The
Connection Machine
System

CM-5 C* User's Guide

Version 7.1
May 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, May 1993

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

Contents

About This Manual

Objectives of This Manual

This manual describes how to develop, compile, execute, and debug C* programs on a CM-5 Connection Machine system.

Intended Audience

Readers are assumed to have a working knowledge of the C* language and of the UNIX operating system.

Organization of This Manual

Chapter 1 **Introduction**
Chapter 1 is a brief overview.

Chapter 2 **Developing a C* Program**
This chapter describes C* libraries and associated header files, and explains how to call CM library functions and CM Fortran subroutines from a C* program.

Chapter 3 **Compiling a C* Program**
Chapter 3 describes the C* compiler and its command line options.

Chapter 4 **Executing a C* Program**
Chapter 4 describes how to run a C* program.

Chapter 5 **Debugging a C* Program**
This chapter gives an overview of how to debug a C* program in Prism, the Connection Machine's programming environment.

Appendix **Man Pages**
The appendix contains man pages for the **cs** compiler command and C* header files.

Associated Documents

See the *C* Programming Guide* for a description of the C* language.

The manual *Getting Started in C** provides an overview of C* for beginning users.

The manual *CM-5 C* Performance Guide* provides information on how to increase the performance of your C* program on the CM-5.

Notation Conventions

The table below displays the notation conventions used in this manual:

Convention	Meaning
bold typewriter	C* and C language elements, such as keywords, operators, and function names, when they appear embedded in text. Also UNIX and CMOST commands, command options, and file names.
italics	Parameter names and placeholders in function and command formats.
typewriter	Code examples and code fragments.
% **boldface**	In interactive examples, user input is shown in **boldface** and system output is shown in regular typewriter font.

Customer Support

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

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

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

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

Chapter 1

Introduction

C* is an extension of the C programming language designed for the Connection Machine data parallel computing system. This chapter presents an overview of the process of developing, executing, and debugging a C* program on a CM-5 system. The rest of this manual goes into the process in more detail.

1.1 Developing a C* Program

Develop C* source code in one or more files. Use the suffix `.cs` if the file contains parallel code or any other features that C* adds to Standard C (for example, the new `<?=` and `>?=` operators). Chapter 2 describes facilities for developing a C* program. It also describes how to call functions in CM libraries, as well as CM Fortran subroutines.

1.2 Compiling a C* Program

Compile the files by issuing the command `cs`. The command can take various options, some of which are identical to options for the C compiler `cc`. Chapter 3 describes the compiler options and the compiling process in detail.

1.3 Executing a C* Program

You execute a C* program on a CM-5 partition manager, just as you would any UNIX program or command. You can also submit the program as a batch request to NQS, the CM's batch system.

Executing a C* program is discussed in more detail in Chapter 4.

1.4 Debugging a C* Program

You can debug and analyze the performance of your C* program using Prism, the CM programming environment. Prism is described briefly in Chapter 5.

Chapter 2

Developing a C* Program

A C* program can consist of:

- standard serial C code
- C* code; see Section 2.1
- header files; see Section 2.2
- calls to the CM timing utility; see Section 2.3
- calls to CM library functions; see Section 2.4
- calls to CM Fortran subroutines; see Section 2.5

In addition, C* programs can be called from CM Fortran programs. See Section 2.6.

2.1 C* .cs Files

All C* code must appear in files with the suffix `.cs`. C* code consists of any of the extensions that C* adds to Standard C; see the *C* Programming Guide* for a discussion of these extensions. Standard C code can appear in either `.c` or `.cs` files; putting it in `.c` files speeds up compilation, as discussed in Section 3.2.

2.1.1 C* Keywords

C* adds these new keywords to Standard C:

```
allocate_detailed_shape
allocate_shape
bool
boolsizeof
current
dimof
everywhere
overload
pcoord
physical
positionsof
rankof
shape
shapeof
where
with
```

C* code must not use these words, except as prescribed in the definition of the language.

2.1.2 Reserved Identifiers

Identifiers beginning with `CM` are reserved for use by the Connection Machine system software. Do not create identifiers beginning with `CM` in your programs.

2.2 Header Files

C* substitutes its own header files for some Standard C header files. To find out the location of these and other C* header files at your site, issue this command:

```
% cs -cm5 -dirs
```

The files are in the include directory listed in response to this command; see Section 3.4.4 for more information. Appendix A contains the man pages for some of these files.

C* accepts other Standard C libraries and associated header files, as long as they are ANSI-compliant. Exceptions include:

- header files that use C* keywords; see Section 2.2.4

- header files containing syntax that is not accepted by the C* compiler (for example, `<a.out.h>`)
- header files that depend on internal compiler support not provided by C* (for example, `<alloc.h>`)

In addition to its versions of some Standard C header files, C* includes these header files:

- `<cscomm.h>`, which is the header file for the communication functions described in Part III of the *C* Programming Guide.*
- `<csshape.h>`, which is the header file for a group of routines used in obtaining information about shape layout, as described in the *C* Programming Guide.*
- `<cstable.h>`, which provides routines used in C*'s table lookup utility, also described in the *C* Programming Guide.*
- `<csfort.h>`, which you use when calling CM Fortran routines from a C* program. See Section 2.5.

2.2.1 The <math.h> File

The C* version of `<math.h>` declares all the functions in the UNIX serial math library and extends all ANSI serial functions with parallel overloadings. No special library is required to use these functions.

2.2.2 The <stdlib.h> File

The file `<stdlib.h>` contains scalar and parallel declarations of the function **abs**, **rand**, and **srand**; the parallel versions of **rand** and **srand** are named **prand** and **psrand**. The file also contains the declarations of **palloc**, **pfree**, and **deallocate_shape**, which are described in the *C* Programming Guide*. No special library is required to use these functions.

Note that **prand** returns a parallel variable in which each element is sampled independently from the uniform distribution.

2.2.3 The <string.h> File

The file **<string.h>** contains parallel declarations of the functions **memcpy**, **memmove**, **memcmp**, and **memset**. In addition, it contains declarations of boolean versions of these functions, called **boolcpy**, **boolmove**, **boolcmp**, and **boolset**. Use the boolean versions to maintain source compatibility with CM-200 C*.

The boolean versions take pointers to **bool**s for arguments and return pointers to **bool**s (except for **boolcmp**). If you are dealing with arguments that are not **bool**s, you must cast them to be pointers to **bool**s. Also, note that the size argument for **memcpy** and related functions is in terms of **char**s, while the size argument for the boolean versions is in terms of **bool**s.

For example, in this code fragment, both **memcpy** and **boolcpy** copy **source** to **dest**:

```
#include <string.h>
/* ... */
int:current dest[2], source[2];

memcpy(dest, source, 2*sizeof(int:current) );
boolcpy( (bool:current *)dest,
    (bool:current *)source, 2*boolsizeof(int:current) );
```

2.2.4 Header Files and C* Keywords

A difficulty can occur when you want to include a standard header file that also makes use of a C* keyword. For example, the X Window System header file **<X11/Xlib.h>** uses the C* keyword **current** as the name of a structure member. Including this file would result in a syntax error. In such a situation, you can do this:

```
#define current Current
#include <X11/Xlib.h>
#undef current
```

This redefines **current** to be **Current** while **<X11/Xlib.h>** is being included, then undefines it. Of course, if you subsequently want to use the **<X11/Xlib.h>** structure member in your program, you must refer to it as **Current**.

2.3 Timing a Program

CMOST provides a timing utility that you can use in a C* program to determine how much time any part of a program takes to execute on the nodes.

The timing utility has these features:

- A timer calculates total time the processing nodes were active, with microsecond precision.
- Multiple timers can be active at the same time.
- Timers can be nested. This allows you, for example, to start one timer that will time the entire program, while using other timers to determine how different parts of the program contribute to the overall time.

You can have up to 64 timers running in a program. An individual timer is referenced by an unsigned integer (from 0 to 63 inclusive) that is used as an argument to the timing instructions. Instructions with the same number as an argument affect only the timer with that number.

To start timer 0, for example, put a call to the **CM_timer_start** routine in your program, using 0 as an argument:

```
CM_timer_start(0);
```

You can subsequently stop timer 0 by calling the **CM_timer_stop** routine later in your program:

```
CM_timer_stop(0);
```

This function stops the timer and updates the values for total elapsed time and total node idle time being held by the timer. You can subsequently call **CM_timer_start** again to restart timer 0; the timing starts at the values currently held in the timer. This is useful for measuring how much time is spent in a frequently called subroutine. The timer keeps track of the number of times it has been restarted.

You can start or stop other timers while timer 0 is running; each timer runs independently.

To get the results from timer 0, call this routine after you have called **CM_timer_stop**:

```
CM_timer_print(0);
```

CM_timer_print prints information like this to your standard output:

```
Starts: 1
CM Elapsed time: 27.7166 seconds
CM busy time: 23.1833 seconds
```

These routines return specific information from the timer for use in a program:

- **CM_timer_read_starts** returns an integer that represents the number of times the specified timer has been started.
- **CM_timer_read_elapsed** returns a double-precision value that represents the total elapsed time (in seconds) for the specified timer. "Elapsed time" refers to process time, not wall-clock time.
- **CM_timer_read_cm_idle** returns a double-precision value that represents the total CM idle time (in seconds) for the specified timer.
- **CM_timer_read_cm_busy** returns a double-precision value that represents the total time (in seconds) the CM was busy for the specified timer. CM busy time is the total elapsed time minus the CM idle time.
- **CM_timer_read_run_state** returns 1 if and only if the specified timer is running. Otherwise, the routine returns 0.

If you use any of these **CM_timer_read_*xxx*** routines, include the file **<cm/timers.h>**.

In addition, **CM_timer_set_starts** takes a timer number and an integer as arguments. It sets the number of starts for the specified timer to the specified value. This is useful if you want to write a function that can query a running timer without changing the number of starts. Not changing the number of starts is important if you want to know how many times a large chunk of code was called, but you also want to get sub-timings within that block.

To clear the values maintained by a timer, call **CM_timer_clear**. For example, to clear the value maintained by timer 0, put this call in your program:

```
CM_timer_clear(0);
```

This zeroes the total elapsed time, the total node idle time, and the number of starts for this timer.

NOTE: For compatibility with CM-200 C*, CM-5 C* also provides versions of the timing routines that begin with **CMC** instead of **CM**. If your program contains the **CMC** versions of the routines, make sure you include the file **<cstimer.h>**.

2.3.1 Hints on Using the Timing Utility

The elapsed time reported by a timer includes time when the process is swapped out on the partition manager. The more processes that are running, the more distorted this figure will be. Therefore, we recommend that you use a partition manager that is as unloaded as possible.

If you can't guarantee that you will have exclusive use of the CM-5, try to run the process several times; the minimum elapsed time reported will be the most accurate.

In addition, we recommend that you avoid stopping a process that is being timed.

Note that the inclusion of calls to the timer functions can change the generated code somewhat, and therefore itself affect performance.

Finally, note that if you are using Prism to analyze the performance of a program that includes timer calls, Prism performance data will include the overhead assigned to these calls; thus, the elapsed time reported by Prism will be somewhat greater than the elapsed time reported by the timing routines.

2.4 Calling CM Libraries

You can call routines from the standard CM libraries from within a C* program. Specifically:

- Call routines from the CMX11 library to display images on an X Window System. See Section 2.4.1.
- Call routines from the CM/AVS library to create modules that you can use in a distributed visualization application within the AVS visual programming environment. See Section 2.4.2.
- Call routines from the CMFS library to perform standard I/O functions — for example, reading data into the processing nodes from a scalable disk array or other I/O device. See Section 2.4.3.
- Call routines from the CMMD library to do node-level message passing on CM-5s with vector units. See Section 2.4.4.

- Call routines from the Connection Machine Scientific Software Library (CMSSL) to perform data parallel versions of standard mathematical operations such as matrix multiply and Fast Fourier Transform. You currently call the CM Fortran versions of CMSSL routines, as described in Section 2.5.

NOTE: These libraries are not available if you compile with the `-cmsim` option to run on a Sun-4.

2.4.1 CMX11

You can make calls to the CMX11 library from C*. This library provides routines that allow the transfer of parallel data between the CM and any X11 terminal or workstation. See the *CMX11 Reference Manual* for information on the C* interface.

2.4.2 CM/AVS

CM/AVS is an extension of the Advanced Visualization System (AVS) to the CM-5. AVS provides a graphic programming environment in which you can build a distributed visualization application. CM/AVS enables such an application to operate on data that is distributed on CM-5 nodes and to interoperate with data from other sources.

You can write your own modules for CM/AVS in C*. You can then combine these modules with standard CM/AVS and AVS modules to create your visualization application. See the *CM/AVS User's Guide* for information on how to call the CM/AVS routines in a C* program.

2.4.3 I/O

CM-5 C* provides synchronous parallel I/O support to the DataVault and scalable disk array via an interface to the CMFS functions; I/O to other devices is not currently supported. The interface is the same for CM-5s with and without vector units. For complete information on these functions, see the documentation for the CM-5's I/O system..

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:

- Do not include **<cm/paris.h>** or **<cm/cmtypes.h>**.
- You must include **<cm/cmfs.h>**.
- 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-200 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 in bytes. 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-2/200 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 documentation 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.
- 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).
- The `CMFS_lseek` routine when called from C* seeks into a file the number of bytes you specify multiplied by the number of positions in the current shape. The routine `CMFS_serial_lseek` seeks an absolute number of bytes into a file.

2.4.4 CMMD

You can call routines in the CMMD communication library from C*, as of CMMD Version 3.0. For complete information, see the *CMMD User's Guide* and *CMMD Reference Manual.*

To call CMMD routines, you must compile your C* program with the **-node** and **-cmmd_root** options; see Section 3.4.8.

2.5 Calling CM Fortran

You can call CM Fortran subroutines from within a C* program. This section describes how. See Section 3.6 for a discussion of how to link in the CM Fortran program and other required files.

2.5.1 Overview

To call a CM Fortran subroutine, do the following:

- Include the file `<csfort.h>`.
- The name of the C* main routine must be `MAIN_()`.
- Use the function `CMC_CALL_FORTRAN` to call one or more CM Fortran subroutines. You must convert the subroutine name to lowercase and add an underscore to the end of it.
- To pass a parallel variable to a subroutine, create a scalar variable of type `CMC_descriptor_t`. Call the function `CMC_wrap_pvar`, with a pointer to the parallel variable as an argument, and assign the result to the scalar variable you created. Pass this scalar variable to the CM Fortran subroutine when you call it via `CMC_CALL_FORTRAN`.
- Pass scalar variables to a CM Fortran subroutine by reference.
- After you are finished with a descriptor, free it by calling `CMC_free_desc` with the scalar variable as an argument.

2.5.2 In Detail

Include File

As mentioned in the overview, you must include the file `<csfort.h>` if your program includes a call to a CM Fortran subroutine.

Calling the Subroutine

To call a CM Fortran subroutine, use this syntax:

```
CMC_CALL_FORTRAN(subroutine_(args), ... );
```

where:

subroutine	is the name of the subroutine. It must be in lowercase (even if the original subroutine name is in uppercase), and you must add an underscore to the end of the subroutine name.
args	are the subroutine's arguments, if any.

To call multiple subroutines, separate them with commas within the argument list. For example:

```
CMC_CALL_FORTRAN(subroutine1_(), subroutine2_());
```

The subroutine is not constrained by the current shape or the context as established by the C* program. When the call to **CMC_CALL_FORTRAN** returns, however, both the shape and the context are what they were before the function was called.

What Kinds of Variables Can You Pass?

You can pass both parallel and scalar variables as arguments to a CM Fortran subroutine. The parallel variables you pass can be of any shape. The variables can be of these standard types:

```
signed int
signed long int
float
double
long double
```

In addition, `<csfort.h>` provides **`typedefs`** for two new types: **`CMC_complex`** and **`CMC_double_complex`**. The **`typedefs`** are defined as follows:

```
typedef struct{float real, imag;} CMC_complex;
typedef struct{double real, imag;} CMC_double_complex;
```

Use these types to pass variables that can be treated as complex numbers by CM Fortran.

Passing Parallel Variables

A two-step process is required to pass a C* parallel variable to a CM Fortran subroutine.

First, declare a scalar variable of type **`CMC_descriptor_t`**. For example:

```
CMC_descriptor_t desc_a;
```

Next, make this variable a descriptor for the parallel variable by calling the function **`CMC_wrap_pvar`**, with a pointer to the parallel variable as its argument, and assigning the result to the scalar variable. For example, if **`p1`** is the parallel variable you want to pass, call the **`CMC_wrap_pvar`** function as follows:

```
desc_a = CMC_wrap_pvar(&p1);
```

You can wrap a parallel variable of any shape.

You can then pass the descriptor to the CM Fortran subroutine. For example:

```
CMC_CALL_FORTRAN(subroutine_(desc_a));
```

The descriptor stores the address of the parallel variable, and the parallel variable is passed by reference in this way. The CM Fortran subroutine can then operate on the parallel variable referenced by the descriptor.

C* code can operate on the parallel variable even after it has been wrapped.

You can reuse a descriptor in a program, but first you must free it; see below.

One restriction on passing parallel variables: You cannot pass a member of a parallel structure or an element of a parallel array to CM Fortran. Only simple parallel variables can be passed.

Passing Scalar Variables

Pass scalar variables to a CM Fortran subroutine by reference. For example:

```
int s1;

CMC_CALL_FORTRAN(subroutine_(&s1));
```

Freeing the Descriptors

When you are through using a descriptor, free it by calling **CMC_free_desc** with the descriptor as the only argument. For example:

```
CMC_free_desc(desc_a);
```

You can free a descriptor to a parallel variable of any shape.

An Example

Below is a C* program that calls a CM Fortran subroutine.

```
#include <stdio.h>
#include <csfort.h>

shape [16384]S;
CMC_descriptor_t desc_a;
int s1;
int:S p1;

MAIN_()
{
  with (S) {
    s1 = 1;
    p1 = 1;
    desc_a = CMC_wrap_pvar(&p1);

    CMC_CALL_FORTRAN(fortran_op_(desc_a,&s1));

    CMC_free_desc(desc_a);
    printf("Result for last position is %d\n",
           [16383]p1);
    }
}
```

And here is the simple CM Fortran subroutine it calls:

```
subroutine fortran_op(a,s)
integer a(16384)
integer s

a = a + s

return
end
```

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. The example in Section 2.6.5 shows how to do this.

2.6 Writing Functions that Are Callable from CM Fortran

The previous section described how to call a CM Fortran routine from C*; this section describes how to write a CM-5 C* routine that can be called from a CM Fortran program. See Section 3.7 for information on compiling the CM Fortran program.

Functions callable from CM Fortran must include the header file **<csfort.h>**.

2.6.1 Names

Name the function according to Fortran conventions. Specifically:

- The name must end with an underscore (_).
- Any alphabetic characters in the name must be lowercase.

In the CM Fortran program that calls the function, the alphabetic characters may be either upper- or lowercase (since Fortran is not case-sensitive). The trailing underscore must be omitted.

2.6.2 Shapes

A C* shape and a CM Fortran "geometry" (that is, the shape and layout of a CM array) are not exactly the same, since C* shapes include context information as well as information on extents, rank order, and layout. Multiple shapes can share the same geometry.

Use the function **CMC_allocate_shape_from_desc** to dynamically allocate a shape in a C* function to be called from CM Fortran. It takes as an argument a descriptor for a CM array; see the next section. It returns a shape whose geometry is the same as that of the CM array, and whose context initially selects all elements of the geometry.

You must free this shape via the **deallocate_shape** routine before the C* function returns.

2.6.3 Parallel Arguments

Parallel variables are the equivalent of CM arrays in CM Fortran. The C* function must declare CM arrays incoming from the CM Fortran program with the type **CMC_descriptor_t**; this is a *descriptor* for the CM array. The C* function then turns the descriptors into pointers using this function:

```
void:void *CMC_unwrap_pvar(CMC_descriptor_t, shape);
```

(This is the opposite of the **CMC_wrap_pvar** function described in Section 2.5.2.) This function returns a pointer to a parallel variable, given an array descriptor and a shape. The geometry of the array must match the geometry of the shape, or a run-time error is signaled. A function to check that the geometries match is:

```
bool CMC_same_geometry(CMC_descriptor_t, shape);
```

Typically, called functions will create a single shape per geometry, but this interface allows otherwise for flexibility.

2.6.4 Scalar Arguments

All variables are passed by reference in CM Fortran, so the C* function must receive pointers to each of its scalar arguments. The data-type correspondence between CM Fortran and C* is shown in Table 1.

Table 1. Data type correspondence between CM Fortran and C*.

CM Fortran	C*
`REAL`	`float`
`DOUBLE PRECISION`	`double`
`COMPLEX`	`CMC_complex_t`
`DOUBLE COMPLEX`	`CMC_double_complex_t`
`CHARACTER`	`char *`

Character arguments are not guaranteed to be null-terminated, as C often expects. The lengths of all character arguments are appended to the end of the argument list (these lengths are passed by value, not by reference).

`LOGICAL`, `INTEGER`, and `DOUBLE PRECISION` values are returned by value just as in CM Fortran. CM Fortran turns functions that return complex values into subroutines that pass a pointer to a place to write the result. Functions that return character values are turned into calls whose first argument points to a character variable in which to place the result, and whose second argument is the length (passed by value).

This is a CM Fortran program that passes scalar values to a C* program:

```
LOGICAL l
INTEGER i
REAL r
DOUBLE PRECISION d
COMPLEX c
DOUBLE COMPLEX z
CHARACTER*5 ch

l = .TRUE.
i = 2
r = 3.0
d = 4.0
c = (5.0,-5.0)
```

```
z = (6.0,-6.0)
ch = "abcde"

PRINT *,"Main Program (CMF)"
PRINT *,'l = ',l
PRINT *,'i = ',i
PRINT *,'r = ',r
PRINT *,'d = ',d
PRINT *,'c = ',c
PRINT *,'z = ',z
PRINT *,'ch = "',ch,'"'
CALL CMF2C(l,i,r,d,c,z,ch)

END
```

And here is a C* program that it calls. (Note that there is nothing parallel about this program; it merely uses the C* header file.)

```
#include <stdio.h>
#include <csfort.h>

void cmf2c_(lp, ip, rp, dp, cp, zp, chp, chl)     /* fn name
                         lowercased & underscore appended */
int *lp;
int *ip;
float *rp;                         /* all args are pointers */
double *dp;
CMC_complex_t *cp;
CMC_double_complex_t *zp;
char *chp;
int chl;                      /* last arg for string length */
{
    int i;
    float r;
    double d;

    putchar('\n');
    printf("CMF2C (C)\n");
    printf("l = %c\n", (*lp) ? 'T' : 'F');
    printf("i = %d\n", *ip);
    printf("r = %f\n", *rp);
    printf("d = %f\n", *dp);
    printf("c = (%f,%f)\n", cp->real, cp->imag);
    printf("d = (%f,%f)\n", zp->real, zp->imag);
    printf("s = %.*s\n", chl, chp);
    putchar('\n');

    /* modify some args */
    *lp = !(*lp);
    cp->real = -cp->real;
```

```
        cp->imag = -cp->imag;
        zp->real = -zp->real;
        zp->imag = -zp->imag;
        chp[0] = ' ';

    }
```

2.6.5 An Example

Here is a simple CM Fortran program:

```
REAL, ARRAY(7) :: J, I
REAL f
J = I + f
CALL CSTAR_FUNCTION(J,I)
END
```

The C* program below both calls this CM Fortran program and contains the C* function that the CM Fortran program in turn calls:

```
MAIN_()
{
    shape [7]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]i);
         printf("\n");
    for(n=0; n < positionsof(s); n++) printf("%f ", [n]j);
         printf("\n");
}


/* Eventually, the following stub will automatically be gener-
ated by the compiler given the appropriate declaration of
CMF_ROUTINE. For now, we need to write it by hand. */

#include <csfort.h>

CMF_ROUTINE(jp, ip, f)
float: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);
}


/* This is an example of a C* function to be called by CMF.
Again, this is something that will eventually be handled
automatically. The above  #include <csfort.h> is necessary for
this as well. */

void cstar_function_(CMC_descriptor_t ip_desc,
CMC_descriptor_t jp_desc)
{
    shape s = CMC_allocate_shape_from_desc(ip_desc);
    float:s *ip = CMC_unwrap_pvar(ip_desc, s);
    float:s *jp = CMC_unwrap_pvar(jp_desc, s);

    *ip = -*ip;
    *jp = -*jp;

    deallocate_shape(&s);
}
```

Chapter 3

Compiling a C* Program

This chapter describes how to compile and link a C* program. It is organized as follows:

- Section 3.1 describes the compilation process.
- Section 3.2 describes basic options of the `cs` command.
- Section 3.3 describes `cs` options in common with the `cc` command.
- Section 3.4 discusses options more likely to be used by advanced programmers.
- Section 3.5 discusses issues involved in linking CM-5 C* programs.
- Section 3.6 explains how to compile a C* program that calls a CM Fortran subroutine.
- Section 3.7 discusses compiling a CM Fortran program that calls a C* function.
- Section 3.8 discusses symbols for which `cs` provides `#defines`.
- Section 3.9 describes how to use `make` with C* programs.

3.1 The Compilation Process

To compile a C* program, use the `cs` command. Typically, the compiler is available on a CM-5 partition manager, or on a designated server on the network. If the compiler is available on a server, it is generally a good idea to compile on it rather than the partition manager, to avoid tying up the CM-5.

The `cs` command takes a C* source file (which must have a `.cs` suffix) and produces an executable load module. The command also accepts `.c` source files, `.o` output files, `.s` assembly files, `.dp` DPEAC files, and `.a` library files, but all parallel code must be in a `.cs` file.

NOTE: `.c` files are by default passed to the Sun C compiler. This means that ANSI C features are not allowed in `.c` files, but the code will be compiled more efficiently and faster, since the C* compiler is not an efficient serial code compiler.

3.2 Basic Options

The options accepted by `cs` include some that are specific to C* and the Connection Machine system, as well as versions of `cc` options. This section describes commonly used options. All options are listed in Table 1.

Table 2. C* compiler options.

Option	Meaning
Basic options:	
-cm2 **-cm200** **-cm5**	Specify the version of the compiler to use.
-cmsim	Compile to run on a Sun-4.
-h **-help**	Give information about `cs` without compiling.
-sparc	Compile to run on a CM-5 without vector units.
-version	Print the compiler version number.
-vu **-vecunit**	Compile to run on a CM-5 with vector units.

Option	Meaning
Options in common with `cc`:	
`-c`	Compile only.
`-D`*name* `[=`*def*`]`	Define a symbol name to the preprocessor.
`-g`	Produce additional symbol table information for debugging with Prism, at the expense of slower performance.
`-I`*dir*	Search the specified directory for `#include` files.
`-L`*dir*	Add *dir* to the list of directories in the object library search path.
`-l`*lib*	Link with the specified library.
`-o` *output*	Change the name of the final output file to *output*.
`-U`*name*	Undefine the C preprocessor symbol *name*.
Advanced options:	
`-cc` *compiler*	Use the specified C compiler when compiling `.c` files.
`-cmdebug`	Compile for debugging (requires less execution time than `-g`).
`-cmmd_root` *dir*	Use *dir* as the location of the CMMD library (for use with `-node`.)
`-cmprofile`	Compile for Prism performance analysis.
`-dirs`	List the compiler's `bin`, `lib`, `include`, and `temp` directories.
`-dryrun`	Show, but do not execute, compilation steps.
`-force`	Force `.c` files through the C* compiler.
`-keep` *ext*	Keep the intermediate `.s`, `.o`, or `.dp` file.
`-node`	Link to run on a single node.
`-overload`	Display symbol names used by the compiler to call overloaded functions.

Option	Meaning
`-S`	Create an assembly source file for each input source file. (No assembly or linking performed.)
`-temp` *directory*	Change the location of temporary files to *directory*.
`-v` `-verbose`	Show the compilation and linking steps as they are executed.
`-warn`	Suppress warnings from the C* compiler.
`-Wimplicit`	Display a warning when a function is called before it is declared or defined.
`-Z`*comp switch*	Pass option *switch* to component *comp*, where *comp* is `cc`, `ld`, or `cmld`.

3.2.1 Choosing the Compiler: The –cm2, –cm200, –cmsim, and –cm5 Options

If you have more than one CM model at your site, separate C* compilers may be available for each machine. In any case, a version of the compiler is available for creating programs to run on a Sun-4. You invoke this version of the compiler via the `-cmsim` option; see below.

Your system administrator determines which version of the compiler you get by default when you issue the `cs` command. You can override this either by setting the environment variable `CS_DEFAULT_MACHINE` to the machine you want (`cm2`, `cm200`, `cm5`, or `cmsim`), or by specifying the `-cm2`, `-cm200`, `-cm5`, or `-cmsim` option on the `cs` command line. Using the command-line option also overrides the setting of the environment variable.

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. Typically you would use this option to try out a program on a smaller data set before running it on a Connection Machine.

You can determine the default compilation target by simply typing `cs` at your UNIX prompt. If the resulting help message begins

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

the target is a CM-5 without vector units. If it begins

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

the target is a CM-5 with vector units.

See also the discussion of the -vu and -sparc options in Section 3.2.4.

Note that if you specify -cm2 or -cm200, you should consult the user's guide for CM-200 C* for a discussion of supported compiler options.

3.2.2 Getting Help: The -help Option

Specify -help or -h to print a summary of available command-line options for cs, without compiling.

3.2.3 Printing the Version Number: The -version Option

Specify the -version option to cause cs to print its version number before compiling. If you do not specify a source file, cs simply prints the version number and exits.

3.2.4 Compiling for CM-5s with or without Vector Units: The -vu, -vecunit, and -sparc Options

Specify the -vu or -vecunit option to compile your program to run on a CM-5 that has vector units.

Specify the -sparc option to compile your program to run on a CM-5 without using the vector units.

The -vu, -vecunit, and -sparc options imply -cm5; if you specify any of them, you do not have to specify -cm5 in addition.

If you specify -cm5 but omit one of these options, you get the default for your site; typically, this is -vu.

3.3 Options in Common with cc: The –c, –D, –g, –I, –l, –L, –o, and –U Options

The C* compiler allows you to specify the `cc` options `-c`, `-D`, `-g`, `-I`, `-l`, `-L`, `-o`, and `-U` on the `cs` command line. See Table 2 for a brief description of these options; for more information, consult your documentation for `cc`.

Include the `-g` option if you want to debug the compiled program using Prism; see Chapter 5. Using the `-g` option increases execution time considerably; use the `-cmdebug` option instead if this matters. See Section 3.4.2.

3.4 Advanced Options

This section describes `cs` options that would typically be used by an advanced programmer. All options are listed in Table 2, above.

3.4.1 Using Another C Compiler: The –cc Option

The C* compiler works in conjunction with the standard C compiler available on your Sun partition manager or compile server. *The use of C* with other C compilers is not supported and can lead to incorrect results.* However, you can use another compiler if you want to, by including the `-cc` switch, followed by the name of the compiler. For example, to use the Gnu C compiler, specify the `-cc` option as in this example:

```
% cs -cc gcc myfile.c
```

3.4.2 Debugging Your Program: The –cmdebug Option

Use the `-cmdebug` option (instead of the standard `-g` option) to create symbol table information for debugging your program. The `-cmdebug` option speeds up execution time, at the expense of making debugging somewhat less precise. (For example, you may not be able to set a breakpoint at every executable line of code.)

3.4.3 Obtaining Performance Data: The –cmprofile Option

Use the **-cmprofile** option if you want to run your program under Prism to obtain performance data. See Chapter 5 for more information about Prism.

3.4.4 Displaying the bin, lib, include, and temp Directories: The –dirs Option

Use the **-dirs** option to find out where the compiler searches for binaries, libraries, include files, and temporary files. It produces output like this:

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

The binary search path is:

1. the "**bin_dir**" directory specified in this message
2. your **$PATH**
3. **/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**

3.4.5 Displaying Compilation Steps: The -dryrun Option

Specify `-dryrun` to cause `cs` to show, but not carry out, the commands to be executed in compiling and linking.

3.4.6 Putting .c Files through the C* Compiler: The -force Option

Specify `-force` to put `.c` files through the C* compiler. Otherwise, such files are passed unread to the C compiler. You might want to specify `-force` to take advantage of the C* compiler's type checking of prototyped function declarations.

3.4.7 Keeping an Intermediate File: The -keep Option

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

- `s`, to keep the assembly-language source file
- `o`, to keep the object file
- `dp`, to keep the DPEAC assembly-language file; 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.

Note for users of CM-200 C*: `-keep c` is not allowed because CM-5 C* compiles directly to assembly code.

3.4.8 Using CMMD: The -node and -cmmd_root Options

To use the CMMD message-passing library from a C* program, you must link the program with the `-node` option; this specifies that the program is to be linked so that copies of it run separately on the individual nodes. This option is supported only if you also specify `-vu` to run on the vector units.

If the CMMD library has not been installed in the standard place at your site, use the `-cmmd_root` option, followed by a directory name, to specify its location. 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, set the environment variable `CMMD_ROOT` to the correct pathname.

3.4.9 Displaying Names of Overloaded Functions: The –overload Function

Use the `-overload` option to cause the compiler to display informational messages listing the symbol names it uses internally for overloaded functions; it displays such a message once for every call to an overloaded function being compiled. Knowing these symbol names is necessary if you want to invoke such a function directly using Prism.

3.4.10 Creating Assembly Source Files: The –S Option

Use the `-S` compiler option to create assembly source files for each input source file. For example:

```
% cs -S foo.cs
```

produces the files `foo.s` (for the program that runs on the partition manager) and `foo.pe.s` (for the program that runs on the processors). See Section 3.5 for more information.

You cannot combine the `-S` option with either the `-c` option or the `-o` option.

3.4.11 Changing the Location of Temporary Files: The –temp Option

Use the `-temp` option, followed by the pathname of a directory, to cause temporary files used during C* compilation to be created in that directory. Issue the `cs` command with the `-dirs` option to find out the standard location in which they are created; see Section 3.4.4.

3.4.12 Turning On Verbose Compilation: The –v or –verbose Option

Specify `-verbose` or `-v` to display informational messages as the compilation proceeds. This can be useful if you want to see which part of the compilation process produced an error message.

3.4.13 Turning Off Warnings: The –warn Option

Specify `-warn` to suppress warnings produced during C* compilation.

3.4.14 Turning On Warnings about Undeclared Functions: The –Wimplicit Option

Specify `-Wimplicit` to tell the compiler to print a warning when you call a function that has not previously been declared or defined.

3.4.15 Specifying Options for Other Components: The –Z Option

Use the `-Z` option to specify a `cc`, `cmld`, `ld`, `dpas`, or `as` option that `cs` does not recognize. Use `-Zld` only when you specify the `-cmsim` option. Use `-Zdpas` only when compiling for the vector units.

These options are passed directly to the specified component without any interpretation by `cs`. Type `-Z`, followed by the component name, followed by the option. There is no space between `-Z` and the component name; leave at least one space between the component name and the option.

For example, specify

```
% cs -Zcc -w myfile.c
```

to suppress `cc` warning messages.

3.5 Linking

This section discusses issues in linking C* programs.

3.5.1 Names of C* Run-Time Libraries

You need to be aware of the names of the C* run-time libraries only if you are using a command other than `cs` to link — for example, when linking a C* program that calls CM Fortran; see Section 3.6.

The names of the run-time libraries 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`.

3.5.2 Intermediate Files

The C* compiler and the CM-5 linker, `cmld`, generate a single output file that combines a scalar and a parallel executable program. As intermediate output, however, the compiler generates separate scalar and parallel files. For example:

- With the `-S` option, the compiler generates two assembly files: for example, `myprogram.s` and `myprogram.pe.s`.
- With the `-c` option, the compiler generates two object files: for example, `myprogram.o` and `myprogram.pe.o`.

NOTE: The parallel files can have `.pn` extensions instead of `.pe`.

However, the linker generates only one executable file: for example, `a.out`. There is no file `a.out.pe` corresponding to the parallel intermediate files.

If you work with intermediate files — explicitly linking object files, for instance — you need only specify the scalar file. The corresponding parallel file is linked automatically.

If you wish to disable the automatic processing of the parallel (.pe) intermediate file when the corresponding scalar file is specified, set the environment variable **CS_AUTO_PE_FILES** to **0**. Any nonzero value for this variable leaves the feature enabled.

In either case, recall that the separate intermediate files exist. If you copy or move intermediate files to another directory, be sure to move both the scalar and the parallel file. See below for more information on creating libraries for **.pe** files.

3.5.3 Creating Libraries

If you put object files in a library, you must remember to put the **.pe** object files in a separate node library too; see the previous section. Note these points in creating the library for **.pe** files:

- The node library's name must begin with the same name as the library containing the corresponding scalar object files, and it must end in **_pe.a** (or **_pn.a**).
- It must be in the same directory as the one that contains the corresponding scalar object files.

When linking, you then need only specify the library containing the scalar object files.

For example, these commands create object files and store them in libraries in your current working directory:

```
% cs -c -cm5 foo.cs bar.cs
% ar q libmine.a foo.o bar.o; ranlib libmine.a
% ar q libmine_pe.a foo.pe.o bar.pe.o; \
  ranlib libmine_pe.a
```

You can then link the programs in these libraries with another program by issuing a command like this:

```
% cs -cm5 prog.cs libmine.a
```

NOTE: In the example shown above, the library containing the scalar object files can also be called **libmine_sp.a**. C* will still find the corresponding **_pe** or **_pn** library automatically.

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

If your program includes a call to a CM Fortran subroutine, as described in Chapter 2, follow the instructions in this section.

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
   ```

   NOTE: If you are using CM Fortran Version 2.1, you cannot compile with the `-noaxisorder` or `-nopadding` option.

3. Issue `cmf` again to link, using the applicable format:

   - 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`, `-cmdebug`, or `-cmprofile`, you must also specify one of the following before `-lcs_cm5_`*xxx*:

   - `-lprism5` (when the C* program is compiled with `-sparc`)
   - `-lprism5dp` (when the C* program is compiled with `-vu`)
   - `-lprism_sim` (when the C* program is compiled with `-cmsim`)

   For example:

   ```
   % cmf testcs.o testfcm.o -vu -Llib_dir -lprism5dp \
     -lcs_cm5_vu_sp
   ```

The result is an executable load module that you can execute as you normally would.

3.7 Compiling CM Fortran Programs that Call C*

If you are using Version 2.1 Beta or later of the CM Fortran compiler, you must compile the CM Fortran program with the **-padding** option. Otherwise, compile the CM Fortran as you normally would.

3.8 Symbols

The **cs** command predefines these preprocessor symbols to 1 where appropriate in C* code:

unix	Any UNIX system
sun	Sun only
sparc	Sun-4 only
__CM5__	CM-5 only
__CM5_SPARC__	CM-5 with SPARC processors only
__CM5_VECUNIT__	CM-5 with vector units
__CSTAR__	This is a C* compiler
__STDC__	This is an ANSI C compiler

3.9 Using make

The UNIX **make** utility can make object (**.o**) files from C* **.cs** files, just as it does with **.c** files. The only requirement is that this code must appear somewhere in the makefile:

```
CS = cs
CSFLAGS = $(CFLAGS)
.SUFFIXES: .cs
.cs.o:
        $(CS) -c $(CSFLAGS) $<
```

Chapter 4

Executing a C* Program

Once a C* program has been compiled and linked, you can execute the output file on a CM-5. This chapter gives an overview of how to execute a program on a CM-5. Section 4.1 describes how to execute the program directly. Section 4.2 describes how to execute the program in batch mode. Section 4.3 describes how to execute a C* program on a Sun-4.

Note that you can execute C* programs within Prism, the CM's programming environment. See Chapter 5 for information on Prism.

NOTE: Your site may be using DJM (Distributed Job Manager), a batch system/ job manager. If so, see your local documentation for DJM to learn how to execute programs on the CM.

4.1 Executing the Program Directly

To execute a program directly on a CM-5, you must gain access to it first. To do this, you must know the name of one of its partition managers. You can find out the names of partition managers from your system administrator; the system administrator can also tell you if you have permission to use a particular partition manager.

From your terminal or workstation on the network, you can gain access to the partition manager via the UNIX command `rlogin` or `rsh`. For example, to log in to the partition manager Mars, issue the command:

```
% rlogin mars
```

You can then execute your program on the partition just as you would any UNIX command or program. For example:

```
% a.out
```

Use the `rsh` command to execute the program without logging in to the partition manager. Simply specify the name of the partition manager, followed by the name of the executable program, on the `rsh` command line. For example:

```
% rsh mars a.out
```

You are then returned to your local UNIX shell.

4.2 Obtaining Batch Access

You can use NQS, a standard batch system, to obtain batch access to the CM-5. For complete information on NQS, see the manual *NQS for the CM-5*.

In NQS, you submit your program as a request to a queue. The queue may be associated with a partition, in which case the request is generally executed when it reaches the head of the queue. Or, the queue could send the request to another queue for execution.

NQS is configured differently at different sites. To find out what queues exist at your site and when they are active, ask your system administrator, or issue this command:

```
% qstat -x
```

4.2.1 Submitting a Batch Request

Use the `qsub` command to submit a batch request for execution via a queue in NQS. You can submit multiple programs as one batch request. There are two ways of specifying the programs to be executed:

- Put their names in a script file, and specify the name of the script file on the `qsub` command line. For example, the file `myprogram_script` could contain these names of executable C* programs:

```
myprogram1
myprogram2
myprogram3
```

You can then submit these programs for execution by issuing this command:

```
% qsub myprogram_script
```

- Enter the names of the files from standard input. Put the names of the programs on separate lines, and type Ctrl-d at the end to signal that there is no more input. For example:

```
% qsub
  myprogram1
  myprogram2
  myprogram3
  Ctrl-d
```

You can also issue other commands as part of the request.

4.2.2 Options for qsub

This section describes several of the most commonly used options for **qsub**. See its on-line man page for a discussion of all its options.

Specifying a Queue

Use the **-q** option to specify the name of the queue to which the request is to be submitted. If you omit this, the request is sent to the default queue (if one has been set up).

Receiving Mail

Use the **-mb** option to specify that mail is to be sent to you when the request begins execution. Use **-me** to have mail sent to you when the request ends execution.

Setting the Priority

Use the `-p` option, followed by an integer from 0 through 63, to set a priority for this request in its queue. 63 is the highest priority, and 0 is the lowest priority. The priority determines the request's position in the queue. If you don't set a priority, the request is assigned a default priority.

Specifying Output Files

Use the `-o` option, followed by a pathname, to specify the file to which output of the batch request is to be sent. Use the `-e` option to specify the pathname for the standard error output. If you omit these options, the output is sent to default files based on an ID number assigned to the request by the batch system.

4.2.3 Other Commands

There are other commands you can use in working with NQS:

- Use the `qdel` command to delete a batch request from a queue.
- Use the `qstat` command to obtain information about batch requests in a queue.

4.2.4 UNIX Batch Commands

The CM-5 also supports the standard UNIX batch commands `at` and `batch`. For example:

```
% at 0815am Jan 24
  primes
  Ctrl-d
```

This causes the program `primes` to be executed at 8:15 a.m. on January 24th.

4.3 Executing a C* Program on a Sun-4

If you compiled your program with the `-cmsim` option, you can run your program on a Sun-4. You run the program just as you would any other program on a Sun-4. Performance, of course, will be much worse than on a CM-5.

Chapter 5

Debugging a C* Program

Use Prism, the programming environment for the Connection Machine system, to debug your C* program. You can also use Prism to develop and execute your program. This chapter gives a brief overview of Prism. For complete information, see the *Prism User's Guide* and *Prism Reference Manual*. In particular, note that there may be some limitations in Prism's support of C*; for example, it may not recognize all C* syntax in expressions.

NOTE: If you compiled your C* program to run on the nodes and use the CMMD message-passing library, use `pndbx` to debug the program. For information on using `pndbx`, see the *CMMD User's Guide* and the `pndbx` release notes.

Note also that the use of other debuggers is not supported for CM-5 C*.

5.1 Compiling for Prism

To use Prism for debugging your C* program, compile the program with the `-g` or `-cmdebug` option. To use it for performance analysis (see Section 5.7), compile with the `-cmprofile` option.

5.2 Starting Prism

Prism has two modes:

- Graphical Prism operates on terminals or workstations running the X Window System.
- Commands-only Prism lets you operate on any terminal, but without the graphical interface.

5.2.1 Graphical Prism

Before starting Prism, make sure your **DISPLAY** environment variable is set for the terminal or workstation from which you are running X. For example, if your workstation is named Valhalla, you can issue this command (if you are running the C shell):

```
% setenv DISPLAY valhalla:0
```

To start Prism, issue the command **prism** at your UNIX prompt. To load an executable program automatically into Prism, specify its name on the Prism command line. For example:

```
% prism primes.x
```

This displays the main Prism window, as shown in Figure 1.

5.2.2 Commands-Only Prism

To start commands-only Prism, issue the **prism** command with the **-C** option:

```
% prism -C
```

After an introductory message, you receive this prompt:

```
(prism)
```

You can issue any Prism command at the prompt. The rest of this chapter focuses on graphical Prism; however, all of the functionality of graphical Prism is available by issuing commands from commands-only Prism, except for features that require graphics.

5.3 Using Prism

Figure 1 shows the main Prism window, with a source file loaded.

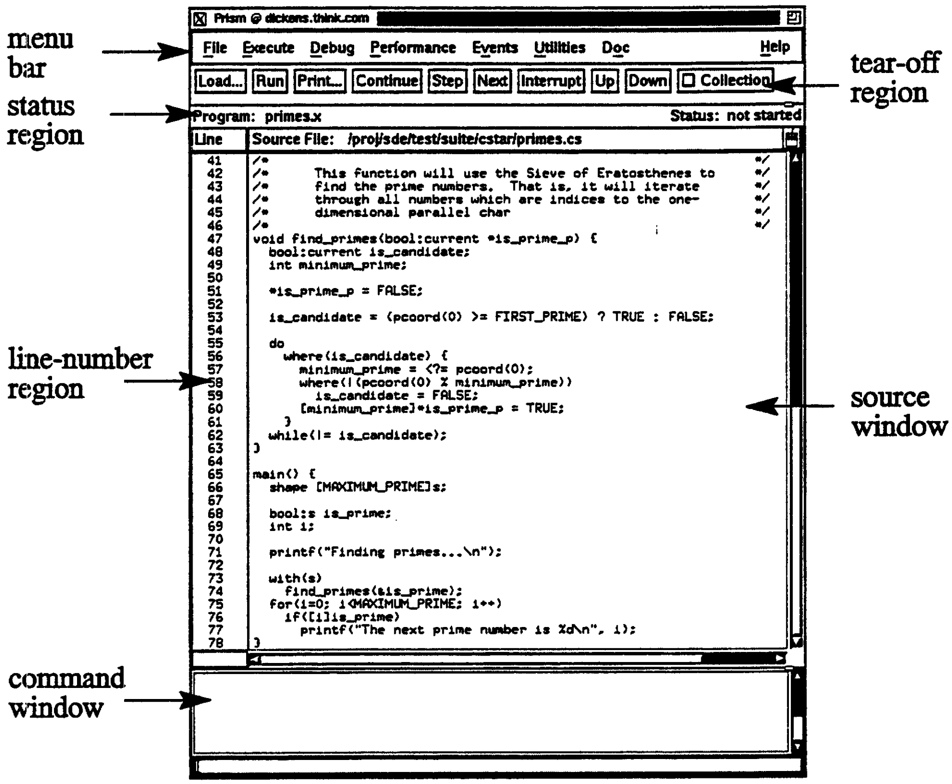

Figure 1. The main Prism window.

Clicking on items in the **menu bar** at the top of the Prism window displays pull-down menus that provide access to most of Prism's functionality.

You can add frequently used menu items and commands to the **tear-off region,** below the menu bar, to make them more accessible. Clicking on a button in the tear-off region is equivalent to selecting the menu item or issuing the command.

The **status region** displays the program's name and messages about the program's status.

The **source window** displays the source code for the executable program. You can scroll through the source code and display any of the source files used to compile the program.

The **line-number region** is associated with the source window. You can click to the right of a line number in this region to set a breakpoint at that line.

The **command window** at the bottom of the main Prism window displays messages and output from Prism. You can also type commands in the command window rather than use the graphical interface.

5.4 Loading and Executing Programs

As mentioned above, you can load an executable program when you start up Prism. You can also load it after Prism has started.

Once the program is loaded, you can run it or step through it, and interrupt execution at any time. You can also attach to a running program or associate a core file with a program.

5.5 Debugging

Prism lets you perform standard debugging operations such as:

- Setting breakpoints and traces
- Displaying and moving through the call stack
- Examining the contents of memory and registers

It also contains an *event table* that provides a unified method for controlling the execution of a program. For example, using the event table, you can specify that execution is to stop at line 33, and Prism is to print out the values of parallel variable **x**.

5.6 Visualizing Data

In debugging a C* program, it is often important to obtain a visual representation of the values of a parallel variable's elements. In Prism, you can create *visualiz-*

ers to display these values graphically. Prism visualizers provide a variety of representations, including:

- *Text*, where the values are shown as numbers or characters.

- *Colormap*, where each value is mapped to a color, based on a range of values and a color map you specify. (This representation is available only on color workstations.)

- *Dither*, where groups of elements are assigned different numbers of black and white pixels, depending on their values. This gives an impression of gray shading on a black-and-white display.

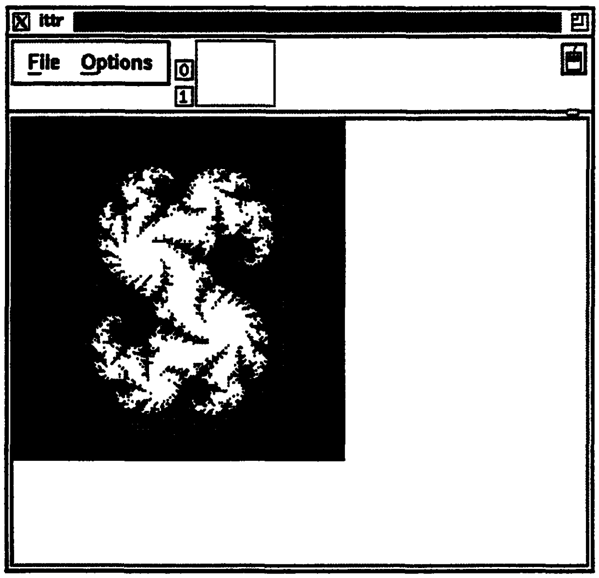

Figure 2. A dither visualizer.

- *Threshold*, where each value is mapped to either black or white, based on a cutoff value that you can specify.

- *Surface*, which renders the 3-dimensional contours of 2-dimensional data.

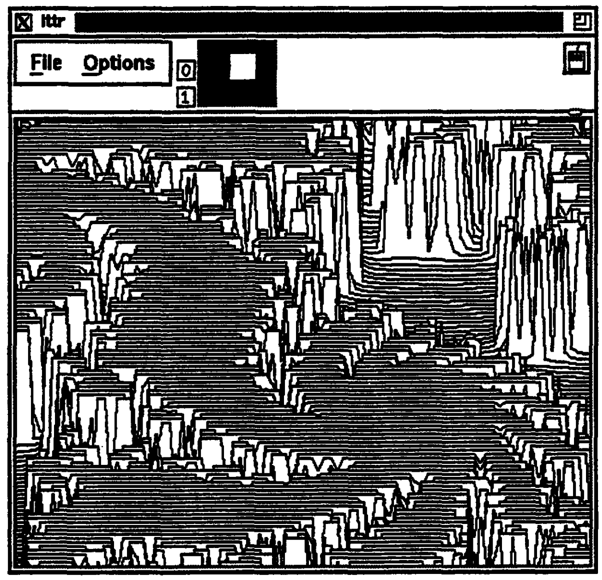

Figure 3. A surface visualizer.

- *Graph*, which displays values as a graph, with the index of each element plotted on the horizontal axis and its value on the vertical axis. A line connects the points plotted on the graph.

If the data doesn't all fit in the display window, you can pan through it. You can also increase the size of the fields being displayed; this gives the effect of zooming in on an area.

You can specify that the visualizer is to update whenever the program stops execution. And you can take a snapshot of the visualizer, so that you can compare it with later updates.

5.6.1 Visualizing Parallel Objects

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]
```

NOTE: CM-5 C* lets you obtain the memory address and stride information via functions in your program; this in turn lets you construct a pointer to a parallel variable. See the *C* Programming Guide* for more information.

5.7 Analyzing a Program's Performance

Prism provides performance data essential for effectively analyzing and tuning C* programs. The data includes:

- processing time on the partition manager
- processing time on the nodes
- time spent doing various forms of communication

The performance data is displayed as histogram bars and percentages for each resource. For each resource, you can also obtain data on usage for each procedure and each source line in the program. You can save the performance data in a file and redisplay it at a later time.

In addition, a performance advisor provides information about and analysis of the data Prism collects.

Appendix

Man Pages

This appendix contains the text of man pages for cs and for certain C* header files. These man pages are also available on-line.

cs

C* compiler for the CM-5 Connection Machine system

SYNOPSIS

cs [option] ... *file* ...

DESCRIPTION

cs invokes the CM-5 C* compiler. File names ending in **.cs** are treated as C* source files; file names ending in **.c** are treated as C source files and are passed directly to the *cc* compiler; file names ending in **.o** are treated as object files and are passed directly to the linker; and file names ending in **.a** are treated as object libraries and are passed directly to the linker.

The resulting executable program can be run on a CM-5 Connection Machine system. An option is also available to run the program on a Sun-4.

cs accepts a number of the options and filename endings that *cc* accepts, plus many specific to *cs*.

OPTIONS

Options specific to *cs*

–cc*cmdname* Use *cmdname*, rather than *cc*, as the compiler to perform C compilations.

–cm*X* Compile and link for CM model *X*. Accepted values of *X* are 2, 200, and 5. The values 2 and 200 invoke the CM-200 C* compiler (if it's installed); see the CM-200 *cs* man page for information about this compiler. There is a site-specific default, which would typically be 5 for a CM-5 site.

–cmdebug Compile for debugging. The program will run faster when compiled with **–cmdebug** than when compiled with **–g**, but debugging will be somewhat less precise.

–cmmd_root *dir* Use *dir* as the CMMD root directory, when compiling as a node-level program. The environment variable **CMD_ROOT** can also be used to specify this directory.

–cmprofile Produce performance analysis code. Performance data can then be generated and viewed using *prism*.

–cmsim Compile and link to run on a Sun-4.

–dirs Print a message showing where the compiler searches for binaries, libraries, include files, and temporary files.

–dryrun Show, but do not execute, all commands to be executed in compiling and linking.

–force Force input files with the **.c** suffix to be passed through the C* compiler rather than the *cc* compiler.

–help Print a summary of available command line switches without compiling.

–h Synonym for **–help**.

–keep *ext* Keep the intermediate file with the extension *ext*. Specify **s** to keep the assembly-language source code. Specify **o** to keep the object file. Specify **dp** to keep a DPEAC assembly source file (ending in **.pe.dp**) for code that runs on vector units. Using this option does not inhibit assembly or linking.

–node Compile and link to run as a node-level program.

–overload For each call to an overloaded function, print the actual symbol name of the function called. This is useful for debugging.

–q Suppress progress reports normally produced by the compiler.

–sparc Compile and link to run on a CM-5 without vector units. This option implies **–cm5**.

–temp *dir* Change the location in which C* temporary files are created from the standard temp directory to *dir*. See tmpnam(3).

–v Show the commands executed as compilation and linking proceeds.

–vecunit Compile and link to run on a CM-5 with vector units. This option implies **–cm5**.

-verbose Synonym for -v.

-version Print the C* compiler version number before compiling.

-vu Synonym for **-vecunit**.

-warn Suppress warnings from the C* compilation phase.

-w Synonym for **-warn**.

-Wimplicit Print warnings when you call a function that has not previously been declared or defined.

-Z*comp switch* Pass option *switch* to *comp*, where *comp* is **cc**, **cmld**, **ld** (if the **-cmsim** option is specified), **dpas**, or **as**. For example, **-Zcc -O** turns on the C compiler's optimizer.

Options in common with *cc*

-c Suppress the linking phase of the compilation and force an object file to be produced even if only one program is compiled.

-D*name*[=*def*] Define the symbol *name* to the preprocessor. If *=def* is not supplied then *name* is defined with a value of 1.

-g Have the compiler produce additional symbol table information for use specifically with *prism*. This slows execution considerably.

-I*dir* Seek **#include** files whose names do not begin with "/" in this directory. The compiler looks for include files first in directories named in **-I** options, then in the include directory listed by the **-dirs** option, then in **/usr/include**.

-L*dir* Add directory *dir* to the list of directories in the object library search path.

-l*x* Look for library **lib*x*.a** in the library search path. The libraries specified via the **-L option** are searched first, followed by the library directory listed by the **-dirs** option, then libraries specified by the environment variable **LD_LIBRARY_PATH**, if set. Finally, *ld* tries the directories **/lib**, **/usr/lib**, and **/usr/local/lib**. Libraries are searched in the order in which their names are encountered, so the placement of a **-l** is significant.

–o *output*	Name the final output file *output*. This option can also be used to rename the output from the **–c** and **–S** options. It applies to parallel **.pe** files as well as scalar files; see **LINKING**, below, for more information on these files. If this option is used, the file **a.out** will be left undisturbed. If this option is not used, the output file is called **a.out**, unless the **–c** or **–S** option is also specified.
–S	Create assembler source files (ending in .S) as output.
–U*name*	Undefine the preprocessor symbol *name*.

PRE-DEFINED PREPROCESSOR SYMBOLS

The C* compiler provides the following default preprocessor symbols, each defined as **1**.

unix	Any UNIX system
sun	Sun only
sparc	Sun-4 only
__CM5__	CM-5 only
__CM5_SPARC__	CM-5 with SPARC processors only
__CM5_VECUNIT__	CM-5 with vector units
__CSTAR__	This is a C* compiler
__STDC__	This is an ANSI compiler

LINKING

The C* compiler and the CM-5 linker, *cmld* (or *ld* if you specify the **–cmsim** option), generate a single output file that combines a scalar executable program for the partition manager and a parallel executable program for the nodes. As intermediate output, however, the compiler generates separate files for the partition manager and the nodes. For example:

With the –S option, the compiler generates both parallel and scalar assembly files: for example, **myprogram.s** and **myprogram.pe.s**.

With the **–c** option, the compiler generates two object files: for example, **myprogram.o** and **myprogram.pe.o**.

However, the linker generates only one executable file: for example, **a.out**. There is no file **a.out.pe** corresponding to the parallel intermediate files.

If you work with intermediate files — explicitly linking object files, for instance — you need only specify the scalar file (for example, **myprogram.o**). The corresponding parallel file is linked automatically.

In any case, recall that the separate intermediate files exist. If you copy or move intermediate files to another directory, be sure to move both the scalar and the parallel files.

If you put object files in a library, you must remember to put the **.pe** object files in a library too. The library's name must begin with the same name as the library containing the corresponding scalar object files, and it must end in **_pe.a**. It must be in the same directory as the one that contains the corresponding scalar object files.

When linking, you then need only specify the library containing the scalar object files.

FILES

file.**cs**	input C* code
file.**o**, *file*.**pe.o**	relocatable object files
file.**s**, *file*.**pe.s**	assembler source files
file.**a**, *file*.**pe.a**	object libraries
a.out	linked executable output
include_dir	directory of C* include files
bin_dir/**cstar-compiler**	C* compiler executable
/bin/cc	C compiler
lib_dir/**libcs_cm5_sparc_sp.a**,	
lib_dir/**libcs_cmf_sparc_pn.a**,	
lib_dir/**libcs_cm5_vu_sp.a**,	
lib_dir/**libcs_cm5_vu_pn.a**,	
lib_dir/**libcs_cm5_cmsim.a**	C* libraries linked by default

where *include_dir*, *bin_dir*, and *lib_dir* are directories listed by the **–dirs** option.

SEE ALSO

cc(1), cmld(1), prism(1)

C Programming Guide*, *C* Release Notes*, *CM-5 C* User's Guide*, *CM-5 C* Performance Guide*, and *Getting Started in C**.

RESTRICTIONS

Bugs and restrictions are listed in the C* bug-update file, by default in **/usr/doc/cstar-*release*.bugupdate**.

Error messages are reported in a non-standard order. Rather than the messages appearing in the order in which the error was encountered, they are grouped together by file, and appear in the order in which the file was first seen. Thus, errors for included files can show up after errors in the source file.

cscomm.h

C* communication functions

SYNTAX

#include <cscomm.h>

SYNOPSIS

```
overload get, send, scan, global;

overload spread, copy_spread, multispread, copy_multispread,
    reduce, copy_reduce;

overload rank, read_from_position, write_to_position,
    make_multi_coord, make_send_address;

overload from_grid, from_grid_dim, to_grid, to_grid_dim;

overload from_torus, from_torus_dim, to_torus, to_torus_dim;

overload read_from_pvar, write_to_pvar;

type:current get(CMC_sendaddr_t:current send_address,
     type :void *sourcep, CMC_collision_mode_t collision_mode);

void get(void:current *destp, CMC_sendaddr_t:current
    *send_addressp, void:void *sourcep, CMC_collision_mode_t
     collision_mode,int length);

type:current send(type:void *destp, CMC_sendaddr_t:current
    send_address, type:current source, CMC_combiner_t combiner,
    bool:void *notifyp);

void:current *send (void:void *destp, CMC_sendaddr_t:current
    *send_addressp, void:current *sourcep, int length,
    bool:void *notifyp);

type:current scan(type:current source, int axis, CMC_combiner_t
    combiner, CMC_communication_direction_t direction,
    CMC_segment_mode_t smode, bool:current *sbitp,
    CMC_scan_inclusion_t inclusion);
```

```
type global(type:current source, CMC_combiner_t combiner);

type:current spread(type:current source, int axis, CMC_combiner_t
    combiner);

type:current copy_spread(type:current *sourcep, int axis,
    int coordinate);

type:current multispread(type:current source, unsigned int
    axis_mask,CMC_combiner_t combiner);

type:current copy_multispread(type:current *sourcep,
    unsigned int axis_mask,CMC_multicoord_t multi_coord);

void reduce(type:current *destp, type:current source, int axis,
    CMC_combiner_t combiner, int to_coord);

void  copy_reduce(type:current *destp, type:current source,
    int axis,int to_coord, int from_coord);

unsigned int:current rank(type:current source, int axis,
    CMC_communication_direction_t direction,
    CMC_segment_mode_t smode, bool:current *sbitp);

type read_from_position(CMC_sendaddr_t send_address,
    type :void *sourcep);

type write_to_position(CMC_sendaddr_t send_address,
    type:void *destp, bool source);

CMC_multicoord_t make_multi_coord(shape s, unsigned int
     axis_mask, CMC_sendaddr_t send_address);

CMC_multicoord_t make_multi_coord(shape s, unsigned int
    axis_mask,int axes[ ]);

CMC_multicoord_t make_multi_coord(shape s, unsigned int
    axis_mask,int axis, ...);

CMC_sendaddr_t:current make_send_address(shape s,
    int:current axis, ...);

CMC_sendaddr_t:current make_send_address(shape s,
    int:current axes[ ]);

CMC_sendaddr_t make_send_address(shape s, int axis, ...);

CMC_sendaddr_t make_send_address(shape s, int axes[ ]);
```

```
type:current from_grid(type:current *sourcep, type:current value,
    int distance, ...);

void from_grid(void:current *destp, void:current *sourcep,
    void:current *valuep, int length, int distance, ...);

type:current from_grid_dim(type:current *sourcep, type:current value,
    int axis, int distance);

void from_grid_dim(void:current *destp, void:current *sourcep,
    void:current *valuep, int length, int axis, int distance);

type to_grid(type:current *destp, type:current source, type:current
    *valuep, int distance, ...);

void to_grid(void:current *destp, void:current *sourcep,
    void:current *valuep, int length, int distance, ...);

void to_grid_dim(type:current *destp, type:current source,
    type:current *valuep, int axis, int distance);

void to_grid_dim(void:current *destp, void:current *sourcep,
    void:current *valuep, int length, int axis, int distance);

type:current from_torus(type:current *sourcep, int distance, ...);

void from_torus(void:current *destp, void:current *sourcep,
    int length, int distance,  ...);

type:current from_torus_dim(type:current *sourcep, int axis,
    int distance);

void from_torus_dim(void:current *destp, void:current *sourcep,
    int length, int axis, int distance);

void to_torus(type:current *destp, type:current source,
    int distance, ...);

void to_torus(void:current *destp, void:current *sourcep,
    int length,int distance, ...);

void to_torus_dim(type:current *destp, type:current source,
    int axis, int distance);

void to_torus_dim(void:current *destp, void:current *sourcep,
    int length, int axis, int distance);

void read_from_pvar(type *destp, type:current source);
```

```
type:current write_to_pvar(type *sourcep);

unsigned int:current enumerate(int axis,
    CMC_communication_direction_t direction,
    CMC_scan_inclusion_t inclusion, CMC_segment_mode_t smode,
    bool:current *sbitp);
```

DESCRIPTION

The C* communication functions, which duplicate and supplement communication features of the language, support grid communication, communication with computation, and general communication. Communication functions are overloaded to support arithmetic, aggregate, and void types.

In the function prototypes listed above, there exists a function definition for the following values of *type*: **bool, signed char, signed short int, unsigned short int, signed int, unsigned int, signed long int, unsigned long int, float, double, long double,** and **void.**

SEE ALSO

cs, *CM-5 C* Users' Guide, C* Programming Guide*

---

math.h

C* mathematical library

SYNTAX

#include <math.h>

SYNOPSIS

```
overload acos, asin, atan;

overload atan2;

overload cos, sin, tan;

overload cosh, sinh, tanh;

overload asinh, acosh, atanh;

overload exp, log, log10, logb;

overload pow, ceil, sqrt, fabs, floor;

overload copysign, drem, finite, scalb; float:current
     acos(float:current);

double:current acos(double:current);

float:current asin(float:current);

double:current asin(double:current);

float:current atan(float:current);

double:current atan(double:current);

float:current atan2(float:current f, float:current f2);

double:current atan2(double:current d, double:current d2);

float:current cos(float:current);

double:current cos(double:current);
```

```
float:current sin(float:current);

double:current sin(double:current);

float:current tan(float:current);

double:current tan(double:current);

float:current cosh(float:current);

double:current cosh(double:current);

float:current sinh(float:current);

double:current sinh(double:current);

float:current tanh(float:current);

double:current tanh(double:current);

float:current acosh(float:current);

double:current acosh(double:current);

float:current asinh(float:current);

double:current asinh(double:current);

float:current atanh(float:current);

double:current atanh(double:current);

float:current exp(float:current);

double:current exp(double:current);

float:current log(float:current);

double:current log(double:current);

float:current log10(float:current);

double:current log10(double:current);

float:current logb(float:current f);

double:current logb(double:current d);

float:current pow(float:current,float:current);

double:current pow(double:current,double:current);
```

```
float:current ceil(float:current);

double:current ceil(double:current);

float:current sqrt(float:current f);

double:current sqrt(double:current d);

float:current fabs(float:current);

double:current fabs(double:current);

float:current floor(float:current);

double:current floor(double:current);

float:current copysign(float:current f, float:current f2);

double:current copysign(double:current d, double:current d2);

float:current drem(float:current f, float:current f2);

double:current drem(double:current d, double:current d2);

int:current finite(float:current f);

int:current finite(double:current d);

float:current scalb(float:current f, int:current i);

double:current scalb(double:current d, int:current i);
```

DESCRIPTION

The mathematical library under C* contains the entire serial C mathematical library, along with parallel overloadings of many of the functions. In addition, only parallel versions of the following functions, which have *no* scalar overloadings, are provided: **acosh, asinh,** and **atanh.**

SEE ALSO

cs, *CM-5 C* User's Guide*, *C* Programming Guide*

RESTRICTIONS

Because the scalar and parallel versions of some routines are implemented using different algorithms, results of routines given the same numerical input may be slightly different in a serial context than in a parallel context.

stdarg.h

C* variable arguments

SYNTAX

#include <stdarg.h>

SYNOPSIS

```
void va_start(va_list ap, parmN) ;

type = va_arg(va_alist ap, type) ;

void va_end(va_list ap) ;
```

DESCRIPTION

The macros **va_start, va_arg,** and **va_end** can be used to write functions that can operate a variable number of arguments.

The **va_start** macro must be called to initialize **ap** before use by **va_arg** and **va_end.**

The **va_arg** macro expands to an expression that has the type and value of the next argument in the call. The value of **ap** is modified so that successive calls to **va_arg** will continue to read arguments in the call.

The **va_end** macro facilitates a normal return from a function that calls the macros **va_start** and **va_arg** to read a variable argument list.

EXAMPLE

```
#include <stdarg.h>
#define MAXARGS    32

void f(int n_params, ...)
{
   int i, array[32] ;
```

```
   va_list ap ;
   va_start(ap, n_params) ;
   for ( i = 0 ; i < MAXARGS; i++)
      array[i] = va_arg(ap, int) ;
   va_end(ap) ;
}
```

SEE ALSO

ANSI C Programming Language Standard, C Programming Guide*

stdlib.h

C* generic utilities

SYNTAX

#include <stdlib.h>

SYNOPSIS

```
int abs(int i);
int rand(void);
void srand(unsigned seed);
overload abs;
int:current abs(int:current i);
void psrand(unsigned seed);
int:current prand(void);
void deallocate_shape(shape *s);
void:void *palloc(shape s, int bsize);
void pfree(void:void *pvar);
```

DESCRIPTION

The C* generic utilities contain the parallel and scalar overloading of **abs.** The serial function is documented on the *abs* man page; the parallel function behaves exactly like the scalar function. Which **abs** function is called depends on whether a scalar or parallel integer is passed as the argument.

The function **psrand** reseeds the random number generator in all processors, even those that are not selected when the call occurs. Even though a scalar integer is passed to **psrand,** every processor will be seeded for a different sequence of random numbers. (Actually, it may be possible for two processors to have the same sequence, given a Connection Machine configuration with many virtual processors.)

The function **prand** is the parallel version of the **rand** function.

SEE ALSO

cs, abs(3), rand(3)

C Programming Guide*

LIMITATIONS

Seed values of **0** and **–1** are not accepted by **psrand.**

---

string.h

C* string handling functions

SYNTAX

#include <string.h>

SYNOPSIS

```
bool:current *boolcpy (bool:current *s1,
    bool:current *s2, size_t n);

bool:current *boolmove (bool:current *s1,bool:current *s2,
    size_t n);

int:current boolcmp (const bool:current *s1, bool:current *s,
    size_t n);

bool:current *boolset (bool:current *s, bool:current c,
    size_t n);

void:current *memcpy (void:current *s1, void:current *s2,
    size_t n);

void:current *memmove (void:current *s1, void:current *s2,
    size_t n);

int:current memcmp (const void:current *s1, void:current *s,
    size_t n);

void:current *memset (void:current *s, int:current c, size_t n);
```

DESCRIPTION

The string handling functions under C* contain the serial C string handling functions along with parallel overloadings of the functions.

SEE ALSO

ANSI C Programming Language Standard

Index

D

E

F

H

I

K

L

M

O

P