The
Connection Machine
System

C* User's Guide

Version 6.0
November 1990

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1990

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

Contents

About This Manual

Objectives of This Manual

This manual describes how to develop, compile, execute, and debug C* programs on a 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 Paris functions, 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 on a CM-2**
Chapter 4 describes how to run a C* program.

Chapter 5 **Debugging a C* Program**
This chapter describes functions useful in debugging a C* program, and contains a sample debugging session.

Appendix A **Man Pages**
This appendix contains man pages for the **cs** compiler command and C* libraries.

Associated Documents

The following document about C* appears in the same volume as this user's guide:

- *C* Programming Guide*

In addition, a technical report is available that provides a reference description of the C* language.

Information about related aspects of the Connection Machine system is contained in the following volumes of the Connection Machine documentation set:

- *Connection Machine Front-End Systems*
- *Connection Machine I/O Programming*
- *Connection Machine Graphics Programming*
- *Connection Machine Parallel Instruction Set*
- *Connection Machine Programming in C/Paris*

Also consult the documentation for your front end (a Sun-4 running SunOS or a VAX running ULTRIX) for further information about program development facilities.

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 CM System Software commands, command options, and file names.
italics	Parameter names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
`% ` **`boldface`** `regular`	In interactive examples, user input is shown in **`boldface typewriter`** 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 back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

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

For Symbolics Users Only

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

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

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 and executing a C* program. 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. You must 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). You can use standard UNIX tools like **gprof** and **make**. Chapter 2 describes facilities for developing a C* program. It also describes how to call Paris functions, functions in the CM graphics, I/O, and scientific software libraries, and CM Fortran subroutines.

1.2 Compiling a C* Program

Compile the files by issuing the command **cs** from your UNIX front end. 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

To execute a program, issue the command **cmattach** along with the name of the executable load module, plus any arguments that the program requires. Or, use the **qsub** command to submit the executable load module as a batch request to the CM batch system.

The **cmattach** and **qsub** commands are discussed in Chapter 4.

1.4 Debugging a C* Program

You can debug a C* program using a standard UNIX debugger like **dbx**. C* provides functions you can call to do such things as print out values of a parallel variable, set the context of a shape, and define a region within a shape on which operations are to take place. Chapter 5 describes the C* debugging functions.

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 Paris, the CM parallel instruction set; see Section 2.3.
- Calls to CM library functions; see Section 2.4.
- Calls to CM Fortran subroutines; see Section 2.5.

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. 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 in the next chapter.

2.1.1 C* Keywords

C* adds the following new keywords to standard C:

```
allocate_detailed_shape
allocate_shape
bool
boolsizeof
```

```
current
dimof
everywhere
overload
pcoord
physical
physical_index
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 this implementation of the language. Do not create identifiers beginning with **CM** in your programs.

2.1.3 Default Shape

Although the language does not define a default shape, this implementation provides one. If you specify a default geometry via the **-g** option to the **cmattach** or **cmcoldboot** command, a VP with this geometry becomes the default shape of the program. If you do not use this option, the default shape is **physical**—that is, a 1-dimensional shape whose size is the number of physical processors to which you are attached.

For more information on **cmattach** and **cmcoldboot**, see the *CM User's Guide*.

To gain access to a default shape specified via the **-g** option, you must first declare a shape that is not fully specified, and assign the current shape to it while the default shape is current. In the following example, the default shape is assigned to shape **S**:

```
Shape S;

main()
```

```
{
   S = current;
   {
      float:S x, y, z;
      with (S)
         x = y + z;
   }
}
```

NOTE: We do not recommend writing code that relies on the default shape, since this feature is implementation-dependent and may change.

2.2 Header Files

C* substitutes its own header files for **<math.h>**, **<stdlib.h>**, and **<string.h>**, as described below. These files are typically in **/usr/include/cs**. C* accepts any other standard C libraries and associated header files. Appendix A contains the man pages for these files.

In addition, C* includes the following header files:

- **<cscomm.h>**, which is the header file for the communication functions described in Part III of the C* Programming Guide.
- **<cstimer.h>**, which provides wrappers for the Paris timing functions described in the *CM User's Guide*. Note that the timer functions in **<cstimer.h>** begin with "**CMC**" rather than "**CM**". If you use these versions of the timing functions and include **<cstimer.h>**, you need not include the much larger header file **<cm/paris.h>**, as described in the *CM User's Guide*.

Man pages for these header files are also contained in Appendix A.

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 different random number for every element of a parallel variable.

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**.

The boolean versions are useful for performing memory operations at the bit level. These functions 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 the following code fragment, both **memcpy** and **boolcpy** copy **source** to **dest**:

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

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

In this case, both functions accomplish the same thing; you would use **boolcpy**, however, if the number of bits to be copied was not a multiple of **char**s.

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 a variable name. Including this file would result in a syntax error. In such a situation, you can do the following:

```
#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>** variable in your program, you must refer to it as **Current**.

2.3 Calling Paris from C*

The header file **<cm/paris.h>** declares the functions in the C* interface to Paris. You might want to use these C functions to obtain Paris features not available with C* syntax. In addition, you need to know the relationship between C* and Paris if you are going to call routines in the CM graphics, I/O, or scientific software libraries. These libraries provide a C/Paris interface; they do not currently provide a direct C* interface.

For general information about Paris functions, see the manual *Introduction to Programming in C/Paris*. For complete reference information on Paris, see the *Paris Reference Manual*. Section 2.4, below, discusses the CM libraries.

2.3.1 The Relationship between Paris and C*

This section explains how concepts in C* map onto underlying Paris concepts.

A *shape* in C* is the VP set ID (**CM_vp_set_id_t**) of a Paris VP set with its associated geometry. When a shape is passed to a function, C* passes this VP set ID. It is also this VP set ID that is returned when you issue a **dbx print** call with the name of the shape as an argument; see Chapter 5. The current shape is the current VP set.

A *parallel variable* in C* is a Paris field, with the length specified by the parallel variable's type. A pointer to a parallel variable is also a Paris field. Thus:

```
pvar == CM_field_id_t
&pvar == CM_field_id_t
```

When you issue the **dbx** command **print** with the name of the parallel variable as an argument, it returns the field ID of a Paris field; see Chapter 5.

When a pointer to a parallel variable is passed to a function, C* passes the field ID of the parallel variable. When a parallel variable is passed to a function, C* passes the field ID of a *copy* of the parallel variable—this allows the parallel variable to be passed by value.

A *position* in C* is a virtual processor. Active positions are processors in the current VP set that have their context flags set.

An *element* in C* is the value of a field stored in an individual virtual processor.

Send and get operations in C* are performed by the corresponding Paris calls. Grid communication functions in C* are performed by the corresponding NEWS calls in Paris.

2.3.2 Calling Paris Functions

In calling Paris functions within a C* program, there is one essential difference from C/Paris programming: C* does all space allocation. You pass C/Paris functions addresses of C* parallel variables rather than direct machine offsets. Little of the usual C/Paris bookkeeping is required—in particular, no initialization or CM stack control is necessary.

Please note the following in calling Paris from C*:

- If you explicitly allocate fields using Paris functions in a C* program, it is important to understand that C* automatically deallocates all stack fields when leaving a block. Therefore, stack fields can exist only within a block. If you want a field to continue to exist after your program leaves a block, allocate a heap field instead.
- Don't manipulate the context using Paris functions; the C* behavior is undefined.

C* Parallel Variables and Paris Field IDs

Pass the addresses of parallel variables rather than field IDs to Paris functions that take field IDs as arguments. (You can simply pass the parallel variable itself, but this requires an unnecessary creation of a temporary variable.)

For example, the following C/Paris code fragment adds the values in two fields:

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

/* C/Paris code omitted. . . . */

CM_set_vp_set(S);
CM_field_id_t p1, p2;

/* Stack fields are explicitly allocated: */

p1 = CM_allocate_stack_field(32);
p2 = CM_allocate_stack_field(32);

/* C/Paris code omitted. . . . */

 CM_s_add_2_1L(p1, p2, 32);
```

Here is the corresponding C*/Paris program fragment, including a call to **`CM_s_add_2_1L`**:

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

/* C* code omitted. . . . */

with (S) {
   int:S p1, p2;

   /* Addresses of parallel variables are passed: */

   CM_s_add_2_1L(&p1, &p2, boolsizeof(int:S));
}
```

Note the use of **`boolsizeof`** to determine the size of a parallel **`int`** in the C* fragment; this makes the code more portable than explicitly specifying the value 32.

C* Shapes and Paris VP Sets

A C* shape represents the VP set ID of the corresponding VP set. For example,

```
CM_vp_set_geometry(S);
```

returns the geometry associated with shape **S**. You could use this geometry in a call to **CM_geometry_axis_vp_ratio**, for example, to obtain the VP ratio of axis 0 of shape **S**:

```
CM_geometry_axis_vp_ratio(CM_vp_set_geometry(S));
```

C* Parallel Arrays and Paris Fields

A parallel array in C* is allocated as one large Paris field, and elements within the array are calculated as offsets within this field, using **CM_add_offset_to_field_id**. If you write a C/Paris function that takes a C* parallel array as an argument, you must treat it similarly. For example, here is a C* parallel array:

```
int:S parray[10];
```

A user-written C/Paris function that takes this array as an argument should be prototyped as in the following example:

```
f(CM_field_id_t x);
```

rather than as:

```
f(CM_field_id_t x[10]); /* This is wrong */
```

To access element 3 of the array, the function should do the following:

```
CM_add_offset_to_field_id(x, 3*sizeof(parray[0]));
```

rather than simply specifying **x[3]**.

C* Memory Layout and Paris Functions

Some advanced Paris functions (for example, **aref32** and **aset32**) require that memory for fields be allocated contiguously on the CM. The current implementation of C* does not guarantee that memory for parallel variables will be laid out contiguously. To ensure that you have the correct memory layout, use **palloc** to allocate the memory yourself for parallel variables that are to be used with these Paris functions.

2.4 Calling CM Libraries

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

- Call routines from the CM graphics libraries to perform basic graphics operations and to display images on the CM graphic display system or on an X Window System.
- Call routines from the CM I/O library to perform standard I/O functions—for example, reading data into the CM from a DataVault or other I/O device.
- 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.

2.4.1 General Information

CM libraries currently provide a C/Paris interface; you can call functions in these libraries as you would any Paris functions. See Section 2.3 for a discussion of the relationship between C* and Paris. Therefore, note the following when consulting the documentation for the libraries:

- When a call requires a field ID as an argument, pass it the address of the corresponding parallel variable.
- As described in Section 2.3, a shape in C* is the VP set ID of a VP set with its associated geometry; positions are virtual processors. Interpret discussions of geometry and VP sets with these correspondences in mind.

- Since C* initializes the CM automatically, ignore the requirement that the Paris routine **CM_init** must be called before making any calls to library routines.

2.4.2 The Graphics and Visualization Library

Use the CMSR library to perform graphics operations and to display images of your data. For complete information on the calls in this library, consult the volume *Connection Machine Graphic Display System* in the Connection Machine documentation set.

2.4.3 The CM I/O Library

I/O operations on the CM are carried out by making calls to functions in the CMFS library (CMFS stands for *CM file system.*) For information on CMFS library calls, consult the *CM I/O System Programming Guide.* This guide describes how to use these calls to do such things as create files in the CM file system, read and write data between a CM and an I/O device in the Connection Machine system, and, if necessary, transpose serial data so that it is in the proper format for parallel operations.

Please note the following, in addition to the information discussed in Section 2.4.1 above:

- Shapes by default are in news order. (You can specify a different order by using the **allocate_detailed_shape** function.) When using the **CMFS_transpose_always** function to transpose data between serial and parallel format, you must include as the final argument one of the provided functions (**CMFS_write_to_row_major**, **CMFS_write_to_column_major**, **CMFS_read_from_row_major**, or **CMFS_read_from_column_major**) to make sure that the data is correctly transposed. Do this even if the data is one-dimensional. The choice between row-major and column-major order depends on the way in which the serial data is laid out; note that data in C is laid out in row-major order. If the data is laid out in an order other than column-major or row-major, you must write your own function to correctly rearrange the data.

2.4.4 The CM Scientific Software Library

A C* program can contain calls to routines in the Connection Machine Scientific Software Library (CMSSL). Currently, C* users must use the C/Paris interface to the CMSSL. For complete information on this interface, see the manual *CMSSL for Paris*.

Please note the following, in addition to the information discussed in Section 2.4.1:

- CMSSL routines typically run more efficiently when fields are allocated contiguously. Normal stack allocation in C* does not work in this way. To get this behavior, you must call the **`palloc`** function to allocate memory individually for each argument to a CMSSL routine.

2.5 Calling CM Fortran

You can call CM Fortran subroutines from within a C* program. This section describes how. See Chapter 3 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>`**.
- 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 the following 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.

VAX users only:

In addition, if you compile on a VAX, you must create and compile a JBL (Jacket Building Language) file to map the subroutine name used in your C* program (*plus a preceding underscore*) to the subroutine name used by the CM Fortran compiler. The CM Fortran compiler converts all subroutine names to uppercase, with no leading or trailing under-

scores. The first line of the file creates the mapping. The second line is a mask of registers to be saved; you need not change the sample line shown below. For example, if the name of the CM Fortran subroutine is `TEST`, create the following file (it must have the suffix **`.jbl`**):

```
_test_:TEST
{pass <r2,r3,r4,r5,r6,r7,r8,r9,r10,r11>};
```

Note that you would use the same file if the CM Fortran routine were called "`test`" (in lowercase), because the CM Fortran compiler converts the name to uppercase.

You can create multiple mappings in a single JBL file.

See Chapter 3 of this manual for information on compiling this file. See the *VAX Fortran for ULTRIX System User Manual* for more information about JBL.

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 the following standard types:

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

In addition, **`<csfort.h>`** provides **`typedef`**s for two new types: **`CMC_complex`** and **`CMC_double_complex`**. The **`typedef`**s 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 (field ID) 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.

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

The following 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
```

2.6 Using UNIX Utilities

You can use standard UNIX utilities like **make**, **prof**, **gprof**, and **dbx** with C* programs. **dbx** is discussed in Chapter 5.

2.6.1 The Program Maintenance Utility make

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

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

2.6.2 The Profiling Tools prof and gprof

The two profiling tools **prof** and **gprof** work with C* programs just as they do with C programs. We recommend using **gprof** rather than **prof**, since **prof** shows only the execution times of individual Paris operations, without adding them into the times shown for the C* routines that called them. In contrast, **gprof** shows the total time spent in each C* routine. See the *CM User's Guide* for more information about profiling.

Chapter 3

Compiling a C* Program

To compile a C* program, use the **cs** command.

Section 3.1 describes the basics of using **cs**, including the most common options. This section contains all the information most programmers need.

Section 3.2 provides a more detailed look at the compilation process and discusses options more likely to be used by advanced programmers.

Section 3.3 discusses symbols for which **cs** provides **#defines**.

Section 3.4 explains how to compile a C* program that calls a CM Fortran subroutine.

3.1 The Basic Compilation Process

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, **.obj** JBL files (VAX only), and **.a** library files, but all parallel code must be in a **.cs** file.

3.1.1 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 1. C* compiler options

Option	Meaning
Basic options:	
-help	
-h	Give information about **cs** without compiling.
-O[0]	Invoke extra C* optimization. Specify **-O0** to turn off all optimization.
-release *number*	Compile using the compiler version *number*.
-ucode *number*	Link with CM software version *number*.
-version	Print the compiler version number.
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; required for C* debugging functions.
-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*.
-pg	Link with profiling libraries for use with **gprof**.
-U*name*	Undefine the C preprocessor symbol *name*.
Advanced options:	
-cc *compiler*	Use the specified C compiler.
-dryrun	Show, but do not execute, compilation steps.
-force	Force **.c** files through the C* compilation phase.
-keep c	Keep the intermediate **..c** file.
-noline	Suppress **#line** directives in the output C file.
-overload	Display names used by the compiler for overloaded functions.
-verbose	
-v	Display informational messages during compilation.
-warn	Suppress warnings from the C* compilation phase.
-Z*comp switch*	Pass option *switch* to component *comp*, where *comp* is **cpp** or **cc**.

Getting Help: The –help Option

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

Changing the Optimization Level: The –O Option

Use the **-o** option to choose extra optimization for your program, or to turn off optimization.

If you do not specify **-o**, default optimization includes local copy propagation, dead code elimination, variable minimization, and some peephole optimizations. You can debug a program compiled at this level by using **dbx** or another debugger.

For more optimization, specify **-o**. At this level, the compiler optimizes user variables as well as compiler-generated temporaries. A program compiled using **-o** is too highly optimized for use with a debugger.

To turn off all optimization, specify **-o0**. This is normally not useful.

Choosing a Specific Version of the Compiler: The –release Option

Use the **-release** option to choose a specific version of the compiler. The initial release of the compiler has two versions: 6.0, which operates with version 5.2 of the CM system software, and 6.0.1, which operates with version 6.0 of the CM System Software. (Note that the compiler version and the CM system software version do not coincide.) Either one of these versions may be installed as the default. To determine which version is the default, use the **-version** option, as described below. To compile using a version that is not the default, use the **-release** option. For example,

```
% cs -release 6.0.1 myfile.cs
```

compiles with C* Version 6.0.1.

Typically, the default C* version will work with the standard installed version of the Paris library. If that is the case, you will have to specify the **-ucode** option along with the **-release** option in order to compile a program under a CM system software version that is not the default; the **-ucode** option is described below.

Choosing a Specific Version of Paris: The –ucode Option

By default, **cs** uses the standard installed version of the Paris library. You might want to use a different version for your program if, for example, you want to compare execution time for the standard version against that obtained using an older version. Use the **-ucode** option, followed by a four-digit number, to specify another version of Paris. Obtain this four-digit number from your system administrator. Then issue the **cs** command as in the following example:

```
% cs -ucode 5211 myfile.cs
```

The Paris library 5211 is the version used in Version 5.2 of CM system software. If Version 6.0.1 is the default version of C*, you would also have to specify the **-release** option to compile your program using this library. For example:

```
% cs -ucode 5211 -release 6.0 myfile.cs
```

Note that the version numbers of C* and CM System Software do not coincide. See above for more information about the **-release** option.

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.

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

The C* compiler allows you to specify the **cc** options **-c**, **-D**, **-g**, **-I**, **-l**, **-L**, **-o**, and **-pg** on the **cs** command line. See Table 1 for a brief description of these options; for more information, consult your documentation for **cc**.

You must include the **-g** option if you want to use the C* debugging functions on the compiled program.

Use the **-pg** option if you want to profile your program using the UNIX **gprof** utility. This option causes the compiler to link your program with special Paris and I/O libraries. These

libraries provide more accurate timings of individual operations, at the expense of decreased efficiency in the program as a whole.

3.2 A Closer Look at the Compilation Process

The **cs** command actually has three phases, which normally are transparent to the user:

- A preprocessing phase
- A C* compilation phase
- A C compilation phase (which includes linking)

The C* compilation phase takes a preprocessed C* source file and generates an equivalent output file, which has the extension **..c**. Any parallel code in the **.cs** file is translated into C/Paris in this **..c** file; serial code in the **.cs** file may be somewhat changed. The C compilation phase takes this C/Paris file and produces an executable load module.

Typically, only **.cs** files are read during the C* compilation phase; **.c**, **.o**, and **.a** files are not read until the C compilation phase. This means that putting serial code in **.c** files speeds up compilation, since these files don't have to go through both compilation phases. You may, however, *want* a **.c** file to go through the C* compilation phase if the code includes ANSI features like function prototyping, since the C* compilation phase implements many ANSI features. To accomplish this, specify the **-force** option to **cs**, as described below.

3.2.1 Advanced Options

This section describes **cs** options that would typically be used by an advanced programmer. All options are listed in Table 1.

Using Another C Compiler: The –cc Option

The C* compiler works in conjunction with the standard C compiler available on your VAX or Sun front end. *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 the following example:

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

Displaying Compilation Steps: The –dryrun Option

Specify **-dryrun** to cause **cs** to show, but not carry out, the steps in the compilation.

Putting .c Files through the C* Compilation Phase: The –force Option

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

Keeping the Intermediate File: The –keep Option

Specify the **-keep** option, followed by the argument **c**, to have **cs** keep the **..c** file produced by the C* compilation phase.

NOTE: The **..c** file is not portable between front ends. You cannot create a **..c** file on a Sun, for example, and execute it on a VAX.

Suppressing Line Directives: The –noline Option

Specify **-noline** to suppress **#line** directives in the output C file. This seriously limits your ability to debug C* source files with **dbx** or other debuggers. It does, however, let you debug at the C/Paris level—in other words, you can debug the **..c** file rather than the **.cs** file.

Displaying Names of Overloaded Functions: The –overload Function

Use the **-overload** option to cause the compiler to display informational messages listing the names it uses internally for overloaded functions. This is necessary if you want to invoke such a function directly using **dbx**.

Turning On Verbose Compilation: The –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.

Turning Off Warnings: The –warn Option

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

Specifying Options for Other Components: The –Z Option

Use the **-Z** option to specify **cpp** or **cc** options that **cs** does not recognize. 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.cs
```

to suppress **cc** warning messages.

3.3 Symbols

The **cs** command provides **#defines** for the symbols listed below:

cstar	The C* language (as opposed to the C language)
unix	Any UNIX system
ultrix	ULTRIX only
vax	VAX only
sun	Sun only
sparc	Sun-4 only

If the symbol is applicable, **cs** automatically defines it as **1**. For example, if you are executing **cs** on a VAX running ULTRIX, the **vax** and **ultrix** symbols are each defined as 1. This lets you place the symbols in **#ifdef** statements to isolate code for execution only when

the program is running under VAX ULTRIX. Thus, you can use these symbols to ensure that your code is portable. The **cstar** symbol lets you isolate parallel code within a program, allowing you to share source code between a C and a C* program.

3.4 Compiling 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 -c testcs.cs
   ```

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

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

3. **For Vax users only**: Compile the JBL file you created to map the subroutine name(s) you used in the C* program to those created by the ULTRIX Fortran compiler. For example:

   ```
   % jbl cstofortran.jbl
   ```

   This creates a file with a **.obj** suffix that you must link in with your other files.

4. Use the **cmf** compiler to link the **.o** files, the .**obj** file (if you are compiling on a VAX), and the required libraries. You must also link the following libraries:

   - **libcsrt.a** — the C* runtime library
   - **libcmcl2.a** — a shared library that is required when calling CM Fortran and C*

   In addition, include any libraries that you need to explicitly link with your program.

   For example, on a Sun you might issue the **cmf** command as follows:

   ```
   % cmf -nocmfmain testcs.o testfcm.o -Zld "-lcsrt -lcmcl2"
   ```

On a VAX you might issue the **cmf** command as follows:

```
% cmf -nocmfmain testcs.o testfcm.o cstofortran.obj \
  -Zlk "-lcsrt -lcmcl2"
```

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

Chapter 4

Executing a C* Program on a CM

Once a C* program has been compiled and linked, you can execute the output file on a Connection Machine system. This chapter gives an overview of how to execute a program on a CM system. For complete information, consult the *CM User's Guide*.

4.1 Overview

To execute a program on a CM, you must gain access to some of its processors; this is known as *attaching* to the CM. Your front end has one or more interfaces, each of which can attach to one or more sequencers within the CM; each sequencer controls a group of CM processors.

There are two basic methods of attaching to a CM: *direct access* and *batch access*.

- For direct access, issue the **`cmattach`** command to attach an interface to one or more sequencers on the CM. Depending on how you issue the command, your programs is executed immediately and you are then detached from the CM, or you enter an interactive subshell from which you can execute the program and other commands.

- For batch access, issue the **`qsub`** command to submit your program to a queue that has been set up in the CM batch system. Your program attaches to the CM and is executed when it reaches the head of the queue. Check with your system administrator to find out if the batch system has been installed at your site.

In both cases, access to the CM can be either *exclusive* or *timeshared*, depending on how your system is configured. With exclusive access, only one user can be attached to an interface and a sequencer at a time. With timeshared access, multiple users can be attached at

a time, and multiple jobs can be running on the same processors. Neither affects the way you compile or execute your program. Performance may be somewhat slower under time-sharing, however.

Direct, exclusive access is ideal when you are developing your program, since it lets you debug interactively on the CM. This kind of access may be relatively difficult to obtain, however.

4.2 Obtaining Direct Access

Use the **cmattach** command to obtain direct access to CM processors.

4.2.1 Executing the Program Immediately

To execute a program immediately, issue **cmattach** with the name of the executable program as an argument. For example,

```
% cmattach myprogram
```

attaches to any free interface and sequencer, initializes (cold boots) the sequencer and its processors, and executes the program. The system displays a message that gives you the following information:

- The name of the CM, the number of the sequencer, and the number of processors to which you are attaching
- The version of the CM microcode that this sequencer is running
- Whether Paris-level safety checking is on

If no sequencers or interfaces are available, you receive an error message.

4.2.2 Obtaining an Interactive Subshell

Issue **cmattach** without the name of an executable program to obtain an interactive subshell. If an interface and a sequencer are available, you are attached to them, and the processors controlled by the sequencer are cold-booted. You are placed in a UNIX subshell, from which you can execute your program, enter the debugger, or issue any standard UNIX commands.

To leave the subshell and detach from the CM, type **exit** or the Ctrl-D key combination at the UNIX prompt.

4.2.3 Options for cmattach

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

Waiting for Resources

Use the **-w** option to tell **cmattach** to wait if initially it cannot obtain an interface and a sequencer. If you don't require interactive use of the CM, it is generally preferable to submit your program to a batch queue.

Specifying a Sequencer, an Interface, and a CM

You can control the CM, the interface, and the sequencer(s) to which **cmattach** will attach you. You might want to specify a particular sequencer if, for example, you want to use a framebuffer that is connected to that sequencer.

Use the **-C** option, followed by the name of a CM, to specify the CM to which you want to attach.

Use the **-S** option to specify number of the sequencer(s) to which you want to attach. Valid values are 0, 1, 2, 3, 0–1, 2–3, and 0–3.

Use the **-i** option to specify the number of the interface to which you want to attach.

Obtaining Exclusive Access

Use the **-e** option to **cmattach** to specify that you require exclusive access to one or more sequencers. If you use this option, the system will not attach you to a sequencer that is running under timesharing.

4.3 Obtaining Batch Access

In the CM batch system, you submit your program as a request to a queue. The queue may be associated with a CM and a sequencer, 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.

The batch system 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 the following command:

```
% qstat -x
```

4.3.1 Submitting a Batch Request

Use the **qsub** command to submit a batch request for execution via a queue in the CM batch system. 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 the following names of executable C* programs:

  ```
  myprogram1
  myprogram2
  myprogram3
  ```

  You can then submit these programs for execution by issuing the following 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—for example, **cmsetsafety** to turn Paris run-time safety on or off.

4.3.2 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.

4.4 Other CM Commands

Other commands are useful in executing programs on the CM. For example:

- Use the **cmfinger** command to obtain information about the CM system: who is using the interfaces and sequencers, and whether any sequencers are free.
- Use the **cmdetach** command to detach an interface from a sequencer, making it available for use by another user.
- Use the **cmps** command to list the processes that are running under timesharing on a sequencer.
- Use the **cmsetsafety** command to turn Paris-level safety checking on and off.
- Use the **cmcoldboot** command to reset the state of the CM hardware to which you are attached.
- 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.

See the *CM User's Guide* and the on-line man pages for complete information on these commands.

Chapter 5

Debugging a C* Program

C* programs can be debugged using a standard debugger like **dbx**. C* provides a variety of functions that you can call within a debugger. Among other things, these functions let you:

- Define a region within a shape to use for debugging.
- Print values of a parallel variable.
- Make positions active or inactive.
- Assign values to a parallel variable.

These functions stay in effect during the execution of a program in **dbx**. They output their results to the standard error device. Table 2 summarizes the debugging functions discussed in this chapter.

As described in Chapter 3, note that debugging is difficult when your program is compiled with the **-O** option.

5.1 Using dbx

To use **dbx** or another debugger, invoke it from within an interactive subshell spawned by **cmattach**. For example:

```
% cmattach
Attaching the Connection Machine system name...
coldbooting... done.
Attached to 8192 physical processors on
sequencer 0, microcode version 6002
```

```
Paris safety is off.

Entering CMATTACH subshell. Type "exit" or control-D
to detach the CM. . .

% dbx a.out
(dbx) [dbx commands such as run, step, and next]
(dbx) quit
% exit
Detaching... done.
%
```

The **dbx** commands behave with C* programs as they do with C programs, with the exception noted below.

Note on the VAX version of dbx

C* programs are translated into C/Paris programs as part of the compilation process. Although **dbx** displays the C* statements, it actually operates on this C/Paris program. In the VAX version of **dbx**, the result is that you may have to issue a **step** or **next** command more than once before a C* statement is completely executed; what is happening is that **dbx** is stepping through the underlying C/Paris statements.

5.1.1 Invoking Overloaded Functions from within dbx

To keep track of different versions of an overloaded function within a program, the compiler assigns separate names to each version. If you want to invoke a particular version of such a function from within **dbx**, you must use this internal name. To obtain these names, use the **-overload** option when compiling the program; this causes the compiler to display the names it uses. For example, if your program contains two versions of the function **sin**, one called with a parallel **float**, the other with a parallel **double**, the compiler might respond as in the following example:

```
% cs -overload myprogram.cs
     "myprogram.cs", line 13: info: overloading is CMC_sin_F_
     "myprogram.cs", line 14: info: overloading is CMC_sin_D_
```

Table 2. C* debugging functions

Function and Arguments	Use
CMC_default_type (*type_sel*)	Change the default data type to *type_sel.*
CMC_define_format (*type_sel, "format"*)	Specify a format for printing.
CMC_define_region (*shape, lower_bound, upper_bound, ...*)	Specify a region of a shape.
CMC_define_type (*pvar, type_spec*)	Specify a data type for a parallel variable.
CMC_define_view (*shape, row, col, ...*)	Specify the order in which axes are to be represented by **CMC_print_region**.
CMC_define_width (*type_sel, width*)	Specify a width for printing a parallel variable.
CMC_help ()	Get information about the debugging functions.
CMC_off (*shape, coord ...*)	Make a position inactive.
CMC_off_region (*shape*)	Make all positions in the shape's region inactive.
CMC_on (*shape, coord, ...*)	Make a position active.
CMC_on_region (*shape*)	Make all positions in the shape's region active.
CMC_pop_context (*shape*)	Change the context to the most recently saved context.
CMC_print (*pvar, coord, ...*)	Print the value of one element of a parallel variable.
CMC_print_region (*pvar*)	Print values of a parallel variable in a region.
CMC_push_context (*shape*)	Save the current context on a stack.
CMC_set (*pvar, value, coord, ...*)	Assign a value to an active element of a parallel variable.
CMC_set_region (*pvar, value*)	Assign a constant value to active elements of a parallel variable in a region.
CMC_status ()	Display debugging status information.

5.1.2 Calling C* Debugging Functions from within dbx

To use one of the C* debugging functions described in this chapter, issue it with the **dbx** command **call**. To lessen the amount of typing this requires, you can use the **dbx** command **alias** to create an abbreviation for the call. For example, issuing the following command:

```
% (dbx) alias ppr "call CMC_print_region"
```

makes **ppr** an alias for the string **"CMC_print_region"**. You can then use **ppr** along with the appropriate argument in place of a call to **CMC_print region**. For example:

```
% (dbx) ppr(p1)
```

To execute these **alias** commands automatically when you start a **dbx** session, put them in the file **.dbxinit** in your current or home directory.

We provide a sample **.dbxinit** file, called **cs-dbxinit**, which is located in the directory that contains sample programs. Ask your system administrator for the path for this directory at your site.

5.2 Defining a Region: The CMC_define_region Function

C* debugging functions let you perform operations either on a single element or position, or on a selected set of positions within a shape; this set of positions is referred to as a *region*. Since shapes usually have thousands of positions, it is easier to focus on a small subset of them for debugging purposes.

Initially a shape's region consists of all positions in the shape. To change the region, use the **CMC_define_region** function. Its first argument is the name of the shape. Then, for each dimension, specify the beginning and ending coordinates that define the region. Start with axis 0, and include beginning and ending coordinates for each axis in the shape. For example,

```
(dbx) call CMC_define_region(ShapeA, 0, 4)
```

defines the region of **ShapeA** as positions 0 through 4; **ShapeA** must be a 1-dimensional shape.

```
(dbx) call CMC_define_region(ShapeB, 0, 4, 0, 9)
```

defines the region of **ShapeB** as those positions with coordinates 0 through 4 along axis 0 and 0 through 9 along axis 1—a total of 50 positions. **ShapeB** must be a 2-dimensional shape.

Note that a region can contain both active and inactive positions.

Once a region is defined for a shape, it stays in effect until you issue another call to **CMC_define_region** to change it, or until the program finishes execution within **dbx**.

NOTE: To indicate the current shape, specify **CM_current_vp_set** rather than the C* keyword **current**.

5.3 Defining Data Types for Parallel Variables: The CMC_default_type and CMC_define_type Functions

C* debugging functions must know the type of a parallel variable before they can operate on it. The default type for use with the debugging functions is **double**. To change the default, call the **CMC_default_type** function. Its sole argument is a data type selector, which must be one of the following:

```
CMC_bool
CMC_char
CMC_double
CMC_float
CMC_int
CMC_long_double
CMC_long_int
CMC_short
```

If you want the default to be an unsigned data type, use the corresponding signed type. For example, use **CMC_int** to make either integers or unsigned integers the default data type.

If a parallel variable does not have the default data type, specify its type using the **CMC_define_type** function. Include the parallel variable and the type selector as arguments. For example,

```
(dbx) call CMC_define_type(p1, CMC_int)
```

defines **p1** as an **int**.

You can't define a type for a parallel variable until you have passed the point in your program where space is allocated for the parallel variable on the CM. For a parallel variable at file scope, this occurs when you enter the first function in the compilation unit where the parallel variable is declared. For an **auto** parallel variable, this occurs when the parallel variable is declared.

Note

It is important to use **CMC_define_type** to define the type for any parallel variable that is not of the default type. If you don't, and the parallel variable is smaller than the default type, the debugger may try to access memory beyond the parallel variable's boundary; this may cause the debugger to abort.

Note that you can't use structure or array syntax to print out parallel structures or parallel arrays.

5.4 Printing Values of a Parallel Variable: The CMC_print and CMC_print_region Functions

C* provides functions that let you print out either the value of a single element of a parallel variable, or the values of a parallel variable's elements in the region specified by **CMC_define_region**. If the parallel variable does not have the default type, you must have previously called the function **CMC_define_type** to specify a type for it.

To print the value of a single element, call the **CMC_print** function. As arguments, specify the parallel variable, followed by the coordinate(s) of the element whose value you want to print. Start with axis 0. For example,

```
(dbx) call CMC_print(p1, 5, 7)
```

prints the value of **[5][7]p1**. If the specified element is in an inactive position, its value is displayed in parentheses.

To print the values of a parallel variable's elements in a region, call **CMC_print_region**. The parallel variable is the only argument. The function prints the values of the elements in the region specified by **CMC_define_region**; if you have not called **CMC_define_region** for the shape of this variable, the function prints the values of all elements of the parallel variable.

If parallel variable **p1** is of shape **ShapeA** and you issue the following commands:

```
(dbx) call CMC_define_region(ShapeA, 0, 4, 0, 9)
(dbx) call CMC_print_region(p1)
```

CMC_print_region prints out the specified values of **p1** as a 5 by 10 table, with axis 0 as the rows and axis 1 as the columns. Values of elements in inactive positions are displayed in parentheses.

If the shape is 3-dimensional, the function prints out a series of tables of values, each representing a plane of positions through axis 2. The function starts with the table of values for elements whose axis 2 coordinate is the lower boundary for axis 2 specified in the call to **CMC_define_region**.

You can change the format in which the values are displayed; see Section 5.4.2.

If the region contains too much data to be displayed on the screen, reissue **CMC_define_region** to reduce the size of the region.

5.4.1 Printing Values of a Pointer to a Parallel Variable

To look at values pointed to by a scalar pointer to a parallel variable, treat the pointer as if it were a regular parallel variable. For example, if you declare **par_ptr** as follows:

```
int:current *par_ptr;
```

you can define its type as follows:

```
(dbx) call CMC_define_type(par_ptr, CMC_int)
```

If the current shape is 1-dimensional, you can print out the value at element [0] as follows:

```
(dbx) call CMC_print(par_ptr, 0)
```

You cannot treat a pointer to a parallel variable as you would a standard C pointer, because the debugger doesn't understand C* syntax. See Chapter 2 for information on the underlying implementation of scalar-to-parallel pointers in C*.

5.4.2 Changing the Format: The CMC_define_format, CMC_define_width, and CMC_define_view Functions

To change the way in which **CMC_print_region** displays the values of a parallel variable, use the **CMC_define_format**, **CMC_define_width**, and **CMC_define_view** functions.

Use **CMC_define_format** to specify a different display format for a data type. The function takes as arguments a type selector and a **printf** format. For example,

```
(dbx) call CMC_define_format(CMC_float, "%5.2f")
```

causes **floats** to be printed out with a field width of 5 and a precision of 2. See Section 5.3 for the list of type selectors.

Due to a restriction in some Sun versions of **dbx**, you cannot pass a string to **dbx** as an argument to a function. This means that you cannot call **CMC_define_format** from a Sun front end. Instead, use **CMC_define_width**. **CMC_define_width** takes as arguments a type selector and an integer that specifies the field width of that type. For example:

```
(dbx) call CMC_define_width(CMC_int, 4)
```

causes integers to be printed with a width of 4.

Note the following in using **CMC_define_width**:

- If the type selector is **CMC_float**, **CMC_double**, or **CMC_long_double**, you must specify two integers; the first specifies the width, and the second specifies the precision. All other type selectors take only one integer.
- If the type selector is **CMC_char**, and you specify a negative width, the **char** is printed as an **int** with the width you specify. If the width is greater than or equal to 0, the **char** is printed as a character.
- If you specify a width of 0, minimum-width columns are printed.

Use **CMC_define_view** to specify the order in which axes are represented for a shape. By default, axis 0 is represented as rows and axis 1 is represented as columns. **CMC_define_view** takes the following arguments:

```
CMC_define_view(shape, row_axis, column_axis, ... )
```

Therefore, to display axis 0 as the columns and axis 1 as the rows for shape **ShapeB**, call the function as follows:

```
(dbx) call CMC_define_view(ShapeB, 1, 0)
```

If **ShapeC** is 3-dimensional, you can reverse the default method of presenting the axes by calling the function as follows:

```
(dbx) call CMC_define_view(ShapeC, 2, 1, 0)
```

The results of the call remain in effect for a shape even if you call **CMC_define_region** to change the region that is to be displayed.

NOTE: To indicate the current shape, specify **CM_current_vp_set** instead of the C* keyword **current**.

5.5 Setting the Context: The CMC_on, CMC_on_region, CMC_off, and CMC_off_region Functions

C* provides functions that let you make positions active or inactive.

For a single position, call the function **CMC_on** to make a position active; call the function **CMC_off** to make it inactive. Both take as arguments a shape and the coordinates of the position, starting with axis 0. For example,

```
(dbx) call CMC_on(ShapeB, 47, 59)
```

makes position [47][59] of **ShapeB** active.

```
(dbx) call CMC_off(ShapeB, 47, 59)
```

makes it inactive.

For a region, call **CMC_on_region** to make all positions within the region active, and call **CMC_off_region** to make them all inactive. The only argument for both functions is a shape. For example,

```
(dbx) call CMC_on_region(ShapeA)
```

makes all positions active in the currently defined region for shape **ShapeA**.

```
(dbx) call CMC_off_region(ShapeA)
```

makes them inactive.

NOTE: To indicate the current shape, specify **CM_current_vp_set** instead of the C* keyword **current**.

5.5.1 Saving and Restoring Contexts: The CMC_push_context and CMC_pop_context Functions

In debugging, you might want to change a shape's context, see what happens, and then change back to an earlier context. To do this, use the functions **CMC_push_context** and **CMC_pop_context**. Both take a shape as an argument. Use **CMC_push_context** to save the current context for a shape. You might then call a function like **CMC_on_region** to change the context. Once you are through using the changed context, call **CMC_pop_context** to return the context to what it was before the change. Do this before leaving a block that changes the context; you will get unexpected results if you issue **CMC_pop_context** when the context is not the same as it was when you issued **CMC_push_context**.

NOTE: To indicate the current shape, specify **CM_current_vp_set** instead of the C* keyword **current**.

5.6 Assigning Values to a Parallel Variable: The CMC_set and CMC_set_region Functions

C* provides two debugging functions that let you change the values in elements of a parallel variable. For both these functions, if the parallel variable's type is not the default, you must first specify its type using **CMC_define_type**.

To assign a value to a single element of a parallel variable, call the **CMC_set** function. As arguments, specify the parallel variable, the value to be assigned, and the coordinates of the element, starting with axis 0. For example,

```
(dbx) call CMC_set(p2, 4.0, 6, 7)
```

assigns the value 4.0 to **[6][7]p2**, provided that the position is active. If the position is inactive, you receive a warning message and the value is not assigned.

To assign a single value to all active parallel variable elements in a region, call **CMC_set_region**. As arguments, specify the parallel variable and the value. For example,

```
(dbx) call CMC_set_region(p2, 4.0)
```

assigns the value 4.0 to all elements of **p2** that are in active positions in the currently defined region. If some positions are inactive, the value is assigned to the elements in active positions, and you receive a warning message about the inactive positions.

5.7 Obtaining Status Information: The CMC_status Function

To find out the current status of your debugging session, call the **CMC_status** function. **CMC_status** prints out information such as the default type, the default formats, the default views, and the current region(s). If you have not defined a region for a shape, but you have called functions like **CMC_print_region** that access a region for the shape, **CMC_status** prints the default, which is all positions in the shape.

Note

In the display, parallel variables are shown as Paris field IDs, and shapes are shown as Paris VP set IDs. To determine the field ID for a parallel variable, use the **dbx** command **print** with the parallel variable name as the argument. Similarly, to obtain the VP set ID for a shape, issue **print** with the name of the shape. See Chapter 2 for a discussion of the relationship between Paris concepts and C* concepts.

5.8 Obtaining Help: The CMC_help Function

Call the function **CMC_help** to obtain a list of the C* debugging functions and their arguments.

5.9 Sample Debugging Sessions

This section presents a series of **dbx** sessions in which we debug a C* program. The sessions are on a Sun; VAX sessions would look somewhat different.

5.9.1 The Program

The program, **primes.cs**, is supposed to find and display prime numbers, and is shown below; line numbers have been added to help you follow the explanations. There are several errors in the program. The rest of the section will show the process by which these errors are uncovered.

```
 1. #define MAXIMUM_PRIME 16384
 2.
 3. #define FALSE 0
 4. #define TRUE 1
 5. #define FIRST_PRIME 2
 6.
 7. /*                                                          */
 8. /*      Function to find prime numbers                      */
 9. /*                                                          */
10. /*  Parameters:                                             */
11. /*                                                          */
12. /*   A pointer, "is_prime_p," to a one-dimensional parallel */
13. /*   char that will have non-zero elements in all positions */
14. /*   where the index is a prime number                      */
15. /*                                                          */
16. /*  Side effects:                                           */
17. /*                                                          */
18. /*   find_primes alters the one-dimensional parallel char   */
19. /*   that is pointed to by "is_prime_p."                    */
20. /*                                                          */
```

```
21. /*  Calling constraints:                                          */
22. /*                                                                */
23. /*   The shape of the parallel char pointed to by                 */
24. /*   "is_prime_p" must be the current shape and all               */
25. /*   positions must be active                                     */
26. /*                                                                */
27. /*  Algorithm:                                                    */
28. /*                                                                */
29. /*   This function will use the Sieve of Eratosthenes to          */
30. /*   find the prime numbers.  That is, it will iterate            */
31. /*   through all numbers that are indices to the one-             */
32. /*   dimensional parallel char pointed to by is_prime_p           */
33. /*                                                                */
34. void find_primes(char:current is_prime_p) {
35.  char:current is_candidate;
36.  int minimum_prime;
37.
38.
39.
40.  is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
41.
42.  do
43.    where(is_candidate) {
44.      minimum_prime = <?= pcoord(0);
45.      where(pcoord(0) % minimum)
46.        is_candidate = FALSE;
47.      [minimum_prime](*is_prime_p) = TRUE;
48.    }
49.  while(|= is_candidate);
50. }
51.
52. main() {
53.  shape [MAXIMUM_PRIME]s;
54.
55.  char:s is_prime;
56.  int i;
57.
58.
59.     find_primes(&is_prime);
60.    for(i=0; i<MAXIMUM_PRIME; i++)
61.    if([i]is_prime)
62.      printf("The next prime number is %d\n", i);
63. }
```

We want to use the following algorithm:

1. Take as candidates all non-negative integers up to a specified value.
2. Eliminate 0 and 1 as candidates.
3. The smallest remaining candidate is a prime; call it *p*.
4. Remove *p* and all multiples of *p* from the set of candidates.
5. Go to step 3.

To implement this algorithm, **primes.cs** uses a function called **find_primes**. **find_primes** declares a parallel variable called **is_candidate** whose coordinates represent possible prime numbers. The function makes positions 0 and 1 inactive (to eliminate 0 and 1 as candidates), then finds the minimum active position (initially, this is 2); this is a prime number. It then turns off all positions whose coordinates are multiples of the coordinate of this minimum active position; these coordinates no longer represent possible primes. The element at the minimum active position is set to 1 in a parallel variable pointed to by **is_prime_p**.

find_primes goes through a **do** loop, repeating these steps until there are no candidates left. The coordinates of the elements set to 1 in the parallel variable pointed to by **is_prime_p** represent the prime numbers.

5.9.2 Compiling the Program

We begin by trying to compile the program:

```
% cs -o primes primes.cs -g
```

But **primes.cs** does not make the compiler happy. The compiler responds:

```
"primes.cs", line 47: illegal indirection
"primes.cs", line 59: type clash on argument 1
"primes.cs", line 45: undefined identifier minimum
```

We look at the offending lines:

```
47.     [minimum_prime](*is_prime_p) = TRUE;
59.     find_primes(&is_prime);
45.     where((pcoord(0) % minimum))
```

The problems are fairly easy to spot. The difficulties with lines 47 and 59 are caused by the same error. We defined **find_primes** on line 34 as taking a parallel variable, rather than as taking a pointer to a parallel variable. Thus, the compiler complained when it reached the lines where we treat the parameter as a pointer.

The problem with the identifier **minimum** on line 45 is also obvious. We declared the variable as **minimum_prime** on line 36, but refer to it as **minimum** here.

5.9.3 The Initial Debugging Session

We make the corrections, which are shown below in bold (we omit the initial comments):

```
 1. #define MAXIMUM_PRIME 16384
 2.
 3. #define FALSE 0
 4. #define TRUE 1
 5. #define FIRST_PRIME 2
34. void find_primes(char:current *is_prime_p) {
35.  char:current is_candidate;
36.  int minimum_prime;
37.
38.
39.
40.  is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
41.
42.  do
43.    where(is_candidate) {
44.      minimum_prime = <?= pcoord(0);
45.      where(pcoord(0) % minimum_prime)
46.        is_candidate = FALSE;
47.      [minimum_prime](*is_prime_p) = TRUE;
48.    }
49.  while(|= is_candidate);
50. }
51.
52. main() {
53.  shape [MAXIMUM_PRIME]s;
```

```
54.
55.  char:s is_prime;
56.  int i;
57.
58.
59.     find_primes(&is_prime);
60.    for(i=0; i<MAXIMUM_PRIME; i++)
61.      if([i]is_prime)
62.        printf("The next prime number is %d\n", i);
63. }
```

We confidently compile the revised program. As expected, there are no errors, so we attach to a CM. In the **cmattach** subshell, we first set Paris safety on (this is recommended when you are going to be debugging):

```
% cmsetsafety on
```

And then we run the program:

```
% primes
```

Unfortunately, something seems to be wrong. The program isn't printing out a list of prime numbers; it isn't doing anything. It seems to be stuck in an infinite loop. We use Ctrl-C to kill the program, and we go into **dbx**:

```
% dbx primes
Reading symbolic information...
Read 1693 symbols
```

We have created aliases for the C* debugging functions and some frequently used **dbx** commands. We print them out to refresh our memory:

```
(dbx) alias
df       "call CMC_define_format"
dr       "call CMC_define_region"
dt       "call CMC_define_type"
dv       "call CMC_define_view"
dw       "call CMC_define_width"
off      "call CMC_off"
offr     "call CMC_off_region"
on       "call CMC_on"
onr      "call CMC_on_region"
```

```
phelp     "call CMC_help()"
popc      "call CMC_pop_context"
pp        "call CMC_print"
ppr       "call CMC_print_region"
pstat     "call CMC_status()"
pushc     "call CMC_push_context"
set       "call CMC_set"
setr      "call CMC_set_region"
type      "call CMC_default_type"
n         "next"
p         "print"
s         "step"
```

We will use these abbreviations in our debugging sessions. Note that you must create your own aliases for the C* debugging functions; put the appropriate **alias** commands in **.dbxinit** if you want them to be in effect when you start your **dbx** session.

We then start debugging **primes.cs**. We decide to take a look at **is_candidate** before the program enters the first **where** statement of the **find_primes** function. We don't need to see very much of the parallel variable; we know that elements [0] and [1] should be set to 0, and elements [2] and above should be set to 1. All elements should be active.

We begin by stepping through **main**. Note that we use a **next** command at the beginning of **main**. When you enter the first function in a compilation unit, use the **next** command to avoid stepping into functions required by C* runtime support. Note also the use of the **step** command to enter the **find_primes** function; we use **step** in this case so that we do go into the function.

```
(dbx) stop in main
(1) stop in main
(dbx) run
Running: primes
stopped in main at line 52 in file "primes.cs"
   52   main() {
(dbx) n
stopped in main at line 59 in file "primes.cs"
   59       find_primes(&is_prime);
(dbx) s
stopped in find_primes at line 34 in file "primes.cs"
   34   void find_primes(char:current *is_prime_p) {
(dbx) n
stopped in find_primes at line 40 in file "primes.cs"
   40 is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
(dbx) n
```

```
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
```

We now call the appropriate C* debugging functions. We use **CMC_define_region** to define a region from 0 through 9 of shape **s**. We call **CMC_define_type** to define **is_candidate** as a **char**. We call **CMC_define_width** to specify that **is_candidate** is to be printed as an **int** with field width 2.

```
(dbx) dr(s, 0, 9)
 Region set to [0, 9]
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
(dbx) dt(is_candidate, CMC_char)
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
(dbx) dw(CMC_char, -2)
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
```

Finally, we call **CMC_print_region** to print the values of **is_candidate**:

```
(dbx) ppr(is_candidate)
```

But something has gone very wrong. Instead of printing out 10 values, the debugger prints out screen after screen of numbers! Puzzled, we call **CMC_print_status** to see what's going on:

```
(dbx) pstat

         CMC debugging status

Default type: CMC_double
Default formats:
              CMC_bool          %1d
              CMC_char          %2d
              CMC_short         %6d
              CMC_int           %10d
              CMC_long_int      %10ld
              CMC_float         %10.4g
              CMC_double        %10g
              CMC_long_double   %10g
Parallel variable types:
   field id 458760   type CMC_char
```

```
Shape regions:
   vp set 2 [0,9]
   vp set 0 [0,8191]
Context stacks:
Shape views:
   vp set 0 0

stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
```

Why are there two "Shape regions" defined? The first one (**vp set 2**) clearly refers to shape **s**, since we defined the region [0,9] for shape **s**; the second one is much larger, and that is apparently the region that was being printed for **is_candidate**. Why isn't **is_candidate** of shape **s**?

This gives us enough evidence to figure out the bug—there is no **with** statement making **s** the current shape. Without the **with** statement, **is_candidate** becomes a parallel variable of the default shape, which for this implementation of C* is the shape **physical**; see Chapter 2.

We now quit **dbx**:

```
(dbx) quit
```

and add the required **with** statement.

5.9.4 The Second Session

Here is the revised program, with the revision shown in bold:

```
 1. #define MAXIMUM_PRIME 16384
 2.
 3. #define FALSE 0
 4. #define TRUE 1
 5. #define FIRST_PRIME 2
34. void find_primes(char:current *is_prime_p) {
35.  char:current is_candidate;
36.  int minimum_prime;
37.
38.
39.
```

```
40.  is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
41.
42.  do
43.    where(is_candidate) {
44.      minimum_prime = <?= pcoord(0);
45.      where(pcoord(0) % minimum_prime)
46.        is_candidate = FALSE;
47.      [minimum_prime](*is_prime_p) = TRUE;
48.    }
49.  while(|= is_candidate);
50. }
51.
52. main() {
53.  shape [MAXIMUM_PRIME]s;
54.
55.  char:s is_prime;
56.  int i;
57.
58.   with(s)
59.    find_primes(&is_prime);
60.  for(i=0; i<MAXIMUM_PRIME; i++)
61.    if([i]is_prime)
62.      printf("The next prime number is %d\n", i);
63. }
```

We now debug this new version. We issue the same debugging calls to define the region, the width, and the type of **is_candidate**. Let's pick up the story where we left off in the previous session—as we attempted to print out a region of **is_candidate** at line 43:

```
(dbx) ppr(is_candidate)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ---------------------------------------
            0   0   1   1   1   1   1   1   1   1

stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
```

This is the behavior we want. We then enter the **where** statement and see what happens to the context:

```
(dbx) n
stopped in find_primes at line 44 in file "primes.cs"
   44         minimum_prime = <?= pcoord(0);
(dbx) ppr(is_candidate)

[./*Row*/]

             0   1   2   3   4   5   6   7   8   9
           ----------------------------------------
            ( 0)( 0)  1   1   1   1   1   1   1   1

stopped in find_primes at line 44 in file "primes.cs"
   44         minimum_prime = <?= pcoord(0);
```

This too is the behavior we want: positions 0 and 1 are now inactive. (Note that inactive positions are shown in parentheses.) We go to the next line of code and print out the value of **minimum_prime**:

```
(dbx) n
stopped in find_primes at line 45 in file "primes.cs"
   45         where(pcoord(0) % minimum_prime)
(dbx) p minimum_prime
minimum_prime = 2
```

This is correct; 2 is the lowest coordinate of all the active positions. We step again and print out the region for **is_candidate**:

```
(dbx) n
stopped in find_primes at line 46 in file "primes.cs"
   46           is_candidate = FALSE;
(dbx) ppr(is_candidate)

[./*Row*/]

             0   1   2   3   4   5   6   7   8   9
           ----------------------------------------
            ( 0)( 0)( 1)  1 ( 1)  1 ( 1)  1 ( 1)  1

stopped in find_primes at line 46 in file "primes.cs"
   46           is_candidate = FALSE;
```

This doesn't look right. We wanted to select the positions whose coordinates were multiples of **minimum_prime**; instead, those positions have been turned off, and the

odd-numbered positions are active. We forge ahead, however, and see what happens after line 46:

```
(dbx) n
stopped in find_primes at line 47 in file "primes.cs"
   47          [minimum_prime](*is_prime_p) = TRUE;
(dbx) ppr(is_candidate)

[./*Row*/]

             0   1   2   3   4   5   6   7   8   9
           ----------------------------------------
           ( 0)( 0)  1   0   1   0   1   0   1   0

stopped in find_primes at line 47 in file "primes.cs"
   47          [minimum_prime](*is_prime_p) = TRUE;
(dbx) n
stopped in find_primes at line 48 in file "primes.cs"
   48        }
```

No, this doesn't look good. We have come out of the inner **where** statement, and only the even coordinates are set to 1. This is the opposite of what we wanted. We decide to take a look at **is_prime_p**. Note that, even though it's a pointer to a parallel variable, we can treat it as a parallel variable in **dbx**. First, we make sure to define its type (the debugger might abort if we try to print it as a **double**, which is the default). Then we print it.

```
(dbx) dt(is_prime_p, CMC_char)
stopped in find_primes at line 47 in file "primes.cs"
   48        }
(dbx) ppr(is_prime_p)

[./*Row*/]

             0   1   2   3   4   5   6   7   8   9
           ----------------------------------------
             0   0   1   0   0   0   0   0   0   0

stopped in find_primes at line 48 in file "primes.cs"
   48        }
```

This looks all right: all the positions are active again, and element [2] is set to 1. We step through the outer **where** statement again as the **do** loop continues, and then print out **is_candidate** and **minimum_prime**:

```
(dbx) n
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
(dbx) n
stopped in find_primes at line 44 in file "primes.cs"
   44         minimum_prime = <?= pcoord(0);
(dbx) n
stopped in find_primes at line 45 in file "primes.cs"
   45         where(pcoord(0) % minimum_prime)
(dbx) ppr(is_candidate)

[./*Row*/]

             0   1   2   3   4   5   6   7   8   9
          ----------------------------------------
          ( 0)( 0)  1 ( 0)  1 ( 0)  1 ( 0)  1 ( 0)

stopped in find_primes at line 45 in file "primes.cs"
   45         where(pcoord(0) % minimum_prime))

(dbx) p minimum_prime
minimum_prime = 2
```

We see now why the program went into an infinite loop. We wanted to select the coordinates that were multiples of **minimum_prime**, so that we could set **is_candidate** to 0 in those positions; instead, we did the opposite in line 45. The even-numbered positions remain set to 1 for **is_candidate**, and therefore remain active; as a result, **minimum_prime** always comes out 2, and the program never stops. The solution is to use **!** to negate the condition for the **where** statement in line 45.

We quit **dbx**, fix the bug, and try again.

5.9.5 The Third Debugging Session

The revised program is shown below; the revised line is in bold.

```
1. #define MAXIMUM_PRIME 16384
2.
3. #define FALSE 0
4. #define TRUE 1
5. #define FIRST_PRIME 2
```

```
34. void find_primes(char:current *is_prime_p) {
35.  char:current is_candidate;
36.  int minimum_prime;
37.
38.
39.
40.  is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
41.
42.  do
43.    where(is_candidate) {
44.      minimum_prime = <?= pcoord(0);
45.      where(!(pcoord(0) % minimum_prime))
46.        is_candidate = FALSE;
47.      [minimum_prime](*is_prime_p) = TRUE;
48.    }
49.  while(|= is_candidate);
50. }
51.
52. main() {
53.  shape [MAXIMUM_PRIME]s;
54.
55.  char:s is_prime;
56.  int i;
57.
58.   with(s)
59.    find_primes(&is_prime);
60.  for(i=0; i<MAXIMUM_PRIME; i++)
61.    if([i]is_prime)
62.      printf("The next prime number is %d\n", i);
63. }
```

This time we pick up the story where we started to go wrong last time: in the inner **where** statement of **find_primes**, where we want to set the context to the positions whose coordinates are multiples of **minimum_prime**. Once again, we have already defined the region, the width, and the type of **is_candidate**.

```
(dbx) n
stopped in find_primes at line 45 in file "primes.cs"
   45         where(!(pcoord(0) % minimum_prime))
(dbx) n
stopped in find_primes at line 46 in file "primes.cs"
   46           is_candidate = FALSE;
(dbx) ppr(is_candidate)
```

```
[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ----------------------------------------
          ( 0)( 0)  1 ( 1)  1 ( 1)  1 ( 1)  1 ( 1)

stopped in find_primes at line 46 in file "primes.cs"
   46           is_candidate = FALSE;
```

This looks right. The even-numbered positions are active. We leave the inner **where** statement and look at **is_candidate** once again:

```
(dbx) n
stopped in find_primes at line 47 in file "primes.cs"
   47         [minimum_prime](*is_prime_p) = TRUE;
(dbx) ppr(is_candidate)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ----------------------------------------
          ( 0)( 0)  0   1   0   1   0   1   0   1

stopped in find_primes at line 47 in file "primes.cs"
   47         [minimum_prime](*is_prime_p) = TRUE;
```

The positions with odd-numbered coordinates are now set to 1 and therefore are still candidates. This is the correct behavior. We then look at **is_prime_p**:

```
(dbx) dt(is_prime_p, CMC_char)
stopped in find_primes at line 47 in file "primes.cs"
   47         [minimum_prime](*is_prime_p) = TRUE;
(dbx) ppr(is_prime_p)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ----------------------------------------
          ( 0)( 0)  1   0   0   0   0   0   0   0

stopped in find_primes at line 47 in file "primes.cs"
   47         [minimum_prime](*is_prime_p) = TRUE;
```

But that can't be right: line 47 hasn't been executed yet to set element [2] to 1 for **is_prime_p**. So why is it set? At this point, we notice another error in our program: we forgot to initialize **is_prime_p** to 0 in the **find_primes** function. This means that the parallel variables retain their values from the previous execution of the program (since we haven't coldbooted the CM), and element [2] of **is_prime_p** remains set to 1.

We make the required change and try again.

5.9.6 The Final Debugging Session

Here is what the code looks like now (the latest revision is in bold):

```
 1. #define MAXIMUM_PRIME 16384
 2.
 3. #define FALSE 0
 4. #define TRUE 1
 5. #define FIRST_PRIME 2
34. void find_primes(char:current *is_prime_p) {
35.  char:current is_candidate;
36.  int minimum_prime;
37.
38.  *is_prime_p = FALSE;
39.
40.  is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
41.
42.  do
43.    where(is_candidate) {
44.      minimum_prime = <?= pcoord(0);
45.      where(!(pcoord(0) % minimum_prime))
46.        is_candidate = FALSE;
47.      [minimum_prime](*is_prime_p) = TRUE;
48.    }
49.  while(|= is_candidate);
50. }
51.
52. main() {
53.  shape [MAXIMUM_PRIME]s;
54.
55.  char:s is_prime;
56.  int i;
57.
```

```
58.  with(s)
59.    find_primes(&is_prime);
60.  for(i=0; i<MAXIMUM_PRIME; i++)
61.    if([i]is_prime)
62.      printf("The next prime number is %d\n", i);
63. }
```

We rerun **dbx** and look at **is_prime_p** before line 38 is executed:

```
% dbx primes
Reading symbolic information...
Read 1711 symbols
(dbx) stop in find_primes
(1) stop in find_primes
(dbx) run
Running: primes
stopped in find_primes at line 34 in file "primes.cs"
   34   void find_primes(char:current *is_prime_p) {
(dbx) n
stopped in find_primes at line 38 in file "primes.cs"
   38     *is_prime_p = FALSE;
(dbx) dt(is_prime_p, CMC_char)
stopped in find_primes at line 38 in file "primes.cs"
   38     *is_prime_p = FALSE;
(dbx) dr(s, 0, 9)

 Region set to [0, 9]

stopped in find_primes at line 38 in file "primes.cs"
   38     *is_prime_p = FALSE;
(dbx) dw(CMC_char, -2)
stopped in find_primes at line 38 in file "primes.cs"
   38     *is_prime_p = FALSE;
(dbx) ppr(is_prime_p)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ----------------------------------------
            0   0   1   0   0   0   0   0   0   0

stopped in find_primes at line 38 in file "primes.cs"
   38     *is_prime_p = FALSE;
```

The values have not changed from the previous execution of the program. We then look at **is_prime_p** after line 38 is executed. As required, all values are now zero.

```
(dbx) n
stopped in find_primes at line 40 in file "primes.cs"
   40 is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
(dbx) ppr(is_prime_p)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ---------------------------------------
            0   0   0   0   0   0   0   0   0   0

stopped in find_primes at line 40 in file "primes.cs"
   40 is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
```

We continue go step through the program, looking at the behavior of **is_prime_p**:

```
(dbx) n
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
(dbx) n
stopped in find_primes at line 44 in file "primes.cs"
   44         minimum_prime = <?= pcoord(0);
(dbx) n
stopped in find_primes at line 45 in file "primes.cs"
   45         where(!(pcoord(0) % minimum_prime))
(dbx) n
stopped in find_primes at line 46 in file "primes.cs"
   46           is_candidate = FALSE;
(dbx) n
stopped in find_primes at line 47 in file "primes.cs"
   47         [minimum_prime](*is_prime_p) = TRUE;
(dbx) n
stopped in find_primes at line 48 in file "primes.cs"
   48       }
(dbx) ppr(is_prime_p)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ---------------------------------------
            0   0   1   0   0   0   0   0   0   0
```

```
stopped in find_primes at line 48 in file "primes.cs"
   48       }
```

The first prime number is set correctly. We go through the loop again:

```
(dbx) n
stopped in find_primes at line 43 in file "primes.cs"
   43       where(is_candidate) {
(dbx) n
stopped in find_primes at line 44 in file "primes.cs"
   44         minimum_prime = <?= pcoord(0);
(dbx) n
stopped in find_primes at line 45 in file "primes.cs"
   45         where(!(pcoord(0) % minimum_prime))
(dbx) print minimum_prime
minimum_prime = 3
(dbx) n
stopped in find_primes at line 46 in file "primes.cs"
   46           is_candidate = FALSE;
(dbx) n
stopped in find_primes at line 47 in file "primes.cs"
   47         [minimum_prime](*is_prime_p) = TRUE;
(dbx) n
stopped in find_primes at line 48 in file "primes.cs"
   48       }
(dbx) ppr(is_prime_p)

[./*Row*/]

            0   1   2   3   4   5   6   7   8   9
          ---------------------------------------
            0   0   1   1   0   0   0   0   0   0

stopped in find_primes at line 48 in file "primes.cs"
   48       }
```

The second prime number is correctly set. We feel confident enough now to see if the program can run to completion:

```
(dbx) cont
The next prime number is 2
The next prime number is 3
The next prime number is 5
```

```
The next prime number is 7
The next prime number is 11
The next prime number is 13 ...
execution completed, exit code is 0
program exited with 0
```

The program runs correctly (we spare you from having to read the rest of the prime numbers). There is nothing left to do, then, except to quit **dbx**:

```
(dbx) quit
```

Appendix A

Man Pages

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

NAME

cs – C* compiler

SYNOPSIS

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

DESCRIPTION

cs is the C* compiler. It translates C* programs into C/Paris, and then invokes the C compiler *cc*(1) to make an executable load module. 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 *cc*, file names ending in **.o** are treated as object files and are passed directly to *ld (1)*, file names ending in **.a** are treated as object libraries and are passed directly to *ld*. In addition, VAX file names ending in **.obj** are treated as object files and are passed directly to *ld*.

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

OPTIONS

Options specific to *cs*

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

–dryrun Show, but do not execute compilation steps.

–force Force input files with the .c suffix to be passed through the C* compilation phase rather than just the cc phase. This option is useful for processing ANSI C programs.

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

–h Synonym for **-help**.

–keep *keyword*
Retain intermediate files generated by the compilation process. Currently the only legal value for *keyword* is **c**.

–noline Suppress #line directives in the output C file.

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

–O*level* Invoke the C* optimizer with level *level*. Legal values for *level* are **0**, **1**, and **2**. Optimization levels are described in more detail below.

–ucode *number*
Link with CM software version *number*.

–verbose Display informational messages as the compilation proceeds.

–v Synonym for **-verbose**.

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

–warn Suppress warnings from the C* compilation phase.

–Z*comp switch*
Pass option *switch* to component *comp*. *comp* is either **cpp** or **cc**. 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 *dbx*(1).

–I*dir* Seek **#include** files whose names do not begin with "/" first in the directory of the *file* argument, then in directories named in **–I** options, then in directories on a standard list.

–L*dir* Add directory *dir* to the list of directories on the object library search path.

–l*x* This option is an abbreviation for the library name '/lib/lib*x*.a', where *x* is a string. If that does not exist, *ld(1)* tries '/usr/lib/lib*x*.a'. If that does not exist, *ld(1)* tries '/usr/local/lib/lib*x*.a'. A library is searched when its name is encountered, so the placement of a **–l** is significant.

–o *output* Name the final output file *output*. If this option is used, the file **a.out** will be left undisturbed.

–pg Link with profiling libraries for use with *gprof(1)*.

–U*name* Undefine the preprocessor symbol *name*.

OPTIMIZATION

The C* compiler has three levels of optimization: zero, one and two.

Level zero turns the optimizer off. This is not normally useful for users. Specify -O0 to turn the optimizer off.

Level one performs local copy propagation, dead-code elimination, variable minimization and some peephole optimizations. This is the default behavior of the compiler. You don't need to specify anything on the command line to get this behavior.

Level 2 performs the same optimizations as level one, except that user variables are optimized in addition to compiler-generated temporaries. A program compiled at optimization level two will generally be too highly-optimized for use with the debugger. Specify -O or -O2 to turn on optimization level 2.

DEFAULT SYMBOLS

The C* compiler provides the following default symbols, each defined as **1**. These symbols are useful in **#ifdef** statements to isolate code for one of the particular cases. Thus, these symbols can be useful for ensuring portable code. The **cstar** symbol can be used to share source between a C and a C* program.

cstar	The C* language (as opposed to the C language)
unix	Any UNIX system
ultrix	ULTRIX only
vax	VAX only
sun	Sun only
sparc	Sun-4 only

FILES

file.cs	input C* code
file.cpp	intermediate, preprocessed code
file..c	intermediate C/Paris code
file.o	relocatable object file
file.obj	VAX JBL object file
file.a	object library
a.out	linked executable output
/lib/cpp	C preprocessor
/usr/include/cs	directory of C* include files
/usr/local/lib/cstar	C* compiler executable
/bin/cc	C compiler
/usr/local/lib/libcsrt.a	C* library linked by default
/usr/local/lib/libcsrt-pg.a	C* default profiling library
/usr/local/lib/libparis.a	Paris library linked by default

SEE ALSO

cc(1), dbx(1)

Thinking Machines Corporation, C* documentation set: *C* Programming Guide*; *C* Release Notes*; and *C* User's Guide*.

DIAGNOSTICS

Occasional messages may be produced by the C compiler, assembler, or linker, in addition to those normally produced by the C* compilation phase.

RESTRICTIONS

Bugs and restrictions are listed in the *C* Release Notes*.

NAME

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);

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

type:current copy_multispread(*type*:current *sourcep, int axis_map, 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, ...);

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

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

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

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

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

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

void to_torus_dim(*type*:current *destp, *type*:current source, 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, *C* Users' Guide*, *C* Programming Guide*

NAME

cstimer.h – C* timer functions

SYNTAX

#include <cstimer.h>

SYNOPSIS

```
extern unsigned CMC_number_of_timers;
void CMC_timer_clear (unsigned timer);
void CMC_timer_print (unsigned timer);
double CMC_timer_read_cm_busy (unsigned timer);
double CMC_timer_read_cm_idle (unsigned timer);
double CMC_timer_read_elapsed (unsigned timer);
int CMC_timer_read_run_state (unsigned timer);
int CMC_timer_read_starts (unsigned timer);
void CMC_timer_set_starts (unsigned timer, unsigned value);
void CMC_timer_start (unsigned timer);
void CMC_timer_stop (unsigned timer);
```

DESCRIPTION

The timer functions under C* are wrappers for the Paris timer functions. See the *CM User's Guide* for more information about the Paris timer functions.

SEE ALSO

CM User's Guide

NAME

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, truncate;

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 truncate(float:current);

double:current truncate(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, *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. This is particularly the case on the VAX, because the VAX and the Connection Machine processors use different floating-point formats (the VAX uses VAX format, while the Connection Machine processors use IEEE format).

NAME

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 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*

NAME

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**.

NAME

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

Symbols

A

B

C

T

U

V

W