial descriptions, the phrase "from the same address" should read "from the same memory address."

The *send-address* operand definition is wrong for both operations. It should read:

*send-address* The send address field. This specifies the processor from which the message is retrieved.

### CM:initialize-random-number-generator

This operation is documented under the name **CM:initialize-random-generator**. It is, however implemented as **CM:initialize-random-number-generator**.

### CM:load-*flag*

**CM:load-overflow-always** and **CM:-load-test-always** are implemented. They are the unconditional versions of **CM:load-overflow** and **CM:load-test** and should be among the **CM:load-*flag*** instructions listed in the dictionary.

### CM:multispread

The definition formula for most of the **CM:multispread** operation dictionary entries contains the following errors. The statement "let r = rank( )" should read "let r = rank(g)." The statement "where *scan-subclass* is as defined on page 36" should read "where *hyperplane* is as defined on page 36."

### CM:my-send-address

The *dest* operand definition fails to mention the lower bound on this value. It should read:

*dest* The unsigned integer destination field. This must be at least equal to the value returned by **CM:geometry-send-address-length**.

### CM:s-s-power

**CM:s-s-power-constant-3-2L** is implemented. It should be among the **CM:s-s-power** instructions listed in the dictionary.

### CM:rank

For all the **CM:rank** instructions, the *dlen* operand definition fails to mention the upper bound on this value. It should read

*dlen* — The length of the *dest* field. This must be nonnegative, no greater than **CM:*maximum-integer-length***, and no larger than the value returned by **CM:geometry-coordinate-length**.

### CM:send-to-news

In the context description for the **CM:send-to-news** instructions, the first two paragraphs erroneously refer to the *context-flag* of the destination rather than to that of the source. It should read

Context — The non-**always** operation is conditional. The source value is sent only by processors whose *context-flag* is 1.
The **-always** operation is unconditional. The source value is sent regardless of the value of the *context-flag*.

### CM:store-*flag*

**CM:store-overflow-always** and **CM:store-test-always** are implemented. They are the unconditional versions of **CM:load-overflow** and **CM:load-test** and should be among the **CM:store-***flag* instructions listed in the dictionary.

The
Connection Machine
System

# Change Pages to Paris Dictionary

## Update fromVersion 5.0 to Version 5.1

June 1989

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, June 1989



Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1214
(617) 876-1111

# About Paris Version 5.1 Change Pages

## Purpose of Change Pages

Change pages correct and update a manual. The change pages in this packet document those Paris instructions that existed in Version 5.0 but which have been changed with the release of Version 5.1. Changed Paris instructions are documented with change pages rather than as part of the 5.1 Supplement to help readers avoid accidentally referring to obsolete documentation.

## What to Do with Change Pages

Take this packet of change pages and insert them, by page number, in the dictionary section of the *Paris Reference Manual*, Version 5.0. To do this, look at the page numbers on the change pages and compare them with those in the dictionary.

There are two kinds of change pages: replacement pages and additional pages.

### Additional Pages

Any change page with a page number ending in a letter must be added to the existing manual. Find the page whose number matches the number part of the change page number and insert the change page behind it. For example, the change page numbered 86a must be inserted after the existing page 86.

### Replacement Pages

Any change page with a normal page number replaces an existing Paris manual page. Tear out the existing page and replace it with the new one. For example, the old page 116 must be replaced by the change page numbered 116.

Note that some of the replacement pages are included only to preserve the order of the Paris dictionary entries. For example, although no changes have been made to the description of **CM:f-cosh**, it is reprinted because it appears on one side of the **CM:create-detailed-geometry** description, which has been updated.

**Placement of Change Pages**

| Change Page Sequence | Add after page | Replace pages |
|---|---|---|
| 86a, 86b | 86 | |
| 115, 116, 117, 117a, 117b, 118 | | 115–118 |
| 294a, 294b | 294 (blank) | |
| 312a, 312b, 312c | 312 | |
| 313, 314, 314a, 315, 316, 316a, 317, 318, 318a | | 313–318 |
| 435, 435a, 435b, 436 | | 435–436 |
| 459, 459a, 459b, 459c, 460, 461, 461a, 462, 463, 463a, 464, 465, 465a, 466 | | 459–466 |

After inserting all change pages, these explanatory pages and the title page for the change pages packet may be discarded.

## What Has Changed?

Two Paris features have been reimplemented and the documentation rewritten. Although the new implementations are backwardly compatible, they necessitate new documentation.

### CM:create-detailed-geometry

The documentation for this instruction has been updated to make it less confusing. In particular, the following changes have been made.

- A recommendation to use **CM:create-geometry** instead of **CM:create-detailed geometry** now follows the initial description.
- The definition of the *rank* operand is corrected to clarify that this operand specifies the rank of the geometry being created.

- The use of axis *weight* values is encouraged and emphasized while the specification of *on-chip-bits* and *off-chip-bits* is discouraged and deemphasized.
- There is a warning that NEWS operations and any grid-oriented operations do not work for axes with **:send** ordering.
- A common C/Paris error is warned against. From C, the *axis-descriptor-array* is an array of pointers to axis descriptors—not an array of axis descriptors.
- Example C/Paris and Lisp/Paris code is included to demonstrate how to use **CM:create-detailed-geometry.**

**CM:read-from-news-array** and **CM:write-to-news-array**

The entire suite of array transfer instructions has been reimplemented. The new documentation reflects this. In particular, the following changes have been made.

- Block transfers of complex numbers are now supported and documented.
- Several arguments are renamed to distinguish arguments that describe front-end data from arguments that describe CM data. Thus *offset-vector* is now *fe-offset-vector* and *start-vector*, *end-vector*, and *axis-vector* are now *cm-start-vector*, *cm-end-vector*, and *cm-axis-vector*.
- The *element-len* argument to all versions of both **CM:read-from-news-array** and **CM:write-to-news-array** has been replaced by a new argument named *format*. This is backwardly compatible with the old *element-len* but allows more explicit specification of the front-end array format.
- Three new instructions are provided and documented: **CM:fe-array-format**, **CM:fe-packed-array-format**, and **CM:structure-array-format**. These each return an array descriptor, which may be used as the value of the new *format* argument to any of the array transfer instructions.

# Contents

# FE-ARRAY-FORMAT

This front-end instruction returns an array format descriptor. An array format descriptor may be passed to any array transfer instruction to specify a front-end array format, although this is not required.

See also CM:fe-packed-array-format and CM:fe-structure-array-format.

---

**Formats** result ← CM:fe-array-format *[cm-element-size, array-element-size, stride, ordering]*

**Operands**

*cm-element-size* A signed integer immediate operand to be used as the number of bits each Connection Machine element occupies in the front-end array. This must be a power of two between 1 and 128.

In Lisp/Paris this is a keyword argument. If not specified, it defaults to *array-element-size*. If *array-element-size* is also not specified, *cm-element-size* defaults to the size of the Connection Machine field being read or written.

*array-element-size* A signed integer immediate operand to be used as the number of bits in each front-end array element. This must be a power of two between 1 and 128.

In Lisp/Paris this is a keyword argument. If not specified, *array-element-size* defaults to the actual front-end element size or, if the front-end array elements are general (i.e., of type t), *array-element-size* defaults to the value of *cm-element-size*.

*stride* A signed integer immediate operand to be used as the distance, in units of *array-element-size*, between adjacent front-end array elements. This must be either a null value or a positive integer between 1 and 65,535 that obeys the following restrictions. The product (*stride* × *array-element-size*) must be either a multiple of *cm-element-size* or a multiple of 32 bits. If *stride* is specified as a null value (null in C, 0 in Fortran, nil in Lisp), it defaults to the minimum legal value. In Lisp/Paris this is a keyword argument.

*ordering* The ordering in which Connection Machine elements are stored in a front-end array. The value of *ordering* must be either a null value or one of: :default-ordering, :lsb-first (least significant bit first), or :msb-first (most significant bit first). (These are CM_default_ordering, CM_lsb_first, or CM_msb_first from C or Fortran.) If specified as a null value (null in C, 0 in Fortran, nil in Lisp), it defaults to :default-ordering, which is the standard ordering for the front end. (Most significant bit first on Suns; least

significant bit first on VAXes.) In Lisp/Paris this is a keyword argument.

Result — The array format descriptor specified.

Context — This is a front-end operation. It does not depend on the value of the *context-flag*.

---

The return value is a format descriptor for arrays; it can be passed to any array transfer instruction as the value of *format*. CM:fe-array-format provides the most generality in specifying an array format for tranfers. More specific descriptors may be obtained with CM:fe-packed-array-format and CM:fe-structure-array-format.

The value of *cm-element-size* defines the unit of measure for the *fe-offset-vector* argument to the CM:read-from-news-array and CM:write-to-news-array instructions.

The value of *array-element-size* defines the unit of measure for the *fe-dimension-vector* argument to the CM:read-from-news-array and CM:write-to-news-array instructions.

If *cm-element-size* is less than *array-element-size*, a packed transfer is specified. That is, multiple Connection Machine array elements are packed into each front-end array element. If *cm-element-size* is greater than *array-element-size*, an extended-element array is specified. That is, more than one front-end array element is used to store each Connection Machine array element.

For most arrays, the value of *stride* is 1. For packed array transfers, *stride* must be 1. For extended-element array transfers, the stride must be large enough to ensure that consecutive elements do not overlap on the front end. To read or write every other (non-packed, non-extended) front-end array element, use a *stride* value of 2.

For a normal (non-packed, non-extended) array transfer, specify *ordering* as a null value.

A packed format with :lsb-first ordering stores the Connection Machine element with the smallest coordinates in the least significant bits of the array element. A packed format with :msb-first ordering stores the CM element with the largest coordinates in the most significant bits of the front-end array.

An extended-element format with :lsb-first ordering stores the low-order bits of the Connection Machine element in the front-end array location with the smallest coordinate. An extended-element format with :msb-first ordering stores the high-order bits of the CM element in the front-end array location with the smallest coordinate.

# F-COSH

Calculates, in each selected processor, the hyperbolic cosine of the floating-point source field value and stores it in the floating-point destination field.

---

**Formats**

CM:f-cosh-1-1L *dest/source*, *s*, *e*
CM:f-cosh-2-1L *dest*, *source*, *s*, *e*

**Operands**

| | |
|---|---|
| *dest* | The floating-point destination field. |
| *source* | The floating-point source field. |
| *s*, *e* | The significand and exponent lengths for the *dest* and *source* fields. The total length of an operand in this format is $s + e + 1$. |

**Overlap** The *source* field must be either disjoint from or identical to the *dest* field. Two floating-point fields are identical if they have the same address and the same format.

**Flags** *overflow-flag* is set if floating-point overflow occurs; otherwise it is unaffected.

**Context** This operation is conditional. The destination and flag may be altered only in processors whose *context-flag* is 1.

---

**Definition**

For every virtual processor $k$ in the *current-vp-set* do
  if *context-flag*$[k] = 1$ then
    *dest*$[k] \leftarrow \cosh$ *source*$[k]$
    if ⟨overflow occurred in processor $k$⟩ then *overflow-flag*$[k] \leftarrow 1$

The hyperbolic cosine of the value of the *source* field is stored into the *dest* field.

# CREATE-DETAILED-GEOMETRY

Creates a new geometry given detailed information about how the grid is laid out.

For most applications, the simpler CM:create-geometry instruction is recommended over this one. Use CM:create-detailed-geometry only to tune the performance of an application with stable, known inter-processor communication patterns.

See also CM:intern-detailed-geometry and CM:intern-geometry.

---

**Formats** result ← CM:create-detailed-geometry *axis-descriptor-array, [rank]*

Operands

*axis-descriptor-array* A front-end vector of descriptors for the grid axes. In the C interface, the elements of the *axis-descriptor-array* must be of type CM_axis_descriptor_t, that is, they must be pointers to structures of type CM_axis_descriptor.

In the Lisp interface, the *axis-descriptor-array* may be either a list of descriptors or an array of descriptors.

*rank* An unsigned integer, the rank (number of dimensions) of the geometry being created. This must be in between 1 and CM:*max-geometry-rank*, inclusive. This argument is not provided when calling Paris from Lisp.

Result A geometry-id, identifying the newly created geometry. This is of type CM_geometry_id_t in C, of type CM:geometry-id in Lisp, and an integer in Fortran.

Context This operation is unconditional. It does not depend on the *context-flag*.

---

CM:create-detailed-geometry takes an array of axis descriptors, one for each axis. The operation returns a geometry-id, which may then be used to create a VP set or to respecify the geometry of an existing VP set.

Each axis descriptor specified by CM:axis-descriptor-array is a structure describing one NEWS axis in some detail. Most of the descriptor components are unsigned integers, but the value of the *ordering* component is different. From Lisp, the *ordering* component must be either :news-order or :send-order. From C or Fortran, it must be either CM_news_order or CM_send_order.

The C definitions of the type of the ordering component and of the axis descriptor are shown below. Notice that the elements of the *axis_descriptor_array* must be pointers to type struct CM_axis_descriptor.

```
typedef enum {CM_news_order, CM_send_order } CM_axis_order_t;

typedef struct CM_axis_descriptor {
  unsigned length;
  unsigned weight;
  CM_axis_order_t ordering;
  unsigned char on_chip_bits;
  unsigned char off_chip_bits;
} * CM_axis_descriptor_t;
```

Actually, this structure has other components as well. C code should use the definition of CM_axis_descriptor from the cmtypes.h include file.

The Fortran/Paris interface defines CM_axis_descriptor as an array:

```
INTEGER RANK,DESCRIPTOR_ARRAY(7,RANK)
```

The elements of each Fortran axis descriptor are defined such that:

$DESCRIPTOR\_ARRAY(1, I)$ is the length of axis $I$
$DESCRIPTOR\_ARRAY(2, I)$ is the weight of axis $I$
$DESCRIPTOR\_ARRAY(3, I)$ is the ordering of axis $I$
$DESCRIPTOR\_ARRAY(4, I)$ is the on-chip bits of axis $I$
$DESCRIPTOR\_ARRAY(6, I)$ is the off-chip bits of axis $I$

Thus CM:axis-descriptor-array is, in Fortran, an array of axis descriptor arrays.

The Lisp definitions of the type of the ordering component and of the axis descriptor are shown below.

```
(deftype cm:axis-order () '(member :news-order :send-order))

(defstruct CM:axis-descriptor
  (length 0) (weight 0) (ordering :news-order)
  (on-chip-bits 0) (off-chip-bits 0))
```

The *axis-descriptor-array* operand must be created by first making one axis descriptor for each axis and then using these to assign values to the array elements. An example in C is given below. Notice that *axis1* and *axis2* are *pointers* to axis descriptor structures and that the descriptor structures are zeroed before any values are assigned.

```
CM_geometry_id_t  my_geometry;
CM_axis_descriptor_t  my_geometry_axes[2];
CM_axis_descriptor_t  axis1, axis2;
```

```
axis1 = malloc(sizeof(struct CM_axis_descriptor));
axis2 = malloc(sizeof(struct CM_axis_descriptor));
bzero(axis1, sizeof(struct CM_axis_descriptor));
bzero(axis2, sizeof(struct CM_axis_descriptor));
axis1->length = 128;
axis2->length = 256;
axis1->weight = 5;
axis2->weight = 10;
axis1->ordering = CM_news_order;
axis2->ordering = CM_news_order;

my_geometry_axes[0] = axis1;
my_geometry_axes[1] = axis2;
my_geometry = CM_create_detailed_geometry(my_geometry_axes, 2);
```

The following example specifies the same axes, descriptor array, and geometry in Lisp. Notice that the constructor CM:make-axis-descriptor is used.

```
(setq my-geometry-axes make-array(2))
(setq axis1
 (CM:make-axis-descriptor :length 128 :weight 5
  :ordering :news-order))
(setq axis2
 (CM:make-axis-descriptor :length 256 :weight 10
  :ordering :news-order)))
(setf (aref my-geometry-axes 0) axis1)
(setf (aref my-geometry-axis 1) axis2)
(setq my-geometry (CM:make-detailed-geometry my-geometry-axes 2)
```

Once the geometry has been created, the user may destroy the descriptors and the array used to provide axis information. All necessary information is copied out of these structures as the geometry is created.

The "length" component of an axis descriptor specifies the length of the axis; it must be a power of two.

The "weight" component of the axis descriptors specifies the relative frequency of interprocessor communication along different axes. For instance, in the above example it is assumed that communication occurs about half as often along *axis1*, which is given a weight of 5, as along *axis2*, which is given a weight of 10. Only the relative values of the weight components matter. The same communication traffic could be specified with weights of 1 and 2, or of 3 and 6. If all weights are 1, it is assumed that all axes are used equally frequently.

## CREATE-DETAILED-GEOMETRY

Given a set of weight components, Paris lays out the hypercube grid for optimal performance. Virtual processors are mapped onto the physical hypercube in a pattern that exploits the fact that communication is especially rapid among virtual processors within the same physical processor and among virtual processors within the same physical chip.

The "ordering" component of an axis descriptor specifies how NEWS coordinates are mapped onto physical processors for that axis. The value :news-order specifies the usual embedding of the grid into the hypercube such that processors with adjacent NEWS coordinates are in fact neighbors within the hypercube. The value :send-order specifies that, if processor A has a smaller NEWS coordinate than processor B, then A also has a smaller send-address than B. This ordering is rarely used. However, :send-order ordering *is* useful for specific applications such as FFT.

**Be careful:** All grid-oriented operations may be used only on axes with :news-order ordering. This includes scans, spreads, reductions, and the get-from-news and send-to-news instructions.

If the "weight" components are all 1, then the mapping of virtual to physical processors can be specified with the "on-chip-bits" and "off-chip-bits" components of the axis descriptors. This is not recommended. To tune performance for communication, use the weight component.

# CREATE-GEOMETRY

Creates a new geometry given the grid axis lengths. See also CM:intern-geometry.

---

**Formats** result ← CM:create-geometry *dimension-array, [rank]*

Operands

*dimension-array* A front-end vector of unsigned integer lengths of the grid axes. In the Lisp interface, this may be a list of dimension lengths instead of an array of dimension lengths, at the user's option.

*rank* An unsigned integer, the rank (number of dimensions) of the *dimension-array*. This must be inbetween 1 and CM:*max-geometry-rank*, inclusive. This argument is not provided when calling Paris from Lisp.

Result A geometry-id, identifying the newly created geometry.

Context This operation is unconditional. It does not depend on the *context-flag*.

---

The *dimension-array* must be a one-dimensional array of nonnegative integers; each must be a power of two. The product of all these integers must be a multiple of the number of physical processors attached for use by this process.

This operation returns a geometry-id for a newly created geometry whose dimensions are specified by the *dimension-array*. The length of axis $j$ of the resulting geometry will be equal to *dimension-array*$[j]$. Such a geometry-id may then be used to create a VP set, or to respecify the geometry of an existing VP set.

The geometry will be laid out so as to optimize performance under the assumption that the axes are used equally frequently for NEWS communication. The operation CM:create-detailed-geometry may be used instead to get more precise control over layout for performance tuning.

Once the geometry has been created, the user may destroy the array used to provide the dimension information. All necessary information is copied out of this array as the geometry is created.

# FE-PACKED-ARRAY-FORMAT

This front-end instruction returns an array format descriptor for a packed front-end array format. A format descriptor may be used as the *format* argument to any array transfer instruction, although this is not required.

See also CM:fe-array-format and CM:fe-structure-array-format.

---

**Formats** result ← CM:fe-packed-array-format *cm-element-size*, *[array-element-size]*

Operands *cm-element-size* A signed integer immediate operand to be used as the number of bits each Connection Machine element occupies in the front-end array. This must be a power of two between 1 and 128.

*array-element-size* A signed integer immediate operand to be used as the number of bits in each front-end array element. This must be a power of two between 1 and 128.
In Lisp/Paris, this argument is optional. If not specified, it defaults to the actual front-end element size or, if the front-end array elements are general (i.e., of type t), *array-element-size* defaults to the value of *cm-element-size*.

Result The array format descriptor specified.

Context This is a front-end operation. It does not depend on the value of the *context-flag*.

---

The return value is a format descriptor for packed arrays; it can be passed to any array transfer instruction. In this format, multiple Connection Machine array elements are packed into each front-end array element during array transfers in either direction between the Connection Machine and the front-end computer.

By using this instruction, it is also possible to specify an extended-element front-end array format. In an extended-element format, each CM element is stored in multiple front-end array elements.

The value of *cm-element-size* defines the unit of measure for the *fe-offset-vector* argument to the CM:read-from-news-array and CM:write-to-news-array instructions.

The value of *array-element-size* defines the unit of measure for the argument *fe-dimension-vector* to the CM:read-from-news-array and CM:write-to-news-array instructions.

The number of Connection Machine elements packed into each front-end array element is the ratio of *array-element-size* to *cm-element-size*. If *array-element-size* is larger than

*cm-element-size*, multiple Connection Machine elements are packed into each front-end array element. Alternatively, if *array-element-size* is smaller than *cm-element-size*, each CM element is stored in more than one front-end array element.

The ordering of the packing defaults to the standard ordering for the front end. For example, on a VAX the Connection Machine element with the smallest coordinates is put into the least significant bits of the front-end array element. On a Sun, the Connection Machine element with the largest coordinates is put into the least significant bits of the front-end array element.

# C-READ-FROM-NEWS-ARRAY

Copies a field within a set of processors forming a subarray of the NEWS grid into a subarray (of the same shape) of an array in the memory of the front end. Both the source and destination values are treated as complex numbers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats** CM:c-read-from-news-array-1L *front-end-array, fe-offset-vector, cm-start-vector, cm-end-vector, cm-axis-vector, source, s, e, [fe-rank, fe-dimension-vector, format]*

**Operands**

*front-end-array* A front-end array (possibly multidimensional) of complex data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers specifying NEWS axes.

*source* The complex source field.

*s*, *e* The significand and exponent lengths for the *source* field. The total length of an operand in this format is $2(s + e + 1)$.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

**Context** This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of the NEWS grid into a similarly shaped subblock of an array in the front end. Complex number values are copied from the Connection Machine processors to the specified *front-end-array*.

The *source* parameter specifies the memory address within each processor of the field to be copied.

The *front-end-array* parameter specifies the front-end destination array into which one element from each processor specified by *source* is copied.

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the source field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) The front-end array is filled in row major order. That is, the last dimension varies fastest. When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element to receive Connection Machine data. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to copy to the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to copy to the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine source field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the source field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, from C or Fortran, one of the following predefined complex *format* values may be used: CM_complex_float_single or CM_complex_float_double. For complex data types in C, two front-end elements are used for each Connection Machine element.

When calling Paris from Lisp, the *format* parameter is a keyword argument; for complex transfers, only arrays of type t may be used.

**Definition** For all $i$ such that $0 \leq i < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}} \leftarrow source[k_i]$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do

for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do

for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do

$\ddots$

for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$front\text{-}end\text{-}array_{offset\text{-}vector_0 + s_0, offset\text{-}vector_1 + s_1, \ldots, offset\text{-}vector_{rank-1} + s_{rank-1}} \leftarrow source[k_{s_0, s_1, \ldots, s_{rank-1}}]$$

# F-READ-FROM-NEWS-ARRAY

Copies a field within a set of processors forming a subarray of the NEWS grid into a subarray (of the same shape) of an array in the memory of the front end. Both the source and destination values are treated as floating-point numbers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats**

CM:f-read-from-news-array-1L *front-end-array*, *fe-offset-vector*, *cm-start-vector*, *cm-end-vector*, *cm-axis-vector*, *source*, *s*, *e*, *[fe-rank*, *fe-dimension-vector*, *format]*

**Operands**

*front-end-array* A front-end array (possibly multidimensional) of floating-point data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*source* The floating-point source field.

*s*, *e* The significand and exponent lengths for the *source* field. The total length of an operand in this format is $s + e + 1$.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

**Context** This operation is unconditional. It does not depend on the *context-flag*.

---

## READ-FROM-NEWS-ARRAY

This operation copies a rectangular subblock of the NEWS grid into a similarly shaped subblock of an array in the front end. Floating-point number values are transferred from the Connection Machine processors to the specified *array*.

The *source* parameter specifies the memory address within each processor of the field to be copied.

The *front-end-array* parameter specifies the front-end destination array into which one element from each processor specified by *source* is copied.

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the source field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) The front-end array is filled in row major order. That is, the last dimension varies fastest. When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element to receive Connection Machine data. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to copy to the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to copy to the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine source field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the source field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, one of the predefined floating-point *format* values may be used. These are CM_float_single or CM_float_double from C or Fortran, and :float-single or :float-double from Lisp.

When calling Paris from Lisp, the *format* parameter is a keyword argument. If not specified, it defaults based on the element type of the front-end array or, if the array is of type t, based on the type and size of the Connection Machine field.

**Definition** For all $i$ such that $0 \leq i < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}} \leftarrow source[k_i]$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$front\text{-}end\text{-}array_{offset_0+s_0, offset_1+s_1, \ldots, offset_{rank-1}+s_{rank-1}} \leftarrow source[k_{s_0, s_1, \ldots, s_{rank-1}}]$$

# S-READ-FROM-NEWS-ARRAY

Copies a field within a set of processors forming a subarray of the NEWS grid into a subarray (of the same shape) of an array in the memory of the front end. Both the source and destination values are treated as signed integers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats**

CM:s-read-from-news-array-1L *front-end-array*, *fe-offset-vector*, *cm-start-vector*, *cm-end-vector*, *cm-axis-vector*, *source*, *len*, *[fe-rank*, *fe-dimension-vector*, *format]*

**Operands**

*front-end-array* A front-end array (possibly multidimensional) of signed integer data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*source* The signed integer source field.

*len* The length of the *source* field. This must be no smaller than 2 but no greater than CM:*maximum-integer-length*.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

**Context** This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of the NEWS grid into a similarly shaped subblock of an array in the front end. Signed integer values are transferred from the Connection Machine processors to the specified *array*.

The *source* parameter specifies the memory address within each processor of the field to be copied.

The *front-end-array* parameter specifies the front-end destination array into which one element from each processor specified by *source* is copied.

When calling Paris from Lisp, the array may be either a general array (of type t) containing signed integers, or a specialized integer-element array (such as an array of type (unsigned-byte 8)).

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the source field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) The front-end array is filled in row major order. That is, the last dimension varies fastest. When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element to receive Connection Machine data. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to copy to the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to copy to the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine source field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the source field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, one of the predefined signed *format* values may be used.

From C or Fortran a value of CM_8_bit, CM_16_bit, or CM_32_bit specifies an unpacked front-end array while CM_2_bit_packed, or CM_4_bit_packed specifies a front-end array in which several CM elements are packed into each array element. From Lisp, the predefined signed format keywords are :8-bit, :16-bit, :32-bit, :2-bit-packed, and :4-bit-packed.

When calling Paris from Lisp, the *format* parameter is a keyword argument. If not specified, it defaults based on the element type of the front-end array or, if the array is of type t, based on the type and size of the Connection Machine field.

**Definition** For all $i$ such that $0 \leq i < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}} \leftarrow source[k_i]$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$front\text{-}end\text{-}array_{offset_0+s_0, offset_1+s_1, \ldots, offset_{rank-1}+s_{rank-1}} \leftarrow source[k_{s_0, s_1, \ldots, s_{rank-1}}]$$

# U-READ-FROM-NEWS-ARRAY

Copies a field within a set of processors forming a subarray of the NEWS grid into a subarray (of the same shape) of an array in the memory of the front end. Both the source and destination values are treated as unsigned integers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats** CM:u-read-from-news-array-1L *front-end-array, fe-offset-vector, cm-start-vector, cm-end-vector, cm-axis-vector, source, len, [fe-rank, fe-dimension-vector, format]*

**Operands**

*front-end-array* A front-end array (possibly multidimensional) of unsigned integer data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*source* The unsigned integer source field.

*len* The length of the *source* field. This must be non-negative and no greater than CM:*maximum-integer-length*.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

**Context** This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of the NEWS grid into a similarly shaped subblock of an array in the front end. Unsigned integer values are transferred from the Connection Machine processors to the specified *array*.

The *source* parameter specifies the memory address within each processor of the field to be copied.

The *front-end-array* parameter specifies the front-end destination array into which one element from each processor specified by *source* is copied.

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the source field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) The front-end array is filled in row major order. That is, the last dimension varies fastest. When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element to receive Connection Machine data. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to copy to the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to copy to the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine source field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the source field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, one of the predefined unsigned *format* values may be used.

From C or Fortran a value of CM_8_bit, CM_16_bit, or CM_32_bit specifies an unpacked front-end array while CM_1_bit_packed, CM_2_bit_packed, or CM_4_bit_packed specifies a front-end array in which several CM elements are packed into each array element. From Lisp, the predefined unsigned format keywords are :8-bit, :16-bit, :32-bit, :1-bit-packed, :2-bit-packed, and :4-bit-packed.

When calling Paris from Lisp, the *format* parameter is a keyword argument. If not specified, it defaults based on the element type of the front-end array or, if the array is of type t, based on the type of the CM field.

**Definition** For all $i$ such that $0 \leq i < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} \textit{make-news-coordinate}(axis_j, start_j + s_{i,j})$$

$$\textit{front-end-array}_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}} \leftarrow source[k_i]$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} \textit{make-news-coordinate}(axis_j, start_j + s_j)$$

$$\textit{front-end-array}_{offset_0 + s_0, offset_1 + s_1, \ldots, offset_{rank-1} + s_{rank-1}} \leftarrow source[k_{s_0, s_1, \ldots, s_{rank-1}}]$$

# STORE-flag

Conditionally stores a flag bit into memory.

---

**Formats**

CM:store-test *dest*
CM:store-overflow *dest*

Operands *dest* The destination bit (a one-bit field).

Context This operation is conditional. The destination may be altered only in processors whose *context-flag* is 1.

---

**Definition** For every virtual processor $k$ in the *current-vp-set* do
if *context-flag*$[k] = 1$ then
*dest*$[k] \leftarrow$ *flag*$[k]$
where *flag* is *test-flag* or *overflow-flag*, as appropriate.

Within each processor, the indicated flag for that processor is stored into memory.

# FE-STRUCTURE-ARRAY-FORMAT

This instruction returns an array format descriptor for a particular slot in an array of structures. A format descriptor may be passed to any array transfer instruction to specify a front-end array format, although this is not required. See also CM:fe-array-format and CM:fe-packed-array-format.

This instruction is not provided for the Lisp interface to Paris.

---

**Formats** result ← CM:fe-structure-array-format *cm-element-byte-size*, *structure-byte-size*

Operands *cm-element-byte-size* A signed integer immediate operand to be used as the number of bytes each Connection Machine element occupies in the front-end array. This must be a power of two between 1 and 16.

*structure-byte-size* A signed integer immediate operand to be used as the length of the front-end structure in bytes. This may be any positive integer.

Result The array format descriptor specified.

Context This is a front-end operation. It does not depend on the value of the *context-flag*.

---

The return value is a format descriptor for a front-end array of structures. Such a format descriptor can be passed to any of the CM array transfer instructions in order to allow transfers in either direction between CM fields and a front-end array of structures. If this is done, one CM element per selected processor is copied into, or receives data from, the specified slot across an array of structures on the front end.

Values for both *cm-element-byte-size* and *cm-structure-byte-size* may be obtained by calls to sizeof(...).

The value of *cm-element-byte-size* specifies the length of the structure slot in bytes. It also defines the unit of measure for the *fe-offset-vector* argument to the CM:read-from-news-array and CM:write-to-news-array instructions.

The value of *structure-byte-size* specifies the length of the entire stucture in bytes. It also defines the unit of measure for the argument *fe-dimension-vector* to the CM:read-from-news-array and CM:write-to-news-array instructions.

If a slot other than the first slot in the front-end structure is the destination of a CM:read-from-news-array or the source for a CM:write-to-news-array transfer instruction, then a pointer to that slot must be provided as the value of *front-end-array*. This is a bit tricky. The

pointer must identify the location of the chosen slot in the structure that is the first element of the array of structures.

Here is an example in C.

```
#define n_foos 256

/* declare array of structure foo */
struct foo { int a; double b; char c; } fooarray[n_foos];

/* declare the format */
CM_array_format_t foo_format;

/* declare an offset for the 'b' slot of struct foo */
/* this is a pointer to a double - b is a double */
double *bslot_pointer;

/* lots of other declarations etc. in here */
...

/* create format descriptor for foo.b */
foo_format = CM_structure_array_format(sizeof(double), sizeof(struct foo));

/* create pointer offset to slot b of struct foo */
bslot_pointer = &fooarray[0].b;

/* store src-field values in slot b of each foo struct in foo_array */
/* all variables xxxx_vector should be self explanatory */

CM_f_read_from_news_array_1L(bslot_pointer, offset_vector,
                             start_vector, end_vector, axis_vector,
                             src_field, 23, 8, rank,
                             dimension_vector, foo_format);
```

Slot b of each foo structure in the array foo_array receives a copy of the value stored in the corresponding CM *src-field* processor.

The value of bslot_pointer is a pointer to the b slot of the first foo structure in foo_array. Given this starting place, foo_format indicates how many bytes must be skipped between b slots.

For further examples, refer to the manual entitled *Introduction to Programming in C/Paris.*

# F-SUB-MULT

Calculates a value $(x - a)b$ and places it in the destination.

---

**Formats**

| | |
|---|---|
| CM:f-sub-mult-1L | *dest*, *source1*, *source2*, *source3*, *s*, *e* |
| CM:f-sub-const-mult-1L | *dest*, *source1*, *source2-value*, *source3*, *s*, *e* |
| CM:f-sub-mult-const-1L | *dest*, *source1*, *source2*, *source3-value*, *s*, *e* |
| CM:f-sub-const-mult-const-1L | *dest*, *source1*, *source2-value*, *source3-value*, *s*, *e* |

**Operands**

*dest* The floating-point destination field.

*source1* The floating-point first source (minuend) field.

*source2* The floating-point second source (subtrahend) field.

*source2-value* A floating-point immediate operand to be used as the second source (subtrahend).

*source3* The floating-point third source (multiplier) field.

*source3-value* A floating-point immediate operand to be used as the third source (multiplier).

*s*, *e* The significand and exponent lengths for the *dest*, *source1*, *source2*, and *source3* fields. The total length of an operand in this format is $s + e + 1$.

**Overlap** The fields *source1*, *source2*, and *source3* may overlap in any manner. Each of them, however, must be either disjoint from or identical to the *dest* field. Two floating-point fields are identical if they have the same address and the same format. It is permissible for all the fields to be identical.

**Flags** *overflow-flag* is set if floating-point overflow occurs; otherwise it is unaffected.

**Context** This operation is conditional. The destination and flag may be altered only in processors whose *context-flag* is 1.

---

**Definition**

For every virtual processor $k$ in the *current-vp-set* do
  if *context-flag*$[k] = 1$ then
    *dest*$[k] \leftarrow$ (*source1*$[k]$ − *source2*$[k]$) × *source3*$[k]$
if ⟨overflow occurred in processor $k$⟩ then *overflow-flag*$[k] \leftarrow 1$

The operand *source2* is subtracted from *source1*, treating them as floating-point numbers, and then the difference is multiplied by a third operand *source3*. The result is stored

# WARM-BOOT

This operation is used by the Lisp/Paris interface to reinitialize the Connection Machine system without disturbing user memory.

---

**Formats** CM:warm-boot

Context This operation is unconditional. It does not depend on the *context-flag*.

---

This operation clears error status indicators for the attached Connection Machine hardware. It also clears the IFIFO and OFIFO in the bus interface and possibly loads fresh microcode into the attached microcontroller(s). The user memory areas in the Connection Machine system are not disturbed, but are checked for errors; any memory errors are reported. Certain system memory areas in the Connection Machine system are reinitialized, but the state of the pseudo-random number generator is not altered and the system lights-display mode is not altered. The intent is to recover from an error condition while preserving as much of the machine state as possible.

The facility for warm-booting Connection Machine hardware is provided in different ways in the Lisp/Paris interface (on the one hand) and the C/Paris and Fortran/Paris interfaces (on the other hand).

In the Lisp/Paris interface, CM:warm-boot is a function.

This operation takes no arguments and returns no values. It signals an error if the warm-boot process was not successful.

There are two sets of initializations, kept in the variables CM:*before-warm-boot-initializations* and CM:*after-warm-boot-initializations*, that are evaluated before and after anything else occurs.

In the C/Paris and Fortran/Paris interfaces, there is no CM:warm-boot operation. Instead, a related operation called CM:init is used.

# C-WRITE-TO-NEWS-ARRAY

Copies a subarray of an array in the memory of the front end into a field within a set of processors forming a subarray (of the same shape) of the NEWS grid. Both source and destination values are treated as complex numbers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats**

CM:c-write-to-news-array-1L *front-end-array*, *fe-offset-vector*, *cm-start-vector*, *cm-end-vector*, *cm-axis-vector*, *dest*, *s*, *e*, *[fe-rank*, *fe-dimension-vector*, *format]*

**Operands**

*front-end-array* A front-end array (possibly multidimensional) of complex data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*dest* The complex destination field.

*s*, *e* The significand and exponent lengths for the *dest* field. The total length of an operand in this format is $2(s + e + 1)$.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

**Context** This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of an array in the front end into a similarly shaped subblock of the NEWS grid. Complex number values are transferred from the specified *front-end-array* to the Connection Machine processors.

The *dest* parameter specifies the memory address within each processor of the field into which the data is stored.

The *front-end-array* parameter specifies the front-end source array from which one element is copied to each processor specified by *dest*.

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the destination field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element transferred to the Connection Machine. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to receive data from the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to receive data from the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine destination field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the destination field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, from C or Fortran, one of the following predefined complex *format* values may be used: CM_complex_float_single or CM_complex_float_double. For complex data types in C, two front-end elements are used for each Connection Machine element.

When calling Paris from Lisp, the *format* parameter is a keyword argument; for complex transfers only arrays of type t may be used

**Definition** For all $i$ such that $0 \leq j < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$dest[k_i] \leftarrow front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}}$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$dest[k_{s_0, s_1, \ldots, s_{rank-1}}] \leftarrow$$
$$front\text{-}end\text{-}array_{offset_0 + s_0, offset_1 + s_1, \ldots, offset_{rank-1} + s_{rank-1}}$$

# F-WRITE-TO-NEWS-ARRAY

Copies a subarray of an array in the memory of the front end into a field within a set of processors forming a subarray (of the same shape) of the NEWS grid. Both source and destination values are treated as floating-point numbers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats**

CM:f-write-to-news-array-1L *front-end-array*, *fe-offset-vector*, *cm-start-vector*, *cm-end-vector*, *cm-axis-vector*, *dest*, *s*, *e*, *[fe-rank*, *fe-dimension-vector*, *format]*

**Operands**

*front-end-array* A front-end array (possibly multidimensional) of floating-point data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*dest* The floating-point destination field.

*s*, *e* The significand and exponent lengths for the *dest* field. The total length of an operand in this format is $s + e + 1$.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

**Context** This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of an array in the front end into a similarly shaped subblock of the NEWS grid. Floating-point number values are transferred from the specified *array* to the Connection Machine processors.

The *dest* parameter specifies the memory address within each processor of the field into which the data is stored.

The *front-end-array* parameter specifies the front-end source array from which one element is copied to each processor specified by *dest*.

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the destination field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element transferred to the Connection Machine. The lengths of the above three vector arguments are measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to receive data from the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to receive data from the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine destination field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the destination field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, one of the predefined floating-point *format* values may be used. These are CM_float_single or CM_float_double from C or Fortran, and :float-single or :float-double from Lisp.

When calling Paris from Lisp, the *format* parameter is a keyword argument. If not specified, it defaults based on the element type of the front-end array or, if the array is of type t, based on the type of the Connection Machine field.

**Definition** For all $i$ such that $0 \leq j < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$dest[k_i] \leftarrow front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}}$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0,s_1,\ldots,s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$dest[k_{s_0,s_1,\ldots,s_{rank-1}}] \leftarrow$$
$$front\text{-}end\text{-}array_{offset_0+s_0, offset_1+s_1, \ldots, offset_{rank-1}+s_{rank-1}}$$

# S-WRITE-TO-NEWS-ARRAY

Copies a subarray of an array in the memory of the front end into a field within a set of processors forming a subarray (of the same shape) of the NEWS grid. Both the source and destination values are treated as signed integers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats**

CM:s-write-to-news-array-1L *front-end-array*, *fe-offset-vector*, *cm-start-vector*, *cm-end-vector*, *cm-axis-vector*, *dest*, *len*, *[fe-rank*, *fe-dimension-vector*, *format]*

Operands

*front-end-array* A front-end array (possibly multidimensional) of signed integer data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*dest* The signed integer destination field.

*len* The length of the *dest* field. This must be no smaller than 2 but no greater than CM:*maximum-integer-length*.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

Context

This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of an array from the front end into a similarly shaped subblock of the NEWS grid. Signed integer values are transferred from the specified *array* to the Connection Machine processors.

The *dest* parameter specifies the memory address within each processor of the field into which the data is stored.

The *front-end-array* parameter specifies the front-end source array from which one element is copied to each processor specified by *dest*.

When calling Paris from Lisp, the array may be either a general array (of type t) containing signed integers, or a specialized integer-element array (such as an array of type (unsigned-byte 8)).

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the destination field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element transferred to the Connection Machine. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to receive data from the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to receive data from the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine destination field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the destination field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, one of the predefined signed *format* values may be used.

From C or Fortran a value of CM_8_bit, CM_16_bit, or CM_32_bit specifies an unpacked front-end array while CM_1_bit_packed, CM_2_bit_packed, or CM_4_bit_packed specifies a front-end array in which several CM elements are packed into each array element. From Lisp, the predefined signed format keywords are :8-bit, :16-bit, :32-bit, :1-bit-packed, :2-bit-packed, and :4-bit-packed.

When calling Paris from Lisp, the *format* parameter is a keyword argument. If not specified, it defaults based on the element type of the front-end array or, if the array is of type t, based on the type of the Connection Machine field.

**Definition** For all $i$ such that $0 \leq j < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$dest[k_i] \leftarrow front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}}$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$dest[k_{s_0, s_1, \ldots, s_{rank-1}}] \leftarrow front\text{-}end\text{-}array_{offset_0 + s_0, offset_1 + s_1, \ldots, offset_{rank-1} + s_{rank-1}}$$

# U-WRITE-TO-NEWS-ARRAY

Copies a subarray of an array in the memory of the front end into a field within a set of processors forming a subarray (of the same shape) of the NEWS grid. Both the source and destination values are treated as unsigned integers.

**Note:** The read-from-news-array and write-to-news-array operations do *not* require that the specified CM field be in the current VP set.

---

**Formats**

CM:u-write-to-news-array-1L *front-end-array, fe-offset-vector, cm-start-vector, cm-end-vector, cm-axis-vector, dest, len, [fe-rank, fe-dimension-vector, format]*

Operands

*front-end-array* A front-end array (possibly multidimensional) of unsigned integer data.

*fe-offset-vector* A front-end vector of signed integer subscript offsets for the *front-end-array*.

*cm-start-vector* A front-end vector of signed integer inclusive lower bounds for NEWS indices.

*cm-end-vector* A front-end vector of signed integer exclusive upper bounds for NEWS indices.

*cm-axis-vector* A front-end vector of signed integer numbers indicating NEWS axes.

*dest* The unsigned integer dest field.

*len* The length of the *dest* field. This must be non-negative and no greater than CM:*maximum-integer-length*.

*fe-rank* A signed integer, the rank (number of dimensions) of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*fe-dimension-vector* A front-end vector of signed integer dimensions of the *front-end-array*. This argument is not provided when calling Paris from Lisp.

*format* The array descriptor for *front-end-array*. This is a keyword argument when calling Paris from Lisp.

Context

This operation is unconditional. It does not depend on the *context-flag*.

---

This operation copies a rectangular subblock of an array from the front end into a similarly shaped subblock of the NEWS grid. Unsigned integer values are transferred from the specified *array* to the Connection Machine processors.

The *dest* parameter specifies the memory address within each processor of the field into which data is stored.

The *front-end-array* parameter specifies the front-end source array from which one element is copied to each processor specified by *dest*.

The *fe-rank* parameter specifies the rank of the front-end array and is normally equal to the rank of the destination field geometry. When calling Paris from Lisp, this value can be deduced from the value of *front-end-array* and must not be specified.

The vector arguments are one-dimensional front-end arrays of length *fe-rank*.

The *fe-dimension-vector* parameter specifies the dimensions of the front-end array. These dimensions are measured in units of *array-element-size*, which is implicitly specified by *format*. (See the description of *format* below.) When calling Paris from Lisp, the front-end array dimensions can be deduced from the value of *front-end-array* and must not be specified.

The *fe-offset-vector* parameter contains the coordinate of the first front-end array element transferred to the Connection Machine. The length of this argument is measured in units of *cm-element-size*, which is implicitly specified by *format*. (See the description of *format* below.)

The *cm-start-vector* parameter specifies the coordinate of the first CM element to receive data from the front end. The *cm-end-vector* parameter specifies the coordinate of the last CM element to receive data from the front end.

The *cm-axis-vector* parameter specifies how Connection Machine axes are mapped to front-end array axes. For example, if *cm-axis-vector*$[A] = B$, then axis $A$ of the Connection Machine source field geometry is mapped to axis $B$ of the front-end array. The length of this vector must be equal to the rank of the source field geometry.

The *format* parameter is an array descriptor that specifies the format of the front-end array. An appropriate descriptor may be obtained by a call to CM:array-format, CM:packed-array-format, or CM:structure-array-format. Alternatively, one of the predefined unsigned *format* values may be used.

From C or Fortran a value of CM_8_bit, CM_16_bit, or CM_32_bit specifies an unpacked front-end array while CM_1_bit_packed, CM_2_bit_packed, or CM_4_bit_packed specifies a front-end array in which several CM elements are packed into each array element. From Lisp, the predefined unsigned format keywords are :8-bit, :16-bit, :32-bit, :1-bit-packed, :2-bit-packed, and :4-bit-packed.

When calling Paris from Lisp, the *format* parameter is a keyword argument. If not specified, it defaults based on the element type of the front-end array or, if the array is of type t, based on the type of the Connection Machine field.

**Definition** For all $i$ such that $0 \leq j < \prod_{j=0}^{rank-1} (end_j - start_j)$ do

for all $m$ such that $0 \leq m < rank$ do

$$\text{let } s_{\langle i,m \rangle} = \left\lfloor \frac{i}{\prod_{j=m+1}^{rank-1} (end_j - start_j)} \right\rfloor \bmod (end_m - start_m)$$

$$\text{let } k_i = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_{i,j})$$

$$dest[k_i] \leftarrow front\text{-}end\text{-}array_{s_{\langle i,0 \rangle}, s_{\langle i,1 \rangle}, \ldots, s_{\langle i,rank-1 \rangle}}$$

Another formulation:

For all $s_0$ such that $0 \leq s_0 < (end_0 - start_0)$ do
  for all $s_1$ such that $0 \leq s_1 < (end_1 - start_1)$ do
    for all $s_2$ such that $0 \leq s_2 < (end_2 - start_2)$ do
      $\ddots$
        for all $s_{rank-1}$ such that $0 \leq s_{rank-1} < (end_{rank-1} - start_{rank-1})$ do

$$\text{let } k_{s_0, s_1, \ldots, s_{rank-1}} = \bigvee_{j=0}^{rank-1} make\text{-}news\text{-}coordinate(axis_j, start_j + s_j)$$

$$dest[k_{s_0, s_1, \ldots, s_{rank-1}}] \leftarrow$$
$$front\text{-}end\text{-}array_{offset_0 + s_0, offset_1 + s_1, \ldots, offset_{rank-1} + s_{rank-1}}$$

# F-WRITE-TO-PROCESSOR

Stores an immediate floating-point number operand value into the destination field of a single specified processor.

---

**Formats** CM:f-write-to-processor-1L *send-address-value*, *dest*, *source-value*, *s*, *e*

Operands

*send-address-value* An immediate operand, the send address of a single particular processor.

*dest* The floating-point destination field.

*source-value* A floating-point immediate operand to be used as the source.

*s*, *e* The significand and exponent lengths for the *dest* field. The total length of an operand in this format is $s + e + 1$.

Context This operation is unconditional. It does not depend on the *context-flag*.

---

**Definition** *dest*[*send-address-value*] ← *source-value*

The specified *source-value*, a floating-point number, is stored into the *dest* field of the processor whose send address is the immediate operand *send-address-value*.