The
Connection Machine
System

CM-5 C* Release Notes

Preliminary Documentation for Version 7.1 Beta

February 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, February 1993

****

PRELIMINARY DOCUMENTATION

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines assumes no liability for errors in this document.

This document does not describe any product that is currently available from Thinking Machines Corporation, and Thinking Machines does not commit to implement the contents of this document in any product.

****

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

Contents

Field Test Support

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

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

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

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

CM-5 C* Version 7.1 Beta Release Notes

1 About CM-5 C* Version 7.1 Beta

CM-5 C* Version 7.1 is a new release of the CM-5 C* compiler; it provides support for CM-5s with vector units, as well as for CM-5s with SPARC processors only. CM-5 C* is an implementation of the C* language, as described in the *C* Programming Guide*. Version 7.1 works with CMOST Version 7.2 S2 or later. CMOST Version 7.2 Beta Patch 3 is required to remove some restrictions in support for CMFS calls on CM-5s with vector units.

Section 2 lists features new in Version 7.1. Sections 3–10 contain other information about the CM-5 C* compiler; this information is repeated, with some changes, from the Version 7.0 Beta release notes. Change bars in the margin indicate new or changed information.

To learn about restrictions in this release, see the on-line bug update report, which is by default in the file `/usr/doc/cstar-7.1-beta.bugupdate`; if this file doesn't exist on your system, check with your system administrator.

Note this restriction in the Beta release:

- Segmented rank operations via the `rank` function are not yet supported in programs compiled for the vector units. They will be included in a future CMOST release.

2 New Features in Version 7.1

Version 7.1 adds the features discussed in this section to CM-5 C*.

2.1 Support for Vector Units

As of Version 7.1, CM-5 C* takes advantage of the processing power of vector units, in CM-5 systems that contain them.

2.1.1 New Compiler Options

Use the **-vu** or **-vecunit** option to the **cs** command to specify that you are compiling or linking for a CM-5 with vector units. Use the **-sparc** option to specify that you do not want to use the vector units. If you include one of these options on the **cs** command line, you do not have to include the **-cm5** option. Either **-vu** or **-sparc** will be the default at your site; ask your system administrator. The target for the compilation is indicated by a message that the compiler prints; see Section 5.

Use the **-keep** option to keep an intermediate file with the extension you specify. Choices are:

- **s**, to keep assembly language source file
- **o**, to keep the object file
- **dp**, to keep the DPEAC assembly-language code; the file is named *file*.**pe**.**dp**, where *file* is the name of the C* source code, without the .**cs** extension

Using this option does not inhibit assembly or linking.

Use the **-temp** option to change the location in which C* temporary files are created from the standard location. Issue the **cs** command with the **-dirs** option to find out the standard location; see Section 5.

See also Section 2.5 for a discussion of the new compiler options that support the interface to CMMD.

2.1.2 New Size for physical Shape

On a CM-5 with vector units, the size of the **physical** shape is 4 times the number of processing nodes in your partition (in other words, the size is the number of vector units in your partition).

Note for low-level programming: Because of subgrid restrictions imposed by the current compiler model, the positions of the **physical** shape are not actually instantiated on each vector unit. We expect that eventually variables in the **physical** shape will have one element in each vector unit, but for now all four elements in the **physical** shape on a given node are instantiated in the first vector unit of the node.

2.2 Increased Performance

Version 7.1 provides faster compilation and performance than Version 7.0:

- Compilation should be approximately 30 percent faster.
- Local optimization has been implemented; this should result in better performance.

Note, however, that if you are executing on the vector units, operations that involve **bool**, **char**, and **short** types are slow because of extra overhead required to load and store those types from memory.

2.3 SDA and DataVault Support

As of the Version 7.0 Beta Patch 1 release, you can use the SDA (Scalable Disk Array) and DataVault file system via calls to CMFS library routines.

To use the SDA file system on a CM-5, you must link your program with:

```
-lcmfs_cs -lcmfs_cm5
```

If you have DataVault files created prior to the Version 7.0 Beta Patch 1 release, see the Patch 1 release notes for a discussion of how to convert your files to the format required by the patch.

2.4 Interface to CMX11

A C* interface to CMX11 is available as of CMX11 Version 1.5. See the release notes for CMX11 Version 1.5 for more information.

2.5 Interface to CMMD

A C* interface to CMMD is available as of CMMD Version 3.0. This works only for C* programs compiled to use the vector units. The CMMD 3.0 final documentation will describe this interface.

There are two new options to the `cs` command to support this interface:

- Use the `-node` option to specify that you are compiling your C* program to run on the nodes.
- Use the `-cmmd_root` option to specify the location of the CMMD library if it's not in the standard place. Check with your system administrator for the correct location at your site. To avoid having to specify this option every time you compile a node-level program, you can also set the environment variable `CMMD_ROOT` to the correct pathname.

2.6 Table Lookup

In Version 7.1, CM-5 C* provides an efficient mechanism for parallel lookups into a single table. If you use this mechanism, C* replicates the table once per node or vector unit, rather than in each position of a shape.

To use the table lookup utility, include the file `<cstable.h>`.

The utility consists of four functions:

- Call `CMC_allocate_shared_table` to allocate the table on the nodes. It takes as its argument the size of the table (the total number of elements in the table times their size in bytes), and it returns a pointer to a parallel variable that indicates the table's location on the nodes. Its definition is:

```
void:void *CMC_allocate_shared_table(size_t table_size);
```

It is legal to use the pointer returned by this function only with the other table lookup functions.

- Call **CMC_initialize_shared_table** to put values into the table. Its definition is:

```
void CMC_initialize_shared_table(void:void *table,
                                 const void *values,
                                 size_t table_size);
```

where:

table	is the pointer to the table, returned by **CMC_allocate_shared_table**.
values	is a pointer to the scalar table values.
table_size	is the size of the table in bytes. This is the same size specified to the **CMC_allocate_shared_table** function.

- Call **CMC_lookup_shared_table** to do a lookup in the table on the nodes. Its definition is:

```
void CMC_lookup_shared_table(void:current *result,
                             void:void *table,
                             int:current index,
                             size_t element_size);
```

where:

result	is a pointer to a parallel variable of the current shape that holds the results of the lookup.
table	is the parallel pointer to the table.
index	is a parallel **int** of the current shape; its values are the indices into the table.
element_size	is the size of each element in the table, in bytes.

- Call **CMC_free_shared_table** to deallocate the memory allocated on the nodes to the table. Its definition is:

```
void CMC_free_shared_table(void:void *table);
```

where **table** is the pointer returned by **CMC_allocate_shared_table**.

2.6.1 An Example

In this example, a table of 24 `int`s is allocated and initialized in a 16384-position shape on the nodes. Random numbers are used as the index into the table, and the results of some lookups are printed. Finally, the memory for the table is freed.

```
#include <stdio.h>
#include <stdlib.h>
#include <cstable.h>

int table_data[24] = {
    14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10,
    23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13, 2
};

main()
{
    shape [16384]s;
    int:s index, result;
    void:void *table;
    int i;

    table = CMC_allocate_shared_table(sizeof(table_data));
    CMC_initialize_shared_table(table, table_data,
            sizeof(table_data));

    with(s)
    {
        index = prand() % 24;
        CMC_lookup_shared_table(&result, table, index,
            sizeof(result));
    }

    for(i = 0; i < 20; ++i)
    {
        printf("%d\n", [i]result);
    }

    CMC_free_shared_table(table);
}
```

2.7 Shape Casting

A pointer to a parallel variable contains information about the shape of the parallel variable. Using the new function `CMC_change_pointer_shape` in

CM-5 C*, you can associate a different shape with the parallel variable; you can then treat the parallel variable as if it were in this other shape. Since the pointer exists on the partition manager, this saves the cost of doing communication between shapes on the nodes.

NOTE: Using this new function correctly requires a detailed understanding of the parallel variable's layout in the memory of the nodes. Inappropriate use of the function will cause run-time errors.

Include the file **<csshape.h>** when using **CMC_change_pointer_shape**. The function's definition is:

```
void:void *CMC_change_pointer_shape(void:void *, shape);
```

where **void:void *** is the pointer to the parallel variable, and **shape** is the new shape to be associated with the parallel variable.

For users familiar with Paris on the CM-2/200: This function is comparable to the Paris **CM:make-field-alias** instruction.

2.8 New Names for C* Run-Time Libraries

The names of the C* run-time libraries have changed in Version 7.1. In Version 7.0, the names were **libcs.a** and **libcs_pe.a** (for the nodes). They now depend on the target of the compilation:

- If you specify **-sparc**, the libraries are **libcs_cm5_sparc_sp.a** and **libcs_cm5_sparc_pn.a** (for the nodes).
- If you specify **-vu**, the libraries are **libcs_cm5_vu_sp.a** and **libcs_cm5_vu_pn.a** (for the nodes).
- If you specify **-cmsim**, the library is **libcs_cm5_cmsim.a**.

You need to be aware of this only if you are using a command other than **cs** to link (for example, when linking in C* functions to a CM Fortran program); see Section 5.1 for an example.

3 Differences from CM-200 C*

This section lists differences between CM-5 C* and C* for the CM-2 and CM-200 (referred to as *CM-200 C**).

3.1 Restriction on Shape Sizes Removed

The CM-200 C* restrictions on shape extents are not present in CM-5 C*. The sizes of a shape's dimensions need not be powers of 2, and the total number of positions in the shape need not be a multiple of the number of physical processors that the C* program is using. The only restriction is that the size of each dimension must be greater than 0.

3.2 Different Size for Parallel bools

On the CM-5, parallel `bool`s occupy 1 byte of storage, not 1 bit, as on the CM-2 and CM-200. (This change is necessary because CM-5 memory is not bit-addressable.) The semantics of using `bool`s remain the same; you need not change an existing program to deal with the new size. Memory usage will go up on the CM-5, however. Also note that on the CM-5, `boolsizeof` gives a size in bytes, and is therefore exactly like `sizeof`.

3.3 Programs Can't Call Paris

CM-5 C* programs can't call Paris routines (because there is no Paris on the CM-5). CM-2-specific header files such as `<cm/paris.h>` are not available on the CM-5.

3.4 Improved Performance of Parallel Right Indexing

Parallel indexing into parallel arrays performs better in CM-5 C* than it does in CM-200 C*.

3.5 New *= and /= Reduction Operators

CM-5 C* implements the *= and /= parallel-to-scalar reduction operators.

As a binary reduction operator, *= multiplies the values of the active elements of the parallel RHS by the value of the scalar LHS and assigns it to the LHS. As a unary operator, it returns the product of the active elements of the parallel variable.

As a binary reduction operator, /= divides the value of the scalar LHS by the product of the parallel RHS and assigns the result to the scalar LHS. When it is used as a unary operator, it returns the reciprocal of the product of all active positions in the parallel variable.

3.6 ANSI Compliance

The CM-5 C* compiler is generally compliant with the ANSI standard. This means that the CM-5 C* compiler will reject some programs that previously compiled without error.

3.7 Parallel enums Are Supported

Unlike the CM-200 C* compiler, CM-5 C* supports parallel **enums**. For example, this code:

```
enum color { red, blue, green };
enum color:ShapeA parallel_color;
```

declares the parallel variable **parallel_color** to be of the enumeration type **color**. You can then assign a value to **parallel_color** as follows:

```
parallel_color = red;
```

This assigns the value **red** to every element of the parallel variable **parallel_color**.

3.8 Limitations on Parallel Unions Removed

The limitations on parallel unions discussed on page 60 of the *C* Programming Guide* are removed in CM-5 C*. Note, however, that taking advantage of the removal of these limitations may make your program nonportable.

3.9 New Versions of read_from_pvar and write_to_pvar

CM-5 C* overloads the communication functions **read_from_pvar** and **write_to_pvar** for parallel data of any length.

The definition of **read_from_pvar** is:

```
void read_from_pvar (
    void *destp,
    void:current *sourcep,
    int length);
```

where **destp** is a pointer to the scalar array to which the values are to be written, **sourcep** is a pointer to the parallel data, and **length** is the length, in **bools**, of the data pointed to by **sourcep**.

And the definition of **write_to_pvar** is:

```
void write_to_pvar (
    void:current *destp,
    void *sourcep,
    int length);
```

where **destp** is a pointer to the parallel data in which the values are to be written, **sourcep** is a pointer to the scalar array, and **length** is the length, in **bools**, of the data pointed to by **destp**.

NOTE: Using these versions of **read_from_pvar** and **write_to_pvar** for aggregate data may make your program nonportable.

3.10 New allocated_detailed_shape Function

CM-5 C* contains a new version of the **allocate_detailed_shape** intrinsic function. As with the CM-200 version, this function lets you specify exactly how a shape is to be laid out on the processing nodes. If your program has known, stable patterns of communication, you may be able to use this function to speed up this communication and thereby improve performance.

Note that, since the CM-5 and CM-2 versions of **allocate_detailed_shape** are different, using this function makes your program nonportable.

Effective use of **allocate_detailed_shape** requires an understanding of how shapes are mapped onto CM-5 processing nodes by the run-time system. For the Beta release, we assume you have this understanding. The *C* Programming Guide* will provide information on this topic for the official release of CM-5 C*.

Include the header file **<cshape.h>** when you call **allocate_detailed_shape**.

The format of **allocate_detailed_shape** is as follows:

```
shape allocate_detailed_shape (
    shape *s,
    int rank,
    unsigned long extents[],
    unsigned long weights[],
    CMC_axis_order_t axis_orderings[],
    int physical_masks[],
    int subgrid_lengths[],
    int subgrid_sequences[]);
```

where:

s is a pointer to a shape. The remaining arguments specify this shape, and the function returns it. You must provide a value for this argument.

rank specifies the number of dimensions in the shape. You must provide a value for this argument.

extents specifies the number of positions along each dimension, starting with axis 0. You must provide values for this argument.

weights specifies the relative frequency of communication along each axis, starting with axis 0. For example, weights of 1 for axis 0 and 2 for axis 1 specify that communication occurs about half as often along axis 0. Only the relative values of the weights matter; for example, weights of 5 for axis 0 and 10 for axis 1 specify the same communication as weights of 1 and 2. Specifying the same values for different axes indicates that they have the same level of communication.

The **weights** values are used only if neither the **physical_masks** nor the **subgrid_lengths** argument is specified.

Pass **NULL** instead of the **weights** array to use the default weights, which are 1 for each axis.

axis_orderings
specifies the ordering of each axis, either **CMC_news_order** or **CMC_serial_order**. Pass **NULL** instead of this array to specify the default ordering, which is NEWS ordering for each axis.

physical_masks
specifies the mapping of positions to physical nodes. The physical mask for each axis must represent a contiguous set of bits, and must not use any bits used by another axis. The sum of the physical masks must use all bits 0 through *n*, where *n* is less than or equal to the total number of physical bits used to represent the positions of the physical shape.

If you are compiling for a CM-5 with vector units, the lowest two bits of the physical mask correspond to the four vector units on a node.

You can pass **NULL** instead of the **physical_masks** array. If you also pass **NULL** for **subgrid_lengths**, the weights are used to create the shape. If you specify values for **subgrid_lengths**, the physical masks will use the fewest bits necessary to accommodate the extent of each axis given the specified subgrid length; the less significant bits are used for lower-numbered axes.

subgrid_lengths
specifies the subgrid length for each axis.

If you pass **NULL** for **subgrid_lengths** and specify values for the **physical_masks** array, the subgrid lengths will be the minimum lengths required to represent the axis.

subgrid_sequences
specifies the sequence of subgrid axes. The default is for the highest-numbered (non-serial) axis to vary fastest (that is, the sequence is the reverse of the axis numbers). You can also specify that the lowest-numbered axis is to vary fastest (that is, the sequence is 0, 1, 2,...). In either case, serial axes must be last in the sequence. Other sequences are not allowed.

CM-5 C* provides several functions you can call to find out how the run-time system actually laid out a shape. You can use these functions to determine if you should use **allocate_detailed_shape** to specify a different layout.

Include the header file **<csshape.h>** when you call any of these functions.

The functions are defined as follows:

```
CMC_axis_order_t CMC_axis_ordering(shape s, int axis);
int CMC_physical_axis_mask(shape s, int axis);
int CMC_subgrid_axis_length(shape s, int axis);
int CMC_subgrid_axis_sequence(shape s, int axis);
int CMC_subgrid_size(shape s);
int CMC_subgrid_axis_increment(shape s, int axis);
int CMC_subgrid_axis_orthogonal_length(shape s, int axis);
int CMC_subgrid_axis_outer_increment(shape s, int axis);
int CMC_subgrid_axis_outer_count(shape s, int axis);
```

CMC_axis_ordering returns **CMC_news_order** if the specified axis is in NEWS order (the standard order), **CMC_serial_order** if it's in serial order.

CMC_physical_axis_mask returns an integer that represents the physical mask for the specified axis.

CMC_subgrid_axis_length returns the subgrid length of the specified axis.

CMC_subgrid_axis_sequence returns an integer that represents the specified axis's place in the sequence of axes within a subgrid.

CMC_subgrid_size returns the total number of positions in the subgrid for the specified shape.

CMC_subgrid_axis_increment returns an integer representing how many positions in memory separate consecutive subgrid positions along the specified axis. This is calculated by multiplying the subgrid lengths of all axes that have smaller subgrid axis increments (that is, the axes with lower subgrid sequences). If the positions along the subgrid axis are contiguous in memory, the function returns 1.

CMC_subgrid_orthogonal_length returns the subgrid-orthogonal-length for the specified axis. This is the total number of positions in the subgrid divided by the subgrid length of the axis.

CMC_subgrid_axis_outer_increment returns the product of the subgrid axis increment and the subgrid axis length for the specified axis.

CMC_subgrid_axis_outer_count returns the product of the subgrid lengths of all axes that have larger subgrid axis increments (that is, axes with higher subgrid sequences) than the axis you specify.

4 Developing a CM-5 C* Program

Develop your CM-5 C* program as described in Chapter 2 of the *C* User's Guide*. Note these points:

- The header files described in Section 2.2 exist in CM-5 C*.
- As mentioned above, you can't call Paris functions from a CM-5 C* program. Therefore, Section 2.3 of the *C* User's Guide* does not apply to CM-5 C*.
- Use the CM Fortran interface to CMSSL by calling the routines as described below.

I/O for this release of CM-5 C* is described in Section 8.

4.1 Calling CM Fortran

You can call CM Fortran subroutines as described in Section 2.5 of the *C* User's Guide*, but note these points:

- The description of the VAX interface is not applicable to CM-5 C*.
- See Section 5 on compiling, below, for information on compiling and linking a C* program that calls CM Fortran.
- As of Version 7.1, you can call CM Fortran programs compiled for the vector units.
- The name of the C* main routine must be `MAIN_()`.
- You can't pass the `CMC_complex` type to CM Fortran; use `CMC_double_complex` instead. This is a temporary restriction due to an overloading bug in the current release.

In the future, we hope to provide a more transparent interface to CM Fortran. To minimize recoding when this interface is available, we recommend that you call the subroutine as if you were calling it directly, then use a stub routine to provide the correct syntax to make it work now.

For example, you might call the CM Fortran routine `CMF_ROUTINE` like this in your program:

```
main()
{
    shape [10]s;
    float:s i, j;
    float x;
    int n;

    with (s)
        i = pcoord(0);
    CMF_ROUTINE(&j, &i, 1.0);
    for(n=0; n < positionsof(s); n++)
        printf("%f ", [n]j);
        printf("\n");
}
```

The stub routine would look like this:

```
#include <csfort.h>

CMF_ROUTINE(jp, ip, f)
int:void *jp, *ip;
float f;
{
    CMC_descriptor_t jp_desc, ip_desc;

    jp_desc = CMC_wrap_pvar(jp);
    ip_desc = CMC_wrap_pvar(ip);
    CMC_CALL_FORTRAN(cmf_routine_(jp_desc, ip_desc, &f));
    CMC_free_desc(jp_desc);
    CMC_free_desc(ip_desc);
}
```

We expect that eventually this stub routine will be generated automatically, given the appropriate declaration of **CMF_ROUTINE**; for now you have to write it by hand.

5 Compiling

CM-5 C* accepts the compiler options listed in Chapter 3 of the *C* User's Guide*, with the additions and changes listed in this section.

See also Section 2.1.1 for a discussion of the compiler options new in Version 7.1 Beta.

Use the **cs** command to compile your C* program. To specify that you want to use the CM-5 C* compiler, include the option **-cm5** on your command line; you can omit this if your site has made the CM-5 compiler the default target of the **cs** command. You can tell if the CM-5 is the default target by simply typing **cs** at your UNIX prompt. You will receive a help message that begins:

```
C* driver [CM5 SPARC 7.1 Beta]
```

if you include the **-sparc** option, or a CM-5 without vector units is the default target. Or:

```
C* driver [CM5 VECUNIT 7.1 Beta]
```

if you include the **-vu** option, or a CM-5 with vector units is the default target.

If the CM-200 compiler is the default, and you want to avoid having to specify the **-cm5** option, set the environment variable **CS_DEFAULT_MACHINE** to **cm5**. In that case, you would then have to specify the **-cm2** or **-cm200** option (depending on your target CM) to use the CM-200 C* compiler.

Note these other changes in compiler options:

- These CM-200 C* options are not accepted:
  - **-noline**
  - **-release**
  - **-ucode**
  - **-cpp**
  - **-keep c**, since the compiler does not generate C code
- The **-O** (optimization) option is currently disabled.
- Use the **-cmsim** option to compile and link a version of your program that can run on a Sun-4 SPARC workstation. The program can't do parallel I/O, graphics, or call other non-C* library routines.
- Using the **-g** option increases execution time considerably; use the **-cmdebug** option instead if this matters. The **-cmdebug** option provides faster execution, at the expense of some precision in debugging.
- The **-pg** option is not supported. The CM-5 system libraries don't have versions for use in profiling; in addition, Prism provides superior profiling capabilities.
- CM-5 C* has a new **-Wimplicit** option. Specify this option to receive a warning from the compiler if you call a function that has not been previously declared or defined.
- CM-5 C* has a new **-dirs** option. Use this option to find out where the compiler searches for binaries, libraries, include, and temporary files. It produces output like this:

```
% cs -cm5 -dirs
    C* driver [CM5 SPARC 7.1 Beta]
    bin_dir is /proj/cstar/release/7.1/beta/bin/
    lib_dir is /proj/cstar/release/7.1/beta/lib/
    include_dir is /proj/cstar/release/7.1/beta/include
    temp_dir is /usr/temp
```

The binary search path is the "**bin_dir**" directory specified in this message, followed by your **$PATH**, followed by **/bin**, **/usr/bin**, and **/usr/local/bin**.

The library search path is:

(1) Any directories you specified via the **-L** option

(2) The "**lib_dir**" directory specified in the **-dirs** message

(3) Directories you specify via **$LD_LIBRARY_PATH**

(4) **/lib**, **/usr/lib**, and **/usr/local/lib**

The include search path is:

(1) Any directories you specified via the **-I** option

(2) The "**include_dir**" directory specified in the **-dirs** message

(3) **/usr/include**

5.1 Compiling and Linking a C* Program that Calls CM Fortran

Follow these steps to compile and link a program that calls CM Fortran subroutines:

1. Compile the C* program, using the **-c** option. For example:

```
% cs -cm5 -c testcs.cs
```

2. Compile the CM Fortran program, also using the **-c** option. For example:

```
% cmf -cm5 -c testfcm.fcm
```

3. Issue **cmf** again to link, using one of these formats:

   - For CM-5s with vector units:

```
% cmf testcs.o testfcm.o -vu -Llib_dir -lcs_cm5_vu_sp
```

   - For CM-5s without vector units:

```
% cmf testcs.o testfcm.o -sparc -Llib_dir \
-lcs_cm5_sparc_sp
```

   - For Sun-4s:

```
% cmf testcs.o testfcm.o -cmsim -Llib_dir -lcs_cm5_cmsim
```

where *lib_dir* is the library directory listed by the **-dirs** option to the **cs** command.

If your programs are compiled with **-g** or **-cmdebug**, you must also specify **-lprism5** (when linking for the CM-5) or **-lprism_sim** (when linking to run on a Sun-4) before **-lcs_cm5_***xxx*. For example:

```
% cmf testcs.o testfcm.o -Llib_dir -lprism5 -lcs_cm5_vu_sp
```

6 Executing

Execute a CM-5 C* program as you would execute any program on the CM-5:

- Log in to the partition manager and, at the UNIX prompt, type the name of the executable program; or
- Submit the program as a batch request to NQS, as described in Section 4.3 of the *C* User's Guide*.

Ignore Section 4.2 of the *C* User's Guide*, since you don't have to explicitly attach to a CM-5.

To execute a program compiled with the **-cmsim** option, simply run it on a Sun-4.

7 Debugging

The debugging functions described in Chapter 5 of the *C* User's Guide* do not exist in CM-5 C*.

You can use the Prism programming environment with CM-5 C* as of Prism Version 1.2.

Note these points in using Prism Version 1.2 with CM-5 C*:

- When you visualize a pointer to a parallel object (for example, a parallel variable or parallel structure), you obtain three pieces of information:
  - The CM memory address of the object being pointed to
  - The address that represents the object's shape
  - A memory stride that indicates how many bytes are between the starting addresses of successive elements of the object on each physical processor

  Here are examples from a command-line Prism session:

```
(prism) whatis p
parallel int *p;
(prism) print p
pp'foo'p = [addr=0xa0000088; shape=0x3c018; stride=4]
(prism) whatis c
parallel struct foo  *c;
(prism) print c
pp'foo'c = [addr=0xa0000000; shape=0x3c018; stride=8]
```

- The Prism **assign** command cannot be used to change the values of parallel variables. Assignment to simple parallel variables will be available in the next Beta release of Prism Version 1.2.

8 I/O

CM-5 C* Version 7.1 provides synchronous parallel I/O support via an interface to the CMFS functions. The interface is the same for CM-5s with and without vector units.

Other interfaces for I/O may exist in the future.

The CMFS I/O library is not available if you compile with the **-cmsim** option to run on a Sun-4.

Note these points in using the current interface:

- Users should not include **<cm/paris.h>** or **<cm/cmtypes.h>**.

- Users must include **<cm/cmfs.h>** as they did on the CM-2.
- Link with **-lcmfs_cs -lcmfs_cm5**, in that order.
- These calls are specific to the CM-2 and are not supported in the CM-5 CMFS library:

```
CMFS_cm_to_standard_byte_order
CMFS_convert_vax_to_ieee_float
CMFS_convert_ieee_to_vax_float
CMFS_partial_read_file_always
CMFS_partial_write_file_always
CMFS_transpose_always
CMFS_transpose_record_always
CMFS_file_geometry
CMFS_twuffle_to_serial_order_always_1L
CMFS_twuffle_from_serial_order_always_1L
```

- There are C*-specific versions of **CMFS_read_file_always**, **CMFS_read_file**, **CMFS_write_file_always**, and **CMFS_write_file**. The declarations (from **<cm/cmfs.h>**) are:

```
overload CMFS_read_file, CMFS_read_file_always;
overload CMFS_write_file, CMFS_write_file_always;

int CMFS_read_file (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_read_file_always (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_write_file (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_write_file_always (int fd, void:void *dest,
                    int bytes_per_position);
```

  These interfaces provide basic compatibility with CM-2 C* code that calls CMFS.

  The functions are called with pointers to parallel variables. A pointer to a parallel variable of any type may be used. The specified length may be any number of bytes, but performance is significantly diminished when the length is not a multiple of four bytes. See below for a further discussion of I/O performance.

- The lengths passed to and returned by these functions are always byte lengths. For the C* interface, they indicate the number of bytes read or written in each position of the parallel variable. Note that on the CM-2 the

`CMFS_read_file` and `CMFS_write_file` functions take bit lengths, and that in either case `boolsizeof()` should be used to specify the length; this will make the program portable.

- There is currently no difference between the regular and the "always" versions of these functions. This is a temporary restriction. Users should only use the "always" versions until this restriction is lifted.
- Streaming and buffered I/O are not supported.
- You can use standard UNIX I/O routines to do serial I/O on CMFS files if the CMFS file system is NFS-mounted. See the *CM-5 I/O System Programming Guide* for more information.
- The total size of a file on a CMFS file system (that is, on the DataVault) will always be rounded up to be a multiple of 512 bytes.
- This interface does not work with files whose size is greater than or equal to 2 gigabytes.
- I/O performance may be significantly diminished if any of the following is true:
  - The size specified to the CMFS functions is not a multiple of 4 bytes.
  - The total amount of data being read or written is not a multiple of 16 bytes on the SDA, or 512 bytes on the DataVault.
  - The file position is not on a 4-byte boundary.
  - The parallel data passed to the CMFS function is an address that is not on a 4-byte boundary (for example, when the pointer points to a member of a parallel structure).
- CM-5 C*'s implementation of `CMFS_lseek` has changed. The routine now seeks into a file the number of bytes you specify multiplied by the number of positions in the current shape. (In the previous implementation, it would seek an absolute number of bytes into the file.) The routine `CMFS_serial_lseek` seeks an absolute number of bytes into a file.

9 Documentation Errors

This section discusses known problems with the *C* User's Guide* and the *C* Programming Guide* Version 6.0.2. Note that these are errors for the CM-200 version of C* as well.

9.1 Length Units for Communication Functions Should Be bools, Not Bits

All grid communication functions, as well as the **get** and **send** functions, are overloaded to operate on parallel data of any length.

The *C* Programming Guide* states that the length argument for these functions is in terms of bits; actually, it's in terms of **bool**s. (Note that this distinction doesn't matter for CM-200 C*, because in that implementation a parallel **bool** is 1 bit. In CM-5 C*, however, where a **bool** is 1 byte, it makes a difference.)

On page 158, the manual correctly gives an example for one of these functions in which **boolsizeof** is used to determine the length. CM-200 C* programs that use this technique to determine the length are portable to the CM-5; if the program uses bits, however, you will have to recode it before it can be ported.

9.2 Arguments Reversed in memcpy, boolcpy Example

Page 6 of the *C* User's Guide* contains an example of the use of the functions **memcpy** and **boolcpy**. The **source** and **dest** arguments are reversed in these examples.

9.3 The rank Function's Behavior with Scan Sets Is Incorrectly Documented

The documentation of the `rank` function on pages 199–202 of the *C* Programming Guide* incorrectly explains the behavior of `rank` when used with scan sets.

The `rank` function operates somewhat differently from other functions when you specify an sbit. The sbit restarts the ranking of values with the scan set, as documented; however, *it does not restart the values assigned to the ranks*. For example, if a scan set extends from position [4] through position [15], the ranks assigned within this scan set are 4 through 15, rather than 0 through 11.

10 Porting CM-200 C* Programs to the CM-5

Most CM-200 C* programs should port without difficulty to the CM-5. You must recompile and relink using the CM-5 C* compiler This list summarizes the changes that you must make (when applicable) to ensure portability:

- Remove all Paris calls.
- Remove all calls to libraries not supported on the CM-5.
- Remove all include files not supported on the CM-5 (for example, `<cm/paris.h>`).
- If you express lengths in terms of bits in a function (for example, in the overloaded versions of the grid communication functions or the `get` or `send` function), rewrite the code to express the size with `boolsizeof` and the appropriate parallel type.
- Change calls to `allocate_detailed_shape` to use the new format.
- The CM-5 C* compiler disallows casts between scalar types and pointers to parallel variables. If you call `palloc()` in a CM-200 C* program without including `<stdlib.h>` (which properly declares its return type) and cast the result, the code won't compile on the CM-5. Thus, this code won't work:

```
/* No included stdlib.h file */

int:current *p = (int:current *)palloc(current,
                 boolsizeof(int:current));
```

Change it to this so that it will work in CM-5 C*:

```
#include <stdlib.h>

int:current *p = palloc(current,
                        boolsizeof(int:current));
```