Chapter 2

Using the C* CMSSL Interface

This chapter contains information about running C* programs that call CMSSL routines. The following topics are included:

- creating a C* CMSSL program
- restrictions and performance guidelines
- complex data types and macros
- the CMSSL safety mechanism
- on-line sample code and man pages

2.1 Creating a C* CMSSL Program

To use the CMSSL from within a C* program, follow these steps:

1. Read the *CMSSL Release Notes* for information specific to the CMSSL release you are using and for updates to the manual. Important information such as switches for linking with CMSSL may change from release to release.

2. Place calls to CMSSL routines into C* code. Your code must satisfy the restrictions described in Section 2.2.

3. Include the header file **`<cm/cmssl.h>`** in C* code that calls CMSSL routines.

4. Use the **`cs`** command to compile your code.

Copyright © 1993 Thinking Machines Corporation

5. Include the following switches on the **cs** command line when linking your code:

   - **-g**
   - **-lcmsslcm5** (for machines without vector units), **-lcmsslcm5vu** (for machines with vector units), or **-lcmsslcm5vu-node** (to use the nodal library on machines with vector units) to link with CMSSL.
   - **-lcmffio_cmfs**, **-lcmfcompiler**, **-lcmfs_cm5**, and **-lcmf77**, to link with the CM Fortran support libraries required by CMSSL, if you are using global CMSSL.
   - **-lcmffio**, **-lcmfcompiler**, and **-lcmf77**, to link with the CM Fortran support libraries required by CMSSL, if you are using nodal CMSSL.

   The switches listed in the last two bulleted items above are required and must follow the **-lcmsslcm5**, **-lcmsslcm5vu**, or **-lcmsslcm5vu-node** switch on the command line.

   These guidelines may change with a future version of C*; watch for a technical bulletin that updates them. Refer to the *C* User's Guide* for descriptions of **cs** command line switches.

The remainder of this chapter contains details about the steps listed above.

2.1.1 Including the CMSSL Header File

You must place the following line at the top of any C*program unit that makes a CMSSL call:

```
#include <cm/cmssl.h>
```

This line allows the C* program to access the header file **<cm/cmssl.h>**, which contains prototypes for the routines and defines the symbols and types required by the interface.

If the C* compiler cannot find the CMSSL include file, check your partition manager for the existence of a path to the appropriate directory. If the file appears to be missing, consult your system administrator or your Thinking Machines Corporation customer service representative.

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

2.1.2 Calling CMSSL Routines

To invoke a CMSSL routine from within a C* program, first make sure you are using compatible versions of C* and CMSSL. The *CMSSL Release Notes* shipped with the version you are using includes a section describing which version of the C* compiler is required.

2.1.3 Compiling and Linking

If the C* program *program* calls CMSSL routines, compile it and link it with CMSSL using one of the following command lines at the UNIX prompt:

For machines without vector units:

```
%cs -g -cm5 -sparc -o program program.cs -lcmsslcm5
-lcmffio_cmfs -lcmfcompiler -lcmfs_cm5 -lcmf77
```

For machines with vector units:

```
%cs -g -cm5 -vu -o program program.cs -lcmsslcm5vu
-lcmffio_cmfs -lcmfcompiler -lcmfs_cm5 -lcmf77
```

For the nodal CMSSL library on machines with vector units:

```
%cs -g -cm5 -vu -node -o program program.cs
-lcmsslcm5vu-node -lcmffio -lcmfcompiler -lcmf77
```

The **-g** option is required only when you are linking. Note that you must have CM Fortran installed to run CMSSL for C*.

If you are using nodal CMSSL, you must use CMMD Version 3.0 or higher. If CMMD Version 3.0 or higher is not the default on your system, add **-cmmd_root /usr/cmmd/3.0** before **-lcmsslcm5vu-node** on the **cs** command line.

The guidelines above may change with a future version of C*; watch for a technical bulletin that updates them. Refer to the *C* User's Guide* for descriptions of **cs** command line switches.

Copyright © 1993 Thinking Machines Corporation

Using the Correct Version of CMOST

CMSSL is a layered product. Any CMSSL version requires specific versions of CMOST, C*, and CM Fortran. If these dependencies are not observed, proper operation of the CMSSL routines is unlikely. Consult the most current version of the *CMSSL Release Notes* to find out which versions of CM Fortran and CMOST are required by the current CMSSL.

2.1.4 Executing CMSSL Programs

Execute a C* CMSSL program just as you would any compiled C* program.

2.2 Restrictions and Performance Guidelines

To use the C* CMSSL routines effectively, you must adhere to certain restrictions and guidelines that stem from the differences in programming model between C* and CM Fortran.

C* supports certain programming styles, data objects, and implementation details that CM Fortran does not support, and vice versa. These differences are resolved in two ways:

- CMSSL imposes restrictions on supported data types and argument passing mechanisms. (Some of these restrictions may be removed in future releases of the CMSSL C* interface.)
- CMSSL supports certain C*-specific features, but at a performance cost. In particular, you can pass a parallel structure slot or array element to CMSSL, but at a performance cost, because CMSSL copies the slot or array element to a temporary parallel variable. To optimize performance, you must understand which situations incur a performance cost, and what that cost is.

Each subsection below describes a key issue and presents the related restrictions and/or performance guidelines. Sections 2.2.6 and 2.2.7 provide summaries of the restrictions and guidelines, respectively.

Copyright © 1993 Thinking Machines Corporation

2.2.1 CMSSL Expects Only Certain Numeric Data Types

The C* CMSSL interface accepts six data types for parallel variables: **int**, **float**, **double**, **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t**. The **CMSSL_complex_t** and **CMSSL_dcomplex_t** types are discussed in Section 2.3. Because parallel variables are passed by reference (see Section 2.2.2), data type promotion is not performed on them.

In the current release, CMSSL does not support masks or scalar logical values whose elements are C-style logical quantities (nonzero = true), and does not support data of type **bool**. It does support masks and scalar logical values whose elements are of type **CMSSL_logical_t**, defined in the header file **<cm/cmssl.h>** as follows:

```
typedef enum {
  CMSSL_true = -1,
  CMSSL_false = 0
} CMSSL_logical_t;
```

2.2.2 CMSSL Expects Parallel Arguments to Be Passed by Reference

C* requires that parallel variables passed by value to an external routine be of shape **current**. Because many CMSSL routines require several parallel arguments with different shapes in the same call, all parallel arguments passed to CMSSL are passed by reference.

Thus, CMSSL requires that you pass it a scalar pointer to a parallel variable. You can pass a pointer to a simple parallel variable, a member of a parallel structure, or an element of a parallel array. However, C* does not allow you to create a pointer to a left-indexed variable or to a parallel array indexed with a parallel index; therefore, you cannot pass such objects to CMSSL. For example, given

```
shape [16][64] sh1;
int:sh1 x[16];
```

you can pass CMSSL **x**[5], but not **x**, [.][37]**x**, [3][4]**x**, or [3][4]**x**[3]. As another example, given

```
shape [64] sh2;
double:sh2 y, x[128];
```


Copyright © 1993 Thinking Machines Corporation

```
int:sh2 x_offset;
x[x_offset] = y;
```

the expression **x[x_offset]** is a legal, but because **x_offset** is parallel, **&x[x_offset]** is not legal C*; therefore, you cannot pass **x[x_offset]** to CMSSL.

Except for cases in which an associated mask parameter is provided in the call, a CMSSL routine references and/or modifies any parallel variables passed to it without regard to any context established in the calling routine, for example in a **where** construct. In effect, the CMSSL routine operates within an **everywhere** block.

Scalar input arguments are passed by value; scalar output arguments are passed by reference.

2.2.3 CMSSL Expects One Data Element per Position

As described in Chapter 1, CMSSL is optimized to work on multiple instances of a problem at the same time. In the current release, the data axes and instance axes of a problem must all be parallel (that is, they must be axes of the data's shape). While a parallel variable may be an aggregate — for example, an array or structure — you may pass only a reference to a *single* array element or structure slot to CMSSL. Only one atomic (non-aggregate) data item per position is operated on by CMSSL routines. (The only exception to this rule is that CMSSL supports complex data types, which are implemented as aggregate data types in C*; see Section 2.3.) For example, given

```
shape [16][32]shape1;
float:shape1 A[8];
```

you may pass CMSSL the component **A**[5], but you may not pass it the entire array **A**; CMSSL will operate on only one element. Moreover, passing CMSSL a component of an aggregate parallel variable may have a performance cost, as discussed in Sections 2.2.4 and 2.2.5, below.

Local and Non-Local Axes

An axis of a shape can be local (reside within a single processing element) or non-local (span multiple processing elements), depending on the shape's

Copyright © 1993 Thinking Machines Corporation

layout. (For more information about layout, see the discussion of the **allocate_detailed_shape** function in the *C* Programming Guide*). A *processing element* is a vector unit on machines with vector units, or a SPARC node on machines without vector units.

2.2.4 CMSSL Prefers Doubleword-Aligned Arguments

CMSSL performs best when the data element in each position of a parallel argument is *doubleword-aligned* within the processing element memory — that is, begins at a memory address that is a multiple of 8. In a C* parallel variable, the first data element in each processing element begins on a doubleword boundary. However, if the data element is an aggregate (for example, an array or structure), components after the first one may not be doubleword-aligned. If you pass one of these non-doubleword-aligned components to CMSSL, CMSSL copies the data into a temporary variable that is doubleword-aligned, with a performance and memory overhead cost. (See Section 2.2.8 for specific information about the cost.)

The rules listed below govern memory alignment of the components of an aggregate data element in C*. (These are the same rules that apply to ordinary variables in C.)

- **int** and **float** components have singleword alignment.
- **double**, **CMSSL_complex_t**, and **CMSSL_dcomplex_t** components have doubleword-alignment.
- The memory alignment of an aggregate, *agg*, equals the largest alignment of any component of *agg*. This rule applies recursively to aggregates of aggregates.

Examples

For example, if we have

```
shape [16][32]shape1;
float:shape1 A[8];
```

then while the first data element, **A**[0], is doubleword-aligned in each processing element, the second element, **A**[1], begins just four bytes after **A**[0], and

Copyright © 1993 Thinking Machines Corporation

therefore is not doubleword-aligned. Thus, supplying **A**[1] to CMSSL incurs a performance cost.

Similarly, if we have

```
shape [16][32]shape1;
struct{
    float aa;
    int bb;
}:shape1 zz[10];
```

then while **zz[0].aa** is doubleword-aligned, **zz[0].bb** begins just four bytes after **zz[0].aa** does, and therefore is not doubleword-aligned. Thus, supplying **zz[0].bb** to CMSSL incurs a performance cost.

As another example, if we have

```
shape [16][32]shape1;
struct{
    float aa;
    double cc;
}:shape1 zz[10];
```

the whole structure has doubleword alignment, because the largest memory alignment within it is the doubleword alignment of **cc**.

2.2.5 CMSSL Prefers Contiguous Data Elements

In the current release, CMSSL performs best when successive data elements in each position of a parallel argument are allocated contiguously within the processing element memory. In C*, a component of an aggregate data element is not allocated contiguously. If you pass such a component to CMSSL, CMSSL copies the data into a temporary variable that is contiguous. This copy incurs a performance and memory overhead cost. Thus, to avoid this overhead, avoid passing components of aggregate parallel arguments to CMSSL. (See Section 2.2.8 for specific information about the cost.)

Examples

As an example, suppose we have

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

```
shape [16][32]sh;
int x:sh [64];
```

The component **x**[0] of **x** is not contiguous, since **x**[0] in one position and **x**[0] in the next position are separated by **x**[1] through **x**[63].

As another example, consider the declaration from the previous section:

```
shape [16][32]shape1;
struct{
     float aa;
     double cc;
}:shape1 zz[10];
```

Suppose that within the memory of a given processing element, **zz** starts at memory location 208. The first element, **zz[0].aa**, occupies locations 208−211. Since **zz[0].cc** is of type **double** and therefore must be double-word-aligned, it begins at the next multiple-of-8 location, 216, and occupies locations 216−223. Locations 212−215 are unfilled. This pattern repeats for all elements of **zz**. Because the elements of **zz** are not contiguous, if you supply any element of **zz** to CMSSL, CMSSL makes a copy of the data, at a performance cost.

2.2.6 Summary of Restrictions

In the current release, CMSSL imposes the following restrictions on C* programs:

- The C* CMSSL interface accepts six data types for parallel variables: **int**, **float**, **double**, **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t**. In the current release, CMSSL does not support masks or scalar logical values whose elements are C-style logical quantities (nonzero = true), and does not support data of type **bool**. It does support masks and scalar logical values whose elements are of type **CMSSL_logical_t**.
- All parallel variables passed to CMSSL are passed by reference. CMSSL requires a scalar pointer to a parallel variable.
- Scalar input parameters are passed by value; scalar output parameters are passed by reference.

Copyright © 1993 Thinking Machines Corporation

- While a parallel variable may be an *aggregate* — for example, an array or structure — you may pass only a reference to a single array element or structure slot (a component of the aggregate) to CMSSL. Only one atomic (non-aggregate) data item per position is operated on by CMSSL routines. The only exception to this rule is that CMSSL supports complex data types, which are implemented as aggregate data types in C*.

2.2.7 Summary of Performance Guidelines

Follow these guidelines to avoid extra performance overhead:

- Be sure that the parallel arguments you pass to CMSSL are doubleword-aligned. If you pass a non-doubleword-aligned parallel argument, CMSSL copies the data into a new variable that is doubleword-aligned, with a performance cost.

- Be sure that the parallel arguments you pass to CMSSL contain contiguous data elements. If you pass a parallel argument containing non-contiguous elements, CMSSL copies the data into a new variable that is contiguous, at a performance cost.

The simplest way to satisfy these guidelines is to pass only atomic (non-aggregate) parallel arguments to CMSSL. If you pass parallel structure members or parallel array elements to CMSSL, you will incur the cost of the copy.

2.2.8 More on Performance Costs

As discussed above, if parallel data passed as an argument to a CMSSL routine is not contiguous in memory or is not doubleword-aligned, CMSSL guarantees contiguity and alignment of the data by allocating a contiguous, doubleword-aligned temporary parallel variable of the same shape and data type. If the argument's shape has p positions and the size of its data type is s bytes, the temporary variable will consume at least ps bytes of parallel heap memory for the duration of the call (somewhat more if one or more axes of its shape end in garbage positions).

Copyright © 1993 Thinking Machines Corporation

Depending on how the CMSSL routine uses such a parameter, there may be time overheads as well, because local copying to and from the temporary variable consumes local memory bandwidth. Input/output parameters incur the heaviest copy cost because they are copied both in and out. Input-only parameters are copied in only; output-only parameters are copied out only.

Certain CMSSL routines, such as **CMSSL_part_scatter_setup** and **CMSSL_fft_setup**, accept one or more parallel variables as "templates" without looking at their data values at all. Templates are used only to extract information about shape, data type, and stride; the actual data values of such arguments are irrelevant to such routines. In these cases, no copy in or copy out is done (but the contiguous aligned temporary parallel variable must be allcoated anyway).

2.3 Complex Data Types and Macros

The complex data types **CMSSL_complex_t** and **CMSSL_dcomplex_t** are defined in the file **<cm/cmssl_complex.h>**, which is included by **<cm/cmssl.h>**. These types are defined so that, although each slot has single-word memory alignment, the whole **CMSSL_complex_t** or **CMSSL_dcomplex_t** is always doubleword-aligned. (Note that this is not usually true for a structure containing two **float** slots.) Thus, you can pass complex numbers to CMSSL without incurring the performance cost of a copy. To accomplish this, **CMSSL_complex_t** and **CMSSL_dcomplex_t** are defined using a union with a **double** data element. The definitions from **<cm/cmssl_complex.h>** are reproduced here:

```
#ifndef __CMSSL_CS_COMPLEX__
#define __CMSSL_CS_COMPLEX__

/* Force doubleword alignment for single complex */
typedef union { struct { float re_; float im_; } values;
                double alignment ; } CMSSL_complex_t;

/* Make double complex consistent with single complex */
typedef union { struct { double re_; double im_; } values;
                double alignment ; } CMSSL_dcomplex_t;
/*
 * initialize both parts with z = { <expr>, <expr> }
 * assign real part with re(z) = <expr>
 * assign imag part with im(z) = <expr>
```


Copyright © 1993 Thinking Machines Corporation

```
 */
#define re(z) ((z).values.re_)
#define im(z) ((z).values.im_)

#endif __CMSSL_CS_COMPLEX__
```

Since C* does not currently support complex arithmetic functions, you must perform all complex arithmetic operations in terms of the real and imaginary parts of complex numbers. To facilitate these operations, CMSSL provides the two macros, **re( )** and **im( )**, defined in the include file above. These macros return the real and imaginary parts of a data element of type **CMSSL_complex_t** or **CMSSL_dcomplex_t**.

The definitions of **CMSSL_complex_t** and **CMSSL_dcomplex_t** may change in future releases. For future compatibility, we recommend that you use the **re( )** and **im( )** macros.

2.4 The CMSSL Safety Mechanism

You can use the CMSSL safety mechanism in two ways:

- by setting the environment variable **CMSSL_SAFETY**
- by using the calls **CMSSL_get_safety** and **CMSSL_set_safety** in a program

2.4.1 Safety Mechanism Features

The CMSSL safety mechanism offers two basic features: it synchronizes the CM-5 parallel processing elements and partition manager so that you can pinpoint the area of code that generated an error, and it performs error checking and reports errors at several levels of detail.

Synchronization

The CM-5 parallel processing elements and partition manager operate asynchronously with respect to one another. Without the CMSSL safety mechanism, an error that occurs in the parallel processing elements is not reported to the partition manager until the next time the partition manager requests information from or checks the status of the elements. Such a re-

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

quest or status check is known as an *implicit synchronization* because it has the side effect of synchronizing the processing elements and partition manager, allowing the processing elements to report any accumulated errors. When an implicit synchronization occurs, there is no way to tell exactly when the reported error occurred, or which module of code produced it.

The CMSSL safety mechanism addresses this problem by forcing *explicit synchronization* between the parallel processing elements and the partition manager before, after, and within each CMSSL call in your code. The safety mechanism traps and reports errors, indicating when the errors occurred in relation to the synchronization points.

Error Checking and Reporting

The safety mechanism can perform error checking and generate run-time error information at several levels of detail. You can turn safety checking on at any level during all or part of a program. One level checks for errors in the usage and arguments of the CMSSL calls in your program; a more de+tailed level also checks for errors generated by internal CMSSL routines. Examples of errors found and reported by the safety mechanism include the following:

- A supplied or returned data element that should be numerical is not; for example, it is identified as a "Not a Number" (NaN) or as infinity. (NaNs are defined in the IEEE Standard for Binary Floating-Point Arithmetic.)
- The code generates a division by 0 (for example, because of bad data, a user error, or an internal software problem).
- The code references a memory location that it has not initialized. The safety mechanism identifies this kind of error by writing NaNs to all allocated processing element memory. If the code references a memory location without first explicitly assigning it a numerical value, the NaN at that location causes further errors that make the original erroneous reference easy to find.

As more debugging checks and safety levels are added in future releases, CMSSL safety checking will become more exhaustive.

Copyright © 1993 Thinking Machines Corporation

2.4.2 Levels of Error Checking

The CMSSL safety mechanism currently provides the following levels:

0 (**off**) Turns off the safety mechanism. Explicit synchronization and error checking are not performed. This level is appropriate for production runs of code that has already been thoroughly tested.

1 (**on**) Checks for and reports errors caused by incorrect usage or arguments in high-level-language CMSSL calls. Performs explicit synchronization before and after each call and locates each error with respect to the synchronization points. This safety level is appropriate during program development or during runs for which a small performance penalty can be tolerated.

9 (**full**) Checks for and reports all level 1 errors, and in addition any errors generated by the lower levels of code that are called by the high-level-language CMSSL calls. Performs explicit synchronization in these lower levels of code and locates each error with respect to the synchronization points. This level performs all implemented error checking and exacts a very high performance price. It is appropriate for detailed debugging when a problem occurs. If you cannot analyze and correct the problem, provide your local site coordinator, applications engineer, or Thinking Machines Corporation customer service representative with the output generated by level 9 safety checking.

At levels 1 and 9, some safety mechanism error messages are displayed at the terminal when you run the program; other information appears in the backtrace when you use a debugger such as Prism.

If you report a software problem to your local site coordinator, applications engineer, or Thinking Machines Corporation customer service representative, you may be asked to run your program with the CMSSL safety mechanism enabled at a level other than 0, 1, or 9. These additional levels are used for pinpointing problems in the internal software or for obtaining internal status information.

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

2.4.3 Setting the CMSSL Safety Environment Variable

To set the CMSSL safety level using the **CMSSL_SAFETY** environment variable, issue the command

```
setenv CMSSL_SAFETY { 0 | 1 | 9 | off | on | full }
```

choosing one of the listed options. As indicated above, 0 is equivalent to **off**, 1 to **on**, and 9 to **full**.

The advantage of using the **CMSSL_SAFETY** environment variable is that you can set or change the safety level without recompiling your code.

2.4.4 Using CMSSL Safety from within a Program

To set the CMSSL safety level, issue the following call and specify the desired level in the integer argument *n*:

CMSSL_set_safety (*n*);

To obtain the current CMSSL safety level, issue the following call:

CMSSL_get_safety ( );

The advantage of using these calls from within a program is that you can set or obtain the safety level at any point within your code. However, you must recompile the code each time you change these calls.

NOTE

The inner product, 2-norm, outer product, matrix vector multiplication, vector matrix multiplication, and matrix multiplication routines described in Chapter 3 perform error checking only when the CMSSL safety mechanism is on. Therefore, we strongly recommend that you turn CMSSL safety on when testing new programs that call these routines.

Copyright © 1993 Thinking Machines Corporation

2.5 On-Line Sample Code and Man Pages

Included with CMSSL are sample on-line programs that demonstrate how to call each CMSSL routine. You are encouraged to experiment with these sample programs. Also included with the CMSSL are on-line man pages for all routines.

The on-line sample programs are located in subdirectories of the CMSSL examples directory. The default location for the examples directory is

`/usr/examples/cmssl`.

Examples for the operation *operation* are included in the subdirectory

operation`/cstar`

or

operation`/`*sub-operation*`/cstar`

of the examples directory. For example, the sample code for the sparse gather utility is located in the subdirectory

`sparse-utilities/cstar`

of the examples directory. If you do not find the on-line examples in **`/usr/examples/cmssl`**, check with your system administrator (or the person who installs CMSSL at your site) to find out where they were installed.

To read the on-line man page for a routine, enter the command

`man` *routine_name*

at the UNIX prompt.

Copyright © 1993 Thinking Machines Corporation