The
Connection Machine
System

CM User's Guide

Version 6.1

October 1991

Thinking Machines Corporation
Cambridge, Massachusetts

Reprinted, with corrections, December 1992

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

The
Connection Machine
System

CM User's Guide

Version 6.1

October 1991

Thinking Machines Corporation
Cambridge, Massachusetts

Reprinted, with corrections, December 1992

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

Contents

Part IV I/O on the Connection Machine System

Part V In the Lisp Environment

About This Manual

Objectives of This Manual

This manual is an introduction to the CM-2 or CM-200 series Connection Machine system. Read this manual to learn the basics of how to develop and execute data parallel programs using the CM system.

Intended Audience

Anyone who uses a CM should read this manual. It is applicable to all front-end computers that connect to the CM; specifically:

- Front-end computers running a version of the UNIX operating system. (Unless otherwise noted, in this manual "UNIX" refers to both the SunOS and ULTRIX operating systems.) We don't assume that you know anything about the CM; we do assume that you are familiar with UNIX.

- Front-end computers from the Symbolics 3600 series of Lisp machines. We don't assume that you know anything about the CM; we do assume that you are familiar with the operation of the Lisp machine.

Revision Information

This manual has been revised to reflect CM System Software, Version 6.1. In particular, Chapter 5 is entirely new. For information on features new to Version 6.1, see the *CMSS V6.1 System Software Summary.*

Organization of This Manual

This manual is written for users who wish to program the CM from either a UNIX front end or a Symbolics 3600-series computer.

UNIX front end users should read Part I of this document for a general introduction to the Connection Machine, then Parts II through IV, which discuss programming the CM from UNIX. They should read Part V only if they intend to program in *Lisp or Lisp/Paris.

Symbolics front end users should read Part I, and then skip to Part V, which discusses programming the CM in the Lisp environment (see Figure 1).

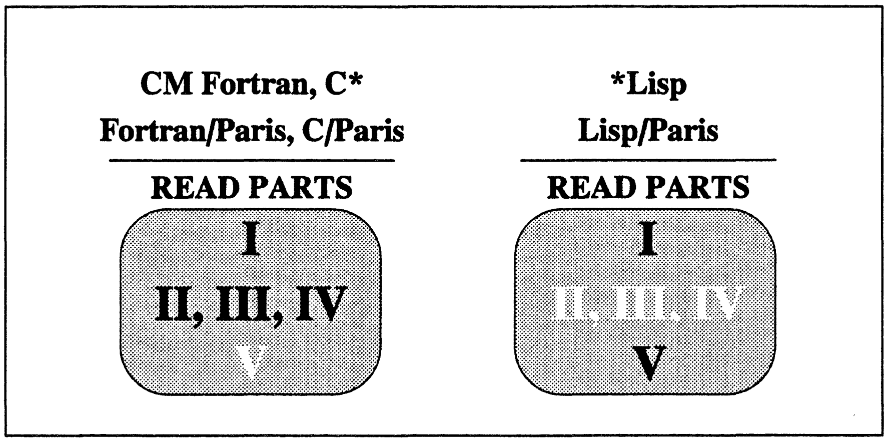

Figure 1. Suggested reading paths, depending on the language in which you intend to program

Part I **Introduction to the Connection Machine System**
Part I gives an overview of the hardware and software components of the Connection Machine system.

Part II **Using the CM Operating System**
Part II describes some basic commands in the CM operating system for UNIX users. Chapter 2 describes how to run programs on the CM; Chapter 3 describes other useful commands.

Part III **Programming with the Connection Machine System**
Part III discusses how to program using your UNIX front end and the CM. Chapter 4 gives the basics of the programming process. Chapter 5 describes how to attach to and detach from a CM from within a program. Chapter 6 describes programming tools like the CM timing utility, its checkpointing package, and its run-time safety checker.

Part IV **I/O on the Connection Machine System**
Part IV provides an overview of I/O on the CM, focusing especially on the commands for using the CM file system.

Part V **In the Lisp Environment**
Part V describes how to use the CM system when in a Lisp environment running on your UNIX or Symbolics front end.

There are six appendixes:

- Appendix A describes back-compatibility mode on the CM.
- Appendix B describes the DFS system, unsupported software that can be used to manage large file sets.
- Appendix C lists Paris functions that may affect the performance of a program running under timesharing using a VAX front end.
- Appendix D is an overview of UNIX features that are important to CM users.
- Appendix E is a glossary.
- Appendix F provides UNIX man pages for CMost user commands.

Related Documents

Some of the material in this manual is covered in a different way in this Thinking Machines Corporation publication:

- *Connection Machine CM-200 Series Technical Summary*

You need not be familiar with the technical summary before reading this manual, however.

If you are involved in configuring or managing a Connection Machine system, you should also read this manual:

- *CM System Administrator's Guide*

Consult the documentation for your front-end computer to learn about its version of UNIX. In addition, there are many books you can choose from to obtain further information about UNIX. For example:

- *The UNIX Programming Environment*, Brian W. Kernighan and Rob Pike. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.

Finally, consult the other volumes of the Connection Machine documentation set to learn more about many of the topics discussed in this manual.

Notation Conventions

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

Convention	Meaning
`boldface`	UNIX and CM System Software commands, command options, and file names. Also, Paris, C, and Lisp language elements, such as keywords, operators, and function names, when they appear embedded in text.
`UPPERCASE`	Fortran language elements, when they appear embedded in text.
Ctrl-D	Combinations of keystrokes are shown with a connecting hyphen. To type the Ctrl-D combination, for example, press the D key while holding down the Control key.
italics	Parameter names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
% **`boldface`** `typewriter`	In interactive examples, user input is shown in **`boldface`** and system output is shown in `typewriter` font.

Part I
Introduction to the Connection Machine System

Chapter 1

The Connection Machine System

The Connection Machine system is an integrated combination of hardware and software designed for high-speed data parallel computing. This chapter introduces data parallel computing and the Connection Machine system; it also provides an overview of how to use the CM to write and execute data parallel programs.

1.1 Data Parallel Computing

In conventional computing, a computer has a single central processor, which operates on data sequentially. If the same operation is to be performed on many data elements, the computer must still perform the operation separately on each element, one after another.

In data parallel computing, there are many processors, and each data element is associated with a processor. All processors can then perform the same operation on all data elements at the same time. This kind of computing takes advantage of the natural computational parallelism inherent in problems with large data sets. For example:

- A graphics program might store pixels one per processor and then have each processor calculate the color value for its pixel, all at the same time.
- A text retrieval program might store articles one per processor and then have each processor search its article for a keyword.
- A modeling program (for example, one that simulates fluid flow) might create a large number of individual cells, stored one per processor. Each cell might have a small number of possible states, which are

simultaneously updated at each "tick" of a clock according to a set of rules that are applied to each cell.

The result can be a dramatic decrease in the amount of time it takes to run such programs.

Programming can also become simpler using the data parallel model, since it avoids the complexity of trying to solve a naturally parallel problem in a serial manner.

1.2 The Hardware of the Connection Machine System

The Connection Machine system provides hardware and software to support the data parallel model of computing. Using the CM, you can write and execute data parallel programs to solve the largest computational problems. This section describes the hardware components of the system; Sections 1.3 and 1.4 describe the software components.

A fully configured Connection Machine system contains these hardware components:

- A parallel processing unit, containing thousands of individual processors
- One or more front-end computers
- An I/O system, which can contain:
    - DataVault mass storage systems
    - A CM-HIPPI system for connecting the CM to an UltraNet network or to other devices that support the high-performance parallel interface (HIPPI) standard
    - General-purpose I/O computers with a VMEbus
    - Other I/O devices such as magnetic tape drives
- A graphic display system

All Connection Machine systems contain a parallel processing unit and at least one front-end computer; other parts of the system are optional. Check with your system administrator for the exact configuration of your system.

1.2.1 The Parallel Processing Unit

The parallel processing unit is the heart of the Connection Machine system (so much so that the term "CM" is often used to refer only to it, and not to the entire system).

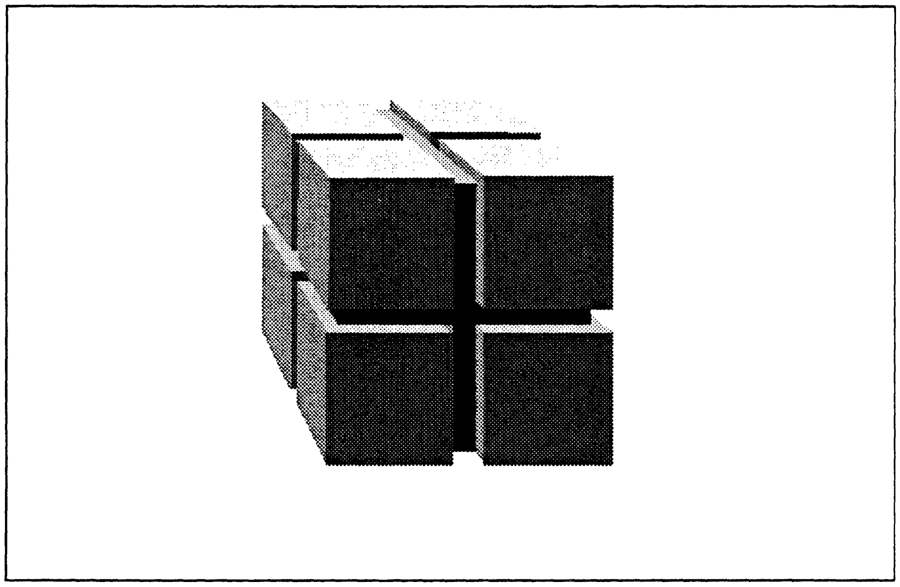

Figure 1. The CM parallel processing unit

The Sequencer

The individual processors within a parallel processing unit are controlled by a device called a *sequencer.* The sequencer's job is to decode commands and to broadcast them to the processors for parallel execution. CMs have up to four sequencers.

Virtual Processors

If there are more data elements than there are processors (which is generally the case), the system creates *virtual* processors by dividing up the memory associated with each physical processor. Thus, the same program can run without change on different parallel processing units with different numbers of physical processors—but the more physical processors, the faster it runs.

Floating-Point Accelerator

The CM-2 parallel processing unit may contain either a single-precision (32-bit) or double-precision (64-bit) floating-point accelerator. Both options support IEEE standard floating-point formats and operations. They each increase the rate of floating-point operations by more than a factor of 20.

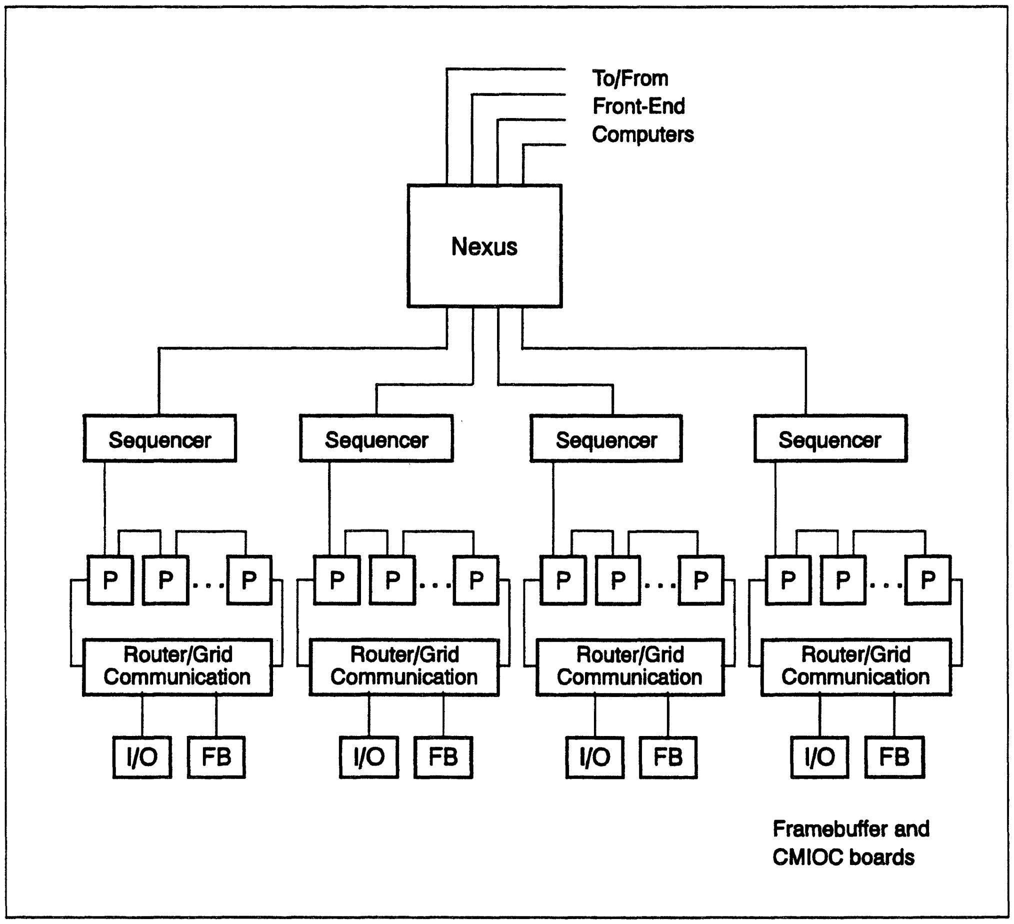

Figure 2. Architecture of the CM parallel processing unit

Communication

The processors are interconnected by a high-speed communication device called a *router*. The router allows processors to send data to or receive data from other processors, in parallel. The parallel processing unit also supports a faster form of communication called *grid communication* (also called *NEWS* *communication*), which allows processors to communicate with their neighbors in a multidimensional grid.

I/O Controllers and Framebuffer Modules

The parallel processing unit also contains I/O channels. Either a CM I/O controller (CMIOC) or a framebuffer module can be connected to each I/O channel. The CMIOC connects the parallel processing unit to the CMIO bus, and the framebuffer module connects it to a high-resolution color monitor; see Section 1.2.3 and Section 1.2.4.

Sections

Processors are divided into *sections*. For example, a 64K parallel processing unit can be divided into four sections of 16K processors each. Each of these sections can be treated as a separate parallel processing unit, or they can be grouped together so that more physical processors are available to the user. Separate sections have their own sequencers, routers, and I/O channels.

The Nexus

The *nexus* is a switch that allows multiple front-end computers to be connected to a single parallel processing unit. It can connect any front end to any section, or valid group of sections, in the parallel processing unit.

1.2.2 The Front End

To the user, the parallel processing unit appears as an extension of the normal environment of a standard serial computer. This serial computer is referred to as a *front end*; it can be a Sun-4 Workstation or one of several models of the VAX minicomputer; on the CM-2 it can also be a Symbolics 3600-series Lisp machine. The front end is the user's gateway to the Connection Machine system. It has three main functions:

- To provide an environment for developing and debugging applications.
- To run applications, transmitting instructions and data to the parallel processing unit.

- To provide maintenance and operations utilities for controlling the CM and diagnosing problems.

The front end communicates with the CM parallel processing unit via a board called the *front-end bus interface* (FEBI). A VAX or Sun front end can have up to four FEBIs, allowing four separate connections to the CM at the same time. In addition, up to four front ends can be attached to a single CM-2 or CM-200 parallel processing unit; up to two front ends can be attached to a CM-2a.

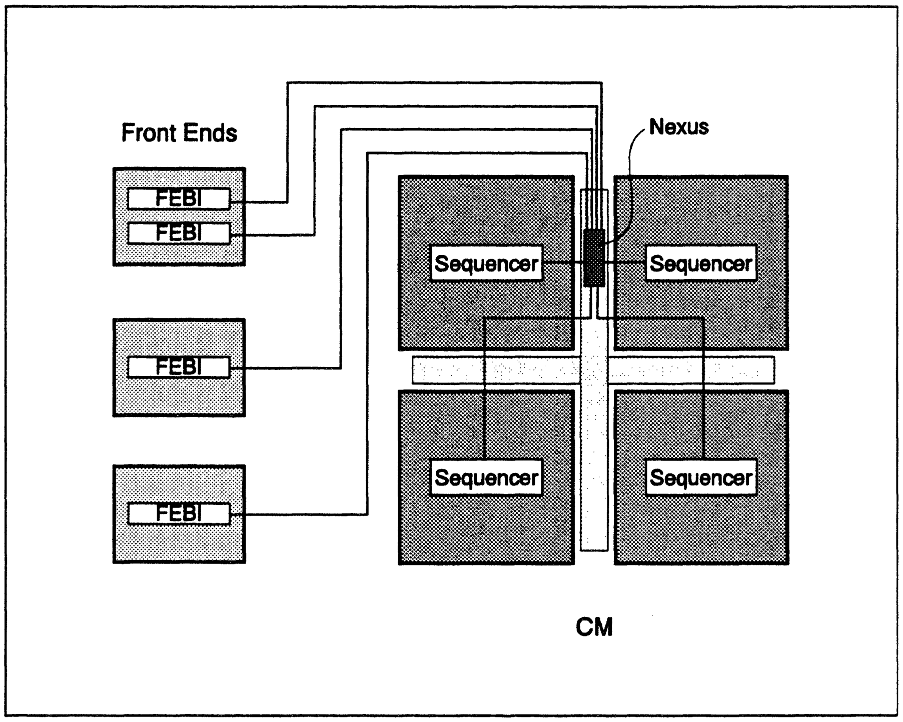

Figure 3. Front ends connected to a CM

1.2.3 The I/O System

The CM I/O system provides a means for moving large amounts of data into and out of the parallel processing unit at high speeds. The I/O hardware consists of the following:

- The I/O channels within the parallel processing unit. There are up to two I/O channels for every group of 8K processors.

- The Connection Machine I/O bus. Each I/O channel can connect to this bus via a CM I/O controller (CMIOC). The bus provides high-speed data transfer (up to 50 Mbytes/sec) among the components of the CM I/O system. Each I/O bus can support up to 16 devices, and there can be multiple buses in the Connection Machine system.

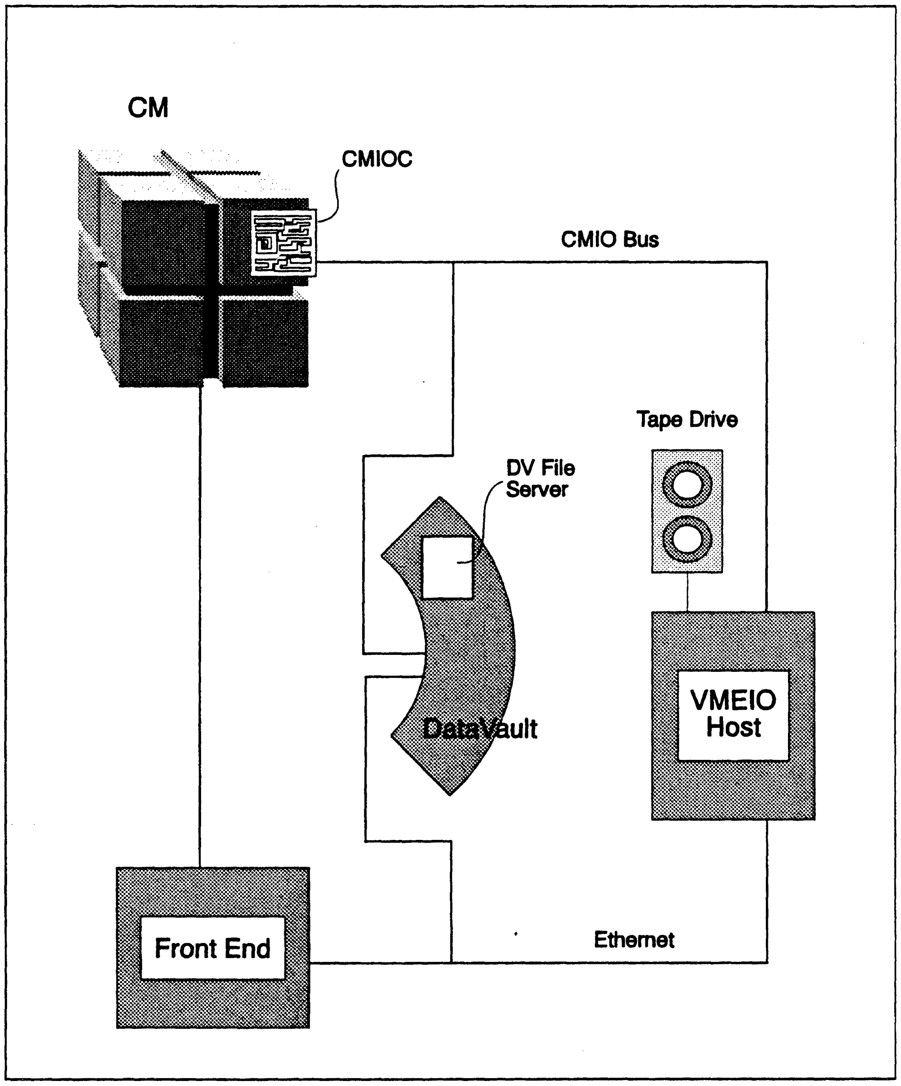

Figure 4. The CM I/O system

- The DataVault mass storage system, which provides storage for up to 20 Gbytes of data on up to 78 disk drives. Each DataVault can be connected to up to two I/O buses.

- A VMEIO interface, which provides a high-speed data path between a CMIO bus and computers having a VMEbus. The VMEIO computer makes it possible to connect a variety of other devices, such as magnetic tape drives, to the CM I/O system.
- A CM-HIPPI system, which connects the CM to an UltraNet network, or directly to another supercomputer or device that supports the HIPPI standard.
- An Ethernet local area network that links the VME computer, the CM-HIPPI system, the DataVault, and the front end. The CM I/O system uses the Ethernet to carry I/O requests from the front end, responses to these requests from the VMEIO computer and the DataVault, and data to and from the front end.

1.2.4 Graphic Display System

The Connection Machine system provides hardware and software for quickly visualizing the huge data sets that are typically used in data parallel applications. The graphics hardware consists of the framebuffer module and a high-resolution color monitor. The framebuffer, as described earlier, is a board connected to the I/O channel of a parallel processing unit. It can transfer graphical information from the processors to the monitor at up to 40 Mbytes per second. This lets you examine data graphically in real time. The software supporting this and other forms of visualization is described in Chapter 6.

1.3 Programming in the Connection Machine System

The Connection Machine system provides several high-level languages for data parallel programming. They are:

- C* (pronounced "see-star"), a data parallel extension of the C programming language.
- CM Fortran, an implementation of the Fortran 77 programming language, extended with array-handling facilities from Fortran 90.
- *Lisp (pronounced "star-lisp"), a data parallel extension of Common Lisp.

In addition, it provides a lower-level parallel instruction set called *Paris*. User interfaces to the Paris instructions are provided for Fortran, C, and Lisp. The instructions can also be called from any of the high-level data parallel languages. Paris calls can sometimes provide programming efficiencies beyond those available in the high-level language.

Note that the high-level data parallel languages are extensions of standard serial languages. Data parallel programs are generally similar to conventional serial programs. Both use a single sequence of instructions; however, in the data parallel case, some of these instructions cause operations to be performed on many data elements at once. CM Fortran, as an implementation of existing standards, adds no new syntax to these standards. C* and *Lisp add a small amount of new syntax to their serial counterparts.

1.3.1 Developing, Compiling, Executing, and Debugging Data Parallel Programs

Your front end has a compiler or interpreter for one or more of the high-level data parallel languages. The programming process is straightforward:

- *Write a program* as you normally would for the language's serial counterpart, using the front end's development environment.
- *Compile the program* using the Connection Machine compiler for the language (*Lisp programs can be either compiled or interpreted).
- *Execute the program* by first "attaching" to one or more sequencers of a parallel processing unit, then running the program as you normally would. You can also execute your program from a UNIX front end by submitting it to a queue in the CM batch system. In both cases, your program may have exclusive use of the sequencers, or it may run under timesharing with other programs. Program execution is described for all languages except *Lisp and Lisp/Paris in Chapter 2; *Lisp and Lisp/Paris are described in Chapter 8.
- *Debug the program* using a standard debugger for your front end, such as **dbx** on UNIX front ends. (Debugging functions for **dbx** are provided for each high-level language; these functions let you, for example, print out individual data elements for processors.) In addition, a **cmdbx** debugger is available for CM Fortran programs.

This process is described in more detail in Chapter 4 for all languages except *Lisp and Lisp/Paris; *Lisp and Lisp/Paris are discussed in Chapter 8.

1.3.2 Programming Tools

You can use standard programming tools available on your front end. In addition, the Connection Machine system provides other tools designed specifically for CM programming:

- *Safety checking.* The CM system provides a run-time safety utility that checks for Paris-level errors and inconsistencies in data parallel programs.
- *Timing.* The CM system's timing utility lets you insert instructions into a program to calculate the amount of time the program (or sections of it) uses the CM.
- *Profiling.* The CM system has special libraries that allow you to use UNIX's `gprof` profiling utility with data parallel programs. The `gprof` utility produces a summary of the amount of time spent in each routine and a list of which routines call, and are called by, other routines. (These libraries are not available for *Lisp.)
- *Checkpointing.* The CM system's checkpointing package lets you save a program's state at specified points during its execution. You can subsequently restart execution of the program from the point at which it was saved. (Checkpointing is not available for *Lisp.)

These tools are described in more detail for UNIX front ends in Chapter 6, and for Symbolics front ends in Chapter 8.

1.3.3 Programming Libraries

The Connection Machine system provides programming libraries in the following areas:

- *I/O.* A program can include calls to library routines that perform various I/O functions—for example, reading data into the parallel processing unit from an I/O device.
- *Graphics and Visualization.* There are graphics libraries available that, among other things, let programs perform basic graphics operations like point and line drawing, and display images on the CM graphic display system or on a workstation running the X Window System.
- *Scientific Software.* The Connection Machine Scientific Software Library (CMSSL) provides routines for performing data parallel versions

of standard mathematical operations such as matrix multiply and Fast Fourier Transform.

These libraries, and others, are listed in Chapter 4 of this manual for UNIX front ends and in Chapter 8 for Symbolics front ends; they are also described in detail in separate volumes of Connection Machine documentation.

1.4 Using the Connection Machine System

The Connection Machine system provides a number of user-level commands on UNIX front ends that let you perform various useful functions. You execute these commands from the front end, just as you would any operating system command. (Versions of some of these commands are available as Lisp functions for execution within a Lisp environment.) For example, these commands let you:

- Attach to one or more sections of a parallel processing unit (the **cmattach** command) to execute a data parallel program.
- Submit a program to a batch queue for execution on the CM (**qsub**).
- Find out the status of the CM (**cmfinger**).
- Reset the CM hardware and clear processors' memory (**cmcoldboot**).

Chapter 2 and Chapter 3 discuss these and other CM commands. Chapter 8 discusses the Lisp function equivalents available in the Lisp environment.

1.4.1 CMFS Commands

Files in the CM I/O system exist in a Connection Machine file system (CMFS), which is similar to a UNIX file system. Separate Connection Machine file systems can exist on DataVaults, on VME computers, and even on a front end, where the system is logically independent of the front end's own file system. There are user-level commands available to perform various functions on the files; most of these commands are analogous to standard UNIX commands.

For example, these commands let you:

- Copy a file within the CM file system (**cmcp** or **dvcp**).
- Copy a file from a UNIX file system to a CM file system (**copytodv**).

- Remove a file (`cmrm`).
- List the contents of a directory (`cmls`).

Chapter 7 describes the CM file system and related user-level commands.

There are also Lisp function equivalents of most CMFS operations available for execution within a Lisp environment; see Chapter 8.

Part II
Using the CM Operating System

Chapter 2

Executing a Program on a CM System

This chapter describes how to execute a data parallel program on a Connection Machine system. In addition to the methods described here, you can also include routines in your program that cause it to run on the CM automatically when you execute it; these routines are discussed in Chapter 5.

For information on executing a Lisp/Paris or *Lisp program, see Part V.

Of course, we haven't yet explained how to *write* a data parallel program. For basic information on this topic, see Part III of this guide. For complete information, see the Connection Machine documentation for the individual languages. Your CM system also contains numerous sample programs, which you can compile and execute; see your system administrator for the location of these programs.

If you simply can't wait to learn data parallel programming before using the CM system, we provide a trivial sample program in the first section of this chapter, followed by instructions on how to compile it. You can use this program to get a taste for how the CM system works.

NOTE: This program is written in CM Fortran, which may not be available at your site. Check with your system administrator.

2.1 A Simple Program

The program shown below is written in CM Fortran.

The program sets up three arrays of five elements each. The elements of array A are assigned the values 1, 2, 3, 4, 5; the elements of array B are each assigned the value 2. The program then squares each of these values, adds each element of A to the corresponding element of B, and puts the results in array C. It then prints the results. (This, of course, is not a typical data parallel program.)

```
      PROGRAM SIMPLE

      INTEGER A, B, C, N
      PARAMETER ( N=5 )
      DIMENSION A(N), B(N), C(N)

      DATA A / 1,2,3,4,5 /
      B = 2

      C = A**2 + B**2

      PRINT *, 'Array C contains:'
      PRINT *, C

      END
```

Type this program in a file on the front end as you normally would; call the file **simple.fcm**. (Remember that in Fortran each program statement must begin in column 7.)

To compile the program, issue the following command at your UNIX prompt (which is represented as a percent sign in this guide):

```
% cmf simple.fcm -o simple
```

You now have a CM Fortran program called **simple** that is ready for execution on the CM.

2.2 Overview of Program Execution on a CM

To execute a program on a CM, you must gain access to some of its processors. We call this *attaching* to the CM. As we described in Chapter 1, a front end connects to a CM parallel processing unit via a FEBI (front-end bus interface). A FEBI can be logically attached to one or more sequencers on the CM; a sequencer controls groups of processors within the CM.

There are two basic methods you can use to attach to a CM: *direct access* and *batch access.*

- For direct access, simply execute the program as you normally would; if a FEBI and a sequencer are available, the program attaches and runs. Or, you can issue the **`cmattach`** command to explicitly attach to the CM. Depending on how you issue the command, your program is executed immediately (if a FEBI and a sequencer are available) 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 batch queue, which is associated with a CM, or to a pipe queue, which is associated with a group of batch queues; the pipe queue then sends it to one of the batch queues. Your program attaches to the CM and is executed when it reaches the head of the batch queue.

In both cases, access to the CM can be either *exclusive* or *timeshared,* depending on how your system administrator has configured the system. With exclusive access, only one user can be attached to a FEBI 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. Exclusive access lets your program run faster once you are attached to a CM, but timeshared access makes it easier to attach. Neither affects the way you compile or execute your program.

The choice between direct access and batch access depends once again on how your system administrator has configured the CM system. The system administrator determines whether batch access is available and, if so, how many queues there are and when the jobs in these queues are submitted for execution. There may be restrictions as to when you can obtain direct access to the CM. Thus, while direct access appears to be a faster way to execute your program, batch access may in fact be easier and surer.

In general, direct access (especially from a subshell) is preferable when you are developing your program, since it lets you debug your program interactively on the CM.

NOTE: Your system administrator may have restricted access to the CM to certain users or groups of users. If you are unable to run a program on the CM, check with your system administrator to make sure you are on the access list.

2.3 Obtaining Direct Access to the CM

2.3.1 Overview

The most straightforward method of attaching to a CM is simply to execute your data parallel program from a front end connected to a CM. If resources are available, the program attaches, runs, and then detaches. This is referred to as *auto-attaching*. NOTE: Your system administrator can disable auto-attaching; check to make sure that it is enabled before trying to use it.

You can also obtain direct access to a FEBI and one or more sequencers of a CM by issuing the **cmattach** command from your UNIX prompt on a front end that is connected to a CM.

There are two ways of issuing **cmattach**:

- If you issue **cmattach** with the name of an executable program as an argument, you are attached to the CM (if a FEBI and a sequencer are available) and the program is executed. You are then automatically detached from the CM. The advantage of this method over simply executing the program is that you can include options to **cmattach** that specify the kind of CM resources you want.
- If you issue **cmattach** without specifying the name of a program as an argument, you are attached to the CM (if a FEBI and a sequencer are available) and placed in an interactive subshell, from which you can execute the program and issue other UNIX commands. The CM processors remain attached until you specifically detach them. This allows you to debug and recompile your program, for example, without having to reattach to the CM.

Both versions of **cmattach** have options that let you specify such things as:

- How many physical processors you want
- Whether **cmattach** is to wait if no processors are currently available
- The CM to which you want to attach (if your front end is connected to more than one CM)

In addition, you can issue **cmattach** in the UNIX background or from a remote machine (via the **rsh** command) just as you would any other UNIX command.

Finally, you can include a routine in your program to do the attaching. See Chapter 5 for a discussion of attaching and detaching from within a program.

NOTE: See Section 2.3.6 on page 28 for a discussion of **cmattach** and timesharing.

2.3.2 Executing the Program

If you simply specify the name of the executable program at the UNIX prompt of a front end connected to a CM, the program runs on the CM, provided that resources are available (and that your system administrator has not disabled the auto-attaching feature). If a FEBI and a sequencer are available, the program attaches to them and executes. If multiple resources are available, it attaches to the highest-numbered sequencer that is free on the CM connected to the lowest-numbered FEBI that is also free.

The output would look like this for the program **simple**:

```
Attaching to NAME, a CM2 on interface 0
cold booting... done.
Attached to 8192 processors on sequencer 0,
microcode version 6104
Paris safety is off.
Array C contains:
         5       8        13        20        29
FORTRAN STOP
Detaching... done.
```

NOTE: The output when the sequencer is running under timesharing is slightly different. See Section 2.3.6 on page 28.

Let's look in detail at this output.

```
Attaching to NAME, a CM-2 on interface 0
```

tells you the name and type (CM-2 or CM-200) of the CM system to which you are attaching, and the front-end bus interface from which you are attaching.

```
cold booting... done.
```

tells you that the processors to which you are attaching have *cold booted.* A cold boot resets the portion of the CM to which you are attaching by clearing the memory of the processors and performing other tasks. A cold boot is automatically performed when a program attaches to a CM.

```
Attached to 8192 processors on sequencer 0,
```

tells you the number of the sequencer to which you are attached, and how many processors are associated with this sequencer.

```
microcode version 6104
```

specifies which version of the CM microcode is running on this sequencer. Knowing which version of the microcode is running is important if your program is going to run under timesharing, since your program must be compiled with the same microcode that timesharing uses.

```
Paris safety is off.
```

tells you that Paris-level safety checking is not being performed on your program. See Chapter 6 for a description of safety checking.

The next lines of the output come from the program **simple**. The final line:

```
Detaching... done.
```

tells you that you are being detached from the CM. You are returned to your UNIX prompt.

As we mentioned above, this method of executing a program on the CM is simple and convenient, but it lacks flexibility; you have no choice as to the CM, sequencer, or interface on which your program is to run. To gain this flexibility, you must:

- Use the **cmattach** command, as described below; or
- Include the routine **CM_attach_to** in your program, as discussed in Chapter 5.

2.3.3 Issuing cmattach with the Name of a Program

A second method of executing a program on a CM system is to issue the **cmattach** command with the name of the executable program as an argument. If the program itself takes arguments, you can specify them on the command line as well.

This command line executes the program **simple**:

```
% cmattach simple
```

This obtains exactly the same result as simply typing the name of the program, as described in Section 2.3.2, provided that resources are available.

The advantage of using **cmattach** is that it provides options that let you specify the CM resource to which you want to attach; see Section 2.3.5.

2.3.4 Using cmattach to Obtain an Interactive Subshell

If you issue **cmattach** without the name of an executable program, the following happens:

- If a FEBI 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, and issue other CM commands, or issue any standard UNIX command

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

The example below shows how you would execute the program **simple** in this way. (Text in bold shows what you type; text in normal typeface shows output from the system.

```
% cmattach
Attaching to NAME, a CM-2 on interface 0
cold booting... done.
Attached to 8192 processors on sequencer 0,
microcode version 6104
Paris safety is off.

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

% simple
Array C contains:
        5       8        13        20        29
FORTRAN STOP
% exit
Detaching... done.
%
```

This method of issuing **cmattach** is most useful when you are developing a program. You can run the program on the CM, debug it, recompile it, and run it again; the CM stays attached until you explicitly detach it or exit from the subshell.

Your system administrator can specify the amount of time users can be idle in a **`cmattach`** subshell. If you exceed this limit, you are automatically detached from the CM.

If your program contains a call to **`CM_attach`** or a related routine, you can still run it in a **`cmattach`** subshell; the program is executed on the sequencer(s) to which you are attached in the subshell. See Chapter 5.

If you forget that you are in a **`cmattach`** subshell and you issue **`cmattach`** with a program name, you are subsequently detached from the CM, but you stay in the subshell.

See Section 2.3.5 for the options you can specify when issuing **`cmattach`**.

2.3.5 Options for cmattach

The **`cmattach`** command provides numerous options that let you control how you attach to a CM; see Table 1. This section describes the most commonly used options. See the **`cmattach`** man page in Appendix F for complete information on all options. See Section 2.3.6 on page 28 for a discussion of these options when timesharing is in effect.

Table 1. Options for the **`cmattach`** command

Option	**Meaning**
`-c` *CMname*	Attach to the specified CM.
`-cm`*n*	Attach to the specified CM model.
`-e`	Obtain exclusive access only.
`-g` *length, length ...*	Create a virtual processor geometry with the specified dimensions upon attaching.
`-h`	Print a help message.
`-i` *interface*	Attach to the specified FEBI.
`-n`	Do not cold boot.
`-p` *nprocs*	Attach to the specified number of processors.
`-q`	Do not display informational messages.
`-s` *sequencer*	Attach to the specified sequencer(s).
`-t`	Obtain timeshared access only.
`-u` *nnnn*	Load the specified version of the microcode.
`-w`	Wait for resources.

NOTE: See Appendix A for options used in back-compatibility mode.

The options are the same whether or not you specify a program name on the command line.

Waiting for Resources: The –w Option

As we mentioned above, you must gain access to both a FEBI and a sequencer to execute a program on the CM. If one or the other is not available, you cannot attach to the CM. Specify the **-w** option if you are willing to wait for the required resources to become available. For example:

```
% cmattach -w simple
```

Access to CM resources via the **-w** option is granted to the oldest request that fits the available resource. The more general your request, the more likely it is to be satisfied quickly. If you request a specific resource (for example, an individual sequencer or individual interface) you may not get it until after more general requests are satisfied. If no resources are available, you receive this message:

```
cmattach: Waiting for CM resources to become
available.
```

You can use this option to execute a program in the UNIX background. For example, if you are using the C shell, you could execute **simple** as follows:

```
% cmattach -w simple >& output &
```

In this example, program output and any error messages are redirected to the file `output`. It is important to redirect *both* standard output and standard error; if both streams are not redirected, the program could be suspended waiting to write to the terminal. A useful addition to this command line is the **-q** option, which suppresses screen display of informational messages from `cmattach`. In addition, if your program requires input, you should redirect the standard input.

Specifying a Sequencer: The –S Option

With no options specified, `cmattach` attaches to the highest-numbered sequencer that is free on the CM connected to the lowest-numbered interface that is also free. Use the `-S` option to specify that you want to attach to a particular sequencer, or to more than one sequencer. You might ask for a particular sequencer if, for example, it has a framebuffer connected to it, and you want to use the CM's graphic display system. You might ask for more than one sequencer if your program has a large data set, and you want it to run on more processors than are provided by a single sequencer.

To specify that you want to attach only to sequencer 1 and execute the program **simple**, your command line would look like this:

```
% cmattach -S1 simple
```

To specify that you want to attach to sequencers 0 and 1, your command line would look like this:

```
% cmattach -S0-1 simple
```

You can specify an individual sequencer, or one of the following combinations of sequencers: 0-1, 2-3, or 0-3.

Specifying the Kind of Access You Want: The –e and –t Options

Use the **-e** option to specify that you require exclusive access to the CM (or part of it). If you use this option, the system will not attach you to a timeshared sequencer.

Use the **-t** option to specify that you require timeshared access to the CM, as opposed to exclusive access.

Specifying an Interface: The –i Option

Use the **-i** option to specify the number of the FEBI to which you want to attach. This option has an effect only if your front end has more than one FEBI from which you can reach the CM. Use the **cmfinger** command to obtain information about interface numbers; see Chapter 3.

Specifying a CM: The –C Option

If you are lucky enough to have more than one CM available from your front end, you can use the **-C** option, followed by the name of a CM, to choose the CM to which you want to attach. For example,

```
% cmattach -w -C ruby
```

attaches you to the first available sequencer on the CM named Ruby; the command is to wait if no resources are available. (Note that case does not matter.)

Specifying the CM Model: The -cm Option

If you have more than one CM available, you may also have more than one model of CM. Use the `-cm` option to specify the model to which you want to attach. The choices are 2 and 200. Use `-cm2` if you want to attach to a CM-2 series machine; use `-cm200` if you want to attach to a CM-200 series machine.

Specifying a Geometry: The -g Option

Virtual processors (VPs) on the CM are arranged in *VP sets*, which have a *geometry*. The geometry specifies the "shape" of a VP set. This shape affects the way the processors communicate when a program is running. Choosing an appropriate geometry can increase the efficiency of a program. You can specify the geometry of a VP set in Paris programs; in programs written in high-level languages, the compiler does this for you. You can also specify an initial geometry (and the size of the initial VP set) for a program by using the `-g` option to `cmattach`. Specify the values for each axis of the geometry, separated by commas, with no spaces in between. Each value must be a power of 2, and the total number of processors must be an integer multiple of the number of physical processors to which you are attached. For example,

```
% cmattach -g 64,256
```

creates a VP set of 16,384 processors, arranged in a 64-by-256 geometry.

If you do not use the `-g` option to specify a geometry, you get a default two-dimensional geometry that depends on the number of processors to which you are attached. These default geometries are listed in Table 2.

Table 2. Default geometries

Number of Processors	Geometry
4K	64-by-64
8K	64-by-128
16K	128-by-128
32K	128-by-256
64K	128-by-512

This option is not useful for running C* or CM Fortran programs.

For more information on VP sets and geometries, consult the *Paris Reference Manual* and *Introduction to Programming in C/Paris.*

Specifying the Microcode Version: The –u Option

Use the **-u** option, followed by a four-digit number, to specify which version of the CM microcode you want the CM to use. If you omit this option, you get the latest version of the microcode. Typically, you would use this option if you had compiled and linked your program using an older version of the microcode, and you didn't want to bother recompiling. If you attempted to run your program without recompiling, you would receive a warning about incompatible microcode versions.

NOTE: Do not use this option if the sequencer to which you are attaching is running under timesharing. In that case, your program must use the current version of the microcode. See "Timesharing and Microcode Version" on page 30. If you do use this this option, and timesharing is running a different version of microcode from the one you specified, **cmattach** exits without attaching you, and it prints an error message.

2.3.6 Obtaining Direct Access under Timesharing

Your CM system may be set up so that one or more sequencers allow timeshared access, under which multiple processes can run on a sequencer at the same time. To find out if timesharing is operational on a sequencer before you attach to it, issue the **cmfinger** command, as described in Chapter 3; if timesharing is operational, **cmfinger** will display "`{CM}*`" in the "COMMAND" field for that sequencer.

You can attach to a timeshared sequencer just as you would to a sequencer that is not running under timesharing. If you attach to a timeshared sequencer, you receive a response that looks like this:

```
{CM}* Timesharing on FOO
Attached to 8192 processors on sequencer 0,
microcode version 6104
Paris safety is off.

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

There are a few restrictions in running processes under timesharing. They are discussed in the sections below. See Appendix A for a discussion of timesharing and back-compatibility mode.

Performance under Timesharing

With a Sun front end, you can in general expect your program to execute at the same speed under both timesharing and exclusive mode (except, of course, for the slowdown related to being swapped out while other processes execute). With a VAX front end, execution can be significantly slower under timesharing. One way to reduce the penalty is to minimize the number of times your program requires the CM to synchronize with the front end, since the mechanism that timesharing uses for this with a VAX is much slower than the mechanism used when a program has exclusive access to the CM. Appendix C lists Paris instructions that cause the CM to synchronize with the front end.

If performance is unacceptable under timesharing, use the **-e** option to **cmattach** to obtain exclusive access to the CM, or submit the program for execution in a batch queue running in exclusive mode. See Section 2.4.4 on page 42 to learn how to determine if a batch queue is running in exclusive mode.

Maximum Number of Processes

There is a maximum number of processes that can use timesharing at the same time; the system administrator sets this number. If you have obtained a **cmattach** subshell, and you receive a message with this format when you try to execute your program:

```
prog-name:waiting for ts-daemon to have a process-
slot...
```

the limit has been reached, and your program cannot run until a process exits. A period is printed every thirty seconds until a slot becomes available and the program runs.

If you don't want this behavior, set the environment variable **CM_WAIT** to **false**. If you do this, you simply receive the message saying that no process-slots are available. You can then try running the program later, or on another CM resource.

It is possible that you will not even be able to obtain a **cmattach** subshell when trying to run a program under timesharing. If this happens, you receive the following message after issuing **cmattach**:

```
Error accessing "/dev/cm" passed in CMDEVICE from
environment: No more processes
Please cmattach again to run this program.
Attach febi failed
```

In this case, you can use the **-w** option to **cmattach** to wait for a **cmattach** subshell. It is likely, however, that the sequencer is extremely busy under these circumstances, and you may be better off trying a different sequencer.

Timesharing and Memory Size

Your system administrator may have restricted the size of processes that can run under timesharing. Even if there is no restriction, timesharing requires about 6 Kbits of overhead in memory; the result is that programs with large memory requirements that ran under exclusive access to the CM may be unable to run under timesharing.

If you receive an error message like this after executing your program:

```
Error: You have run out of CM memory while trying
to allocate 32768 bits of stack.
```

your program required too much memory to execute under timesharing. Try executing it under exclusive access. Or, ask your system administrator to change the timesharing configuration so that programs with larger memory requirements are accepted. (As mentioned above, this may not solve the problem, if it is the timesharing overhead that is preventing your program from running.)

Timesharing and Microcode Version

As mentioned above, you cannot use the **-u** option when attaching to a timeshared sequencer. Your program must be compiled with the same version of the microcode that timesharing itself uses—the microcode version reported when you attach to the sequencer without using the **-u** option. In the example shown at the beginning of this section, this is microcode version 6104.

Timesharing and the Framebuffer

Although timesharing allows multiple programs to use a sequencer at the same time, only one program can have access to the framebuffer module and high-resolution color monitor that may be connected to the sequencer. If your program tries to use the framebuffer when it is already in use, you will receive an error message.

Timesharing and the DataVault

Unlike the framebuffer, the DataVault allows multiple processes to gain access to it. Therefore, there are no restrictions on using the DataVault under timesharing.

Timesharing Signals

Your process can receive one of the following signals when running under timesharing:

- When the timesharing daemon exits because of an administrative request, your process will be detached from the CM and it will receive a SIGURG signal, as happens whenever a process is detached from the CM.

- The timesharing daemon sends out a 20-second warning in the form of a SIGTERM signal when it has been asked to shut down, allowing your process to shut itself down. The timesharing daemon also sends out a SIGTERM signal when it exits—including when it crashes (in this case, there is little you can do to recover your process later).

- When the timesharing daemon detects that your process has corrupted the memory of another process, it will send your process a SIGILL signal (a message is also printed on the controlling terminal of the process).

  SIGILL will also sometimes be sent when your process sends bad data to the sequencer, effectively crashing the sequencer microcode (the timesharing system can't recover your process in this case, but other users won't be affected).

- When the timesharing daemon detects that another process has corrupted the memory of your process, it sends a SIGLOST signal to your process, as well as sending a message to your controlling terminal.

- If the timesharing daemon encounters a swap error when swapping your process in, it sends the process a SIGKILL signal.

2.3.7 Direct Access and Batch Queues

Your CM may provide one or more batch queues to allow batch execution of data parallel programs. When the queue has a job to run, it may automatically detach the process currently attached to the sequencer with which the queue is associated; this depends on how the queue is configured. Therefore, it is a good idea to become familiar with the batch queues on your system, so that you can avoid running your programs on a sequencer where one of these queues is active. To do this, issue the `qstat` command, as described in Section 2.4.4 on page 42. In particular, see the description of the `-x` option and enforce mode.

2.4 Obtaining Batch Access to the CM

2.4.1 Overview of the CM Batch System

In a batch system, you submit one or more programs as a request to a queue. The batch system in turn submits the requests in the queue for execution. Your request is generally executed when it reaches the head of its queue.

The CM batch system is based on NQS (Network Queueing System), a standard batch system. NQS can also be used for batch submissions to computers other than the CM. In this guide, however, we focus only on using NQS to submit requests for execution on the Connection Machine system.

The CM system administrator is in charge of configuring queues to meet the needs of your site. You may not have any queues, or you may have several. You may have only *batch queues*, which submit requests directly for execution, or you may in addition have *pipe queues*, which pass requests along to batch queues. Pipe queues are useful because they can be associated with several different batch queues; if one is unavailable, the pipe queue can try the next, until it finds one that will accept the request. You don't have to worry about finding the available queue yourself.

Here are some of the characteristics of batch queues that a system administrator can configure:

- *What resources the queue uses.* A batch queue can attach to a particular sequencer, for example, leaving the rest of the CM available for direct access.

- *When the queue submits its requests for execution.* A batch queue can operate continuously, or it can operate only at specified times—for example, from midnight to six in the morning.

- *How the queue interacts with direct-access users.* For example, requests submitted to the queue can have exclusive access to a sequencer, or they can compete with other users for access to the sequencer.

To submit a request for execution via either a batch queue or a pipe queue, you either:

- Issue the **qsub** command, using as an argument the name of a script file that contains the name of the program or programs to be run; or
- Submit the program or programs to **qsub** from the standard input.

See Section 2.4.2, below.

To obtain information about a queue, or about the status of a request in a queue, issue the `qstat` argument. See Section 2.4.4 on page 42.

Table 3 lists the user commands for the NQS batch system.

Table 3. User commands for the NQS batch system

Command	**Meaning**
`qdel`	Delete or signal one or more batch requests.
`qlimit`	Display the supported limits on batch queues.
`qstat`	Display the status of queues and batch requests.
`qsub`	Submit a batch request.

2.4.2 Submitting a Batch Request: The qsub Command

Use the **qsub** command to submit a program for execution on the CM via either a batch queue or a pipe queue.

NOTE: The **qsub** command has many options associated with it; in this guide, we discuss only some of the more important. See the man page for **qsub** in Appendix F for a complete discussion of all its options. Table 4 summarizes the **qsub** flags.

Table 4. Options for the **qsub** command

Option	Meaning
-a *time*	Do not run the request before the specified time and/or date.
-e *filename*	Direct the standard error output to the specified file.
-eo	Direct the standard error output to the batch request output file.
-ke	Keep the standard error output on the execution machine.
-ko	Keep the standard output on the execution machine.
-lc *size*	Set the per-process corefile size limit.
-ld *size*	Set the per-process data-segment size limits.
-lf *size*	Set the per-process permanent-file size limits.
-ln *value*	Set the per-process nice execution value limit.
-ls *size*	Set the per-process stack-segment size limits.
-lt *time*	Set the per-process CPU time limits.
-lw *size*	Set the per-process working set limit.
-mb	Send mail when the request begins execution.
-me	Send mail when the request ends execution.
-mu *username*	Send mail about the request to the specified user.
-nr	Declare that the request is not restartable.
-o *filename*	Send the output of the request to the specified file.
-p *priority*	Set the priority for the request in the batch queue.
-q *queue*	Send the request to the specified batch queue.
-r *name*	Assign the specified request name to the request.
-re	Remotely access the standard error output file.
-ro	Remotely access the standard output file.
-s *shell*	Use the specified shell to interpret the request.
-x	Export all environment variables with the request.
-z	Submit the request silently.

The Basics

To execute the program **simple** via the queue **cmq1**, put the program's name in a *script file*. A script file is simply a UNIX file that contains commands to be executed. For example, you could create a file called **simple_script** that contains just the word `simple`. You could then submit this to **cmq1** as follows:

```
% qsub -q cmq1 simple_script
```

The **-q** flag specifies the name of the queue to which you are submitting the request.

The system displays a response like this:

```
Request 276.barney.acme.com submitted to queue: cmq1
```

The number 276 is a *sequence number* assigned to this request by NQS. **`276.barney.acme.com`** is the *request-id* for this request.

When **`simple`** is finally executed, its output is placed in a file; error messages are placed in another file.

Submitting a batch request has these basic elements:

- Specifying the queue to which the request is being submitted
- Specifying the request to be run
- Specifying options that affect the way the request is to be run

You can embed **`qsub`** options at the beginning of the script file, along with the name of the executable program and other commands. See "Specifying the Queue," below, for an example. NQS looks at options in the script file only if they are not specified on the **`qsub`** command line; this lets you override a script file option by specifying a different setting for the option on the **`qsub`** command line.

Specifying the Queue

There are several different methods of specifying the queue to which you want to submit your request. You can find out the names and characteristics of available queues by issuing the **`qstat`** command; see Section 2.4.4 on page 42.

You can use the following methods to specify a batch queue:

- Use the **`-q`** option on the **`qsub`** command line, as shown in the example above. NQS submits the request to the queue you specify.
- Embed the **`-q`** option in a script file that you name on the **`qsub`** command line. All **`qsub`** options must appear at the beginning of the script file, and must begin with a pound sign (#) followed by an "at" sign (@) and a dollar sign ($). The option must begin immediately after the dollar sign—no white space is allowed. Comments must begin with a pound sign. For example, the following script file sends the program **`simple`** to queue **`cmq1`** for execution:

```
#
# Example of a batch script file
#
# @$-q cmq1 # Send request to cmq1 unless
#           # overridden on command line.
#
simple
```

If you named this script file **simple_script**, you could execute the program by issuing the following command:

```
% qsub simple_script
```

- Set the environment variable **QSUB_QUEUE** to the name of the queue to which you want the request submitted. You would typically do this to set up a default queue for all requests, which you could override for a specific request by using the **-q** option. If you use the C shell, you could put the following command in your **.cshrc** file to set the default queue to **cmq2**:

```
setenv QSUB_QUEUE cmq2
```

If you don't use any of these methods for specifying a queue, the request is submitted to the default batch queue for the system, if your system administrator has defined one.

Specifying a Request from a Script File

As we have already shown, you can execute a program by including its name in a script file. A script file is the batch equivalent of a **cmattach** subshell (see Section 2.3.4 on page 23). For example, you might want to execute the program **simple** twice, once with run-time safety off (the default), and once with safety on. You use the CM operating system command **cmsetsafety** to turn safety on; see Chapter 6. A script file could then contain the following commands:

```
simple
cmsetsafety on
simple
```

When the request is run, the first execution of **simple** is with safety off; the second is with safety on.

You can use UNIX commands and other CM operating system commands as well. You shouldn't, however, explicitly attach to or detach from the CM, since typically the queue takes care of that for you. Note that this means you can't specify options to **cmattach**—for example, the **-b** option to execute in back-compatibility mode; see Appendix A. If this is a problem, your system administrator can set up a queue that does not automatically attach to a CM; in that case, you *must* explicitly attach to and detach from the CM. Check with your system administrator to find out if such a queue exists.

Typically, NQS interprets the commands in a script file exactly as if you had typed them at your UNIX prompt. It may, however, use a different shell to interpret the commands, depending on how your system administrator has configured NQS. See "Choosing a Shell" on page 40.

Specifying a Request from Standard Input

Instead of using a script file, you can simply enter the request from standard input—that is, directly after the **qsub** command line. Put each command or program name on a separate line, and type the Ctrl-D key combination at the end to signal that there is no more input. For example:

```
% qsub -q cmq1
  simple
  cmsetsafety on
  simple
  Ctrl-D
```

If you are executing a shell under Emacs or Gmacs, type Ctrl-C Ctrl-D.

The Output from a Request

NQS places the output from a batch request in a file, which is by default placed in your current working directory. You can control the name and location of this file. The default filename consists of the first seven characters of the script name, followed by **.o**, followed by the sequence number of the request. Thus, in our example in "The Basics" on page 34, NQS would put the output in the file **simple_.o276** in your current working directory. Messages to standard error go into a file with **.e** in the name instead of **.o**.

If you submit the request from standard input, the default output and error files would begin with **STDIN.o** and **STDIN.e**, followed by the sequence number.

To specify a different output filename, use the **-o** option, followed by a pathname, on the **qsub** command line or in a script file. NQS writes output to the pathname you specify. For example,

```
% qsub -o /requests/simple.out simple
```

causes the output of **simple** to be written to **/requests/simple.out**.

Similarly, use the **-e** option to specify a different pathname for standard error output.

Another way to change the name of the output file is to use the **-r** option, followed by a *request-name* of up to 15 characters. This request-name identifies the request when you issue the **qstat** command to check the status of the batch queue; if you don't specify a request-name, NQS uses the name of the script file (or **STDIN**) instead. If you do specify a request-name, NQS substitutes it for the name of the script file (or **STDIN**) in the name of the output file.

Here is sample output for our **simple_script** batch request:

```
Cold boot...
Array C contains:
         5       8         13        20        29
FORTRAN STOP
logout
```

In running the batch job, NQS runs a script as if logged in as you; start-up files like **.cshrc** and **.login** are executed. This means that you may see various messages along with the output. In particular, you will probably see the following message:

```
Warning: no access to tty; thus no job control in
this shell...
```

This comes from the shell, warning you that there is no terminal associated with this job. You can ignore this message.

Here is the standard error output:

```
Attached to 8192 processors on sequencer 0
microcode version 6104
Paris safety is off.
```

Note that while the job is running, NQS considers your home directory to be your current working directory (because it runs the job as a newly logged-in process). Thus, if the process dumps core, the corefile is placed in your home directory, rather than the directory from which you submitted the job.

Setting Limits on a Request

The **qsub** command has many options you can specify to set the limits on the amount of front-end resources a batch request can use. The batch queue has its own set of limits. You can find them out by issuing the **qstat** command; see Section 2.4.4 on page 42. You may want to set lower limits to obtain more favorable scheduling for your request, or to avoid running up accounting charges if, for example, your program goes into an infinite loop.

For example, use the **-lt** option to set a limit on the amount of front-end CPU time an individual program within a batch request can use. The following command sets a limit of 120 seconds of CPU time for the program **simple**:

```
% qsub -lt 120 simple
```

SunOS and ULTRIX do not support all of the limit options that **qsub** lets you specify. If you specify an unsupported option, NQS ignores it. To find out which options your front end supports, issue the **qlimit** command. For example, if your front end is named Barney, issue **qlimit** as follows:

```
% qlimit barney
```

The response might look like this:

```
Core file size limit (-lc)
Data segment size limit (-ld)
Per-process permanent file size limit (-lf)
Nice value (-ln)
Stack segment size limit (-ls)
Per-process cpu time limit (-lt)
Working set limit (-lw)

Shell strategy = FREE
```

These are the limits you can set for this front end. The "shell strategy" in this response refers to the default way in which NQS chooses a shell to interpret commands in a script file. See "Choosing a Shell," below.

Choosing a Shell

As we mentioned above, your system administrator can specify how NQS is to interpret commands in batch script files. This is called the *shell strategy*; you can find out the default shell strategy via the `qlimit` command. The possible shell strategies are:

- *Free.* Your login shell determines the appropriate shell to be used to execute the commands in your script file, and executes that shell. This typically means that NQS uses the shell that would have been used if you had issued the commands in the script file interactively. For example, if your script file begins with the line

```
#! /bin/csh
```

your login shell would execute a C shell for the script file.

- *Login.* NQS uses your login shell to execute the commands in your script file, regardless of the contents of your file.

- *Fixed.* NQS uses a specified shell to execute the commands, regardless of the contents of your script file. Use `qlimit` to find out the name of this shell.

You can override this strategy by using the **-s** option of the **qsub** command. For example,

```
% qsub -s /bin/csh simple_script
```

specifies that the C shell is to be used to interpret the commands in the script file **simple_script**.

Setting a Priority for a Batch Request

To set a priority for your batch request in its queue, use the **qsub** option **-p**, followed by an integer from 0 to 63, inclusive; 63 is the highest priority, and 0 is the lowest priority. This priority determines the request's position in the queue. The request is placed in front of all requests with lower priority, and behind all requests with higher or the same priority.

If you don't specify a priority, the request is assigned a default priority, as set by the system administrator. Use the **qstat** command to determine the default priority for a queue; see Section 2.4.4 on page 42.

NOTE: NQS does not necessarily run requests in the order in which they appear in a batch queue. It can take requests out of order to use resources efficiently. Generally, however, requests at the beginning of the queue are run before requests that appear later in the queue.

Receiving Mail about a Batch Request

Use the **qsub** options **-mb** and **-me** to specify that NQS is to send you mail about your batch request. Specify **-mb** to get mail when the request begins execution; specify **-me** to get mail when the request ends execution.

To obtain more information about the status of a batch request, use the **qstat** command; see Section 2.4.4 on page 42.

Wall-clock Limits for Queues

Your system administrator can set a wall-clock limit for a queue. No request can run longer than this limit; once the limit is reached for a request, NQS sends a SIGKILL signal to all processes that are part of the request. (The limit does not include time spent waiting in the queue, but it *does* include time spent swapped out under timesharing.) Use the **qstat** command with the **-x** option to determine the wall-clock limit, if any, for a queue; see Section 2.4.4. You cannot change this wall-clock limit for an individual batch request in the queue.

When a process is killed, its output (.o) file will probably be empty; NQS will send mail to the submitter indicating that the request was aborted.

A queue can also have a *warning limit*, which is less than the wall-clock limit; if a request reaches the warning limit, NQS sends its processes a SIGXCPU signal. If the request's shell script and all its processes contains handlers for SIGXCPU, the request can catch this signal and carry out an orderly shutdown before the wall-clock limit is reached and it is killed. (NOTE: Currently a request has only 60 seconds after the warning limit is reached before it is killed, no matter what the wall-clock limit is.)

Timesharing and Batch Requests

The sequencer (or sequencers) with which a batch queue is associated may operate under timesharing, depending on how the system is configured. To find out if a sequencer is operating under timesharing, use the **cmfinger** command, as discussed in Chapter 3. If a batch queue is associated with a timeshared sequencer, more than one request can run at the same time.

In general, you don't have to be aware of whether a batch queue is associated with a timeshared sequencer. However, see Section 2.3.6 on page 28 for some restrictions on processes running under timesharing.

2.4.3 Deleting a Batch Request: The qdel Command

Issue the `qdel` command to delete a request from a queue. As an argument, specify the request-id that was displayed when you submitted the request. (You can also obtain the request-id by issuing the `qstat` command; see Section 2.4.4, below.) For example,

```
% qsub -q cmq1 simple_script
Request 276.barney.acme.com submitted to queue:
cmq1.
% qdel 276
```

submits a request, then deletes it. (You don't need to specify the hostname if you are issuing the command from the local host.)

This form of the `qdel` command does not delete a request that is actually running. To do this, use the **-k** option. This option sends a SIGKILL signal to the specified request, causing it to exit and be deleted from the queue. If the request contains more than one process, all are signalled.

To send a signal other than SIGKILL to a running request, specify its number instead of **k** (see the discussion of **sigvec** in your UNIX documentation for signal numbers). For example, to send a SIGTERM signal to a running request with request-id 276, issue this command:

```
% qdel -15 276
```

2.4.4 Obtaining Information: The qstat Command

Use the `qstat` command to obtain information about a queue and the batch requests in the queue.

For example, to find out the status of all your requests on any queue, simply issue this command:

```
% qstat
```

The response might look like this:

```
cmq1@barney.acme.com; type=CM_BATCH; CM=(carvel:0);
[ENABLED, RUNNING]; pri=16

0 exit; 0 run; 0 stage; 1 queued; 0 wait; 0 hold; 0 arrive;

  REQUEST NAME    REQUEST ID    USER  PRI    STATE     PGRP
1:simple_script  276.barney    smith  31    QUEUED    2085
```

Let's take a closer look at the information in this response.

cmq1@barney.acme.com identifies the queue.

type=CM_BATCH specifies the kind of queue.

CM=(carvel:0) identifies the CM and sequencer with which this queue is associated.

[ENABLED, RUNNING] shows the state of the queue. A queue can be *enabled*, *closed*, or *disabled*.

- If a queue is *enabled*, requests can be submitted to it.
- If a queue is *closed*, NQS is not running on the front end. No requests can be submitted to it.
- If a queue is *disabled*, the system administrator has prevented any more requests from being placed in the queue.

A queue can also be *running*, *inactive*, *stopped*, *stopping*, or *shutdown*.

- If it is *running*, one or more requests are currently being executed, and other requests are prevented from running only because they haven't been scheduled.
- If it is *inactive*, no requests are being executed, and requests in the queue are prevented from running only because they haven't been scheduled.
- If it is *stopped*, queued requests are blocked from running, and no requests are currently running. The system administrator can stop a queue. If a queue is stopped, you can still submit requests to it (if it is enabled), but they are just added to the queue until it starts again.
- If it is *stopping*, the queue will be stopped once the current request has run.
- If it is *shutdown*, NQS is not running on the front end.

`pri=16` specifies the interqueue priority, set by the system administrator. If more than one queue is attempting to run a request at the same time, the one with the higher priority goes first.

`0 exit; 0 run; 0 stage; 0 queued; 0 wait; 0 hold; 0 arrive` indicates the number of requests at each stage of the batch cycle. A request can be *arriving*, *holding*, *waiting*, *queued*, *staging*, *routing*, *running*, *departing*, or *exiting*.

- A request is *arriving* if it is being placed on the queue from a remote host.
- A request is *holding* if it is currently prevented from entering any other state because a hold has been placed on it; holds are currently not implemented by NQS.
- A request is *waiting* if it was submitted with the constraint that it not be run before a certain date or time, and that date or time hasn't arrived yet. You submit a request in this way by using the **qsub** option **-a**.
- When a request is *queued*, it is eligible to run.
- If the queue is a pipe queue, a request can be *routing* or *departing* as it passes through the queue.
- A request is *staging* when its input files are being brought on to the front end on which it is to execute.
- A request is *running* when it is actually executing. It is *exiting* when it has completed execution and the required output files are being returned.

For each request in the queue, `qstat` displays the request name, the request-id, the name of the user who submitted the request, its priority, its state, and its process group. If the request name began with a digit, an **R** is added to the beginning of it. The priority in this case is the request's priority within this queue. You specify this priority for a request by using the **qsub** option **-p**. All processes that are part of the same batch request are assigned to the same process group.

Options to qstat

By default, `qstat` displays the status of all your batch requests on all queues. Specify a particular queue if you want to see only the requests on that queue. Use the **-a** (all) option to see the status of all batch requests, not just your own. For example,

```
% qstat -a cmq1
```

displays the status of all batch requests on the queue **cmq1**. Use the **-l** (long) or **-m** (medium) option to obtain more information about individual requests. Use the **-x** (extended) option to obtain more information about the queue; information like that shown below is displayed:

```
Run_limit = 1;
Cumulative system space time = 1585.21 seconds
Cumulative user space time = 671.31 seconds
Unrestricted access
Per-process core file size limit = 1 megabytes <DEFAULT>
Per-process data size limit = 1 megabytes <DEFAULT>
Per-process permanent file size limit = UNLIMITED
Per-process execution nice value = 0 <DEFAULT>
Per-process stack size limit = 1 megabytes <DEFAULT>
Per-process CPU time limit = UNLIMITED
Per-process working set limit = 1 megabytes <DEFAULT>
Per-request wall-clock time limit = 1000 seconds (max), no warning
Connection Machine assigned = RUBY
Sequencer resource assigned = 0
Connection Machine usage description = Queue for CM Ruby
Connection Machine exclusive mode = ON
Connection Machine enforce mode = OFF
Restriction Window start time = Wed Aug 31 14:00:00 EDT 1990
Restriction Window stop time = Wed Aug 31 22:00:00 EDT 1990
Restriction Window MODE = TIMEDATE
Restriction Window send TERMinate signal = ON
```

Comments about some of these items:

- **Run_limit** refers to the maximum number of requests in the queue that are allowed to run at any given time. This should be 1 unless the CM is operating under timesharing.
- **Unrestricted access** means that anyone can use the queue. The system administrator can restrict access to specified users and groups.
- The wall-clock time limit for this queue is 1000 seconds; there is no warning limit.
- **Connection Machine assigned** and **Sequencer resource assigned** specify the sequencer and CM that this queue uses. **Connection Machine usage description** gives information

about this sequencer and CM. These fields are applicable to batch queues only.

- The setting of **`Connection Machine exclusive mode`** (applicable to batch queues only) indicates whether this queue allows other users to attach to its resource while a request is running. If the setting is ON, no other users can attach while a request is running; if timesharing was previously in effect, the queue turns it off while the request is running. If the setting of exclusive mode is OFF, batch queue requests compete with other users for the CM resource.

- The setting of **`Connection Machine enforce mode`** (batch queues only) indicates whether this queue forcibly detaches users attached to its resource when a request in the queue is ready to run. It is typically set to ON only if exclusive mode is also set to ON.

  If the setting is ON, and timesharing is not in effect, the queue detaches the user currently attached when it receives a request. If timesharing is in effect, it detaches all users from the resource and turns off timesharing.

  If the setting of enforce mode is OFF, the queue does not forcibly detach users from the resource. The queue waits—perhaps indefinitely—for currently running processes to finish and for the resource to become available.

- The **`Restriction Window`** indicates the time during which the queue is available to run requests.

Chapter 3

Miscellaneous CM Operating System Commands

This chapter describes several useful CM operating system commands. Issue CM commands from the UNIX prompt on a front end, just as you would any standard UNIX command. You can also execute the command from any other computer in the CM system that has the CM System Software loaded. For example, if your system has a VME I/O host computer, you can issue CM commands from that computer as well. See the man pages on-line or in Appendix F for reference descriptions of these commands.

Versions of some commands are available as routines you can call from within C or Fortran programs; see Chapter 5.

Versions of some commands are also available as Lisp functions for execution from within a Lisp environment. See Part V of this guide.

Table 5 lists the commands discussed in this chapter.

Table 5. Miscellaneous CM operating system commands

Command	**Use**
`cm`	Displays information about a `cmattach` subshell.
`cmcoldboot`	Resets a CM.
`cmdetach`	Detaches a user from a CM.
`cmfinger`	Displays CM interfaces.
`cmlist`	Lists available CMs.
`cmman`	Displays CM and UNIX manual pages.
`cmnice`	Runs a program with low timesharing priority.
`cmps`	Lists processes running under timesharing.
`cmrenice`	Changes the timesharing priority of a running process.
`cmtime`	Times a CM program.

3.1 Obtaining Status Information: The cmfinger Command

Use the **cmfinger** command to find out the current status of CMs connected to your front end. This command prints out a table that shows which front ends are connected to which sequencers of a CM system, who is using the sequencers (and who is waiting for them), what command is being executed, and configuration information about the system.

To find out the status of an individual CM, specify its name on the **cmfinger** command line. To find out the status of all CMs in your system, use **cmfinger** with the **cmlist** command; see Section 3.2 on page 50.

The **cmfinger** command displays information like this:

```
CM        Seqs Size   Front end I/F   User    Idle   Command
-----------------------------------------------------------

FOO         1   8K      wotan      0    karen 0h 06m "cmattach"
FOO        --- ---      epicurus   0    nobody

      cm2 with 1024K memory, 32-bit floating point
      framebuffers on sequencers 0 1 (seq 0 is free)
      CMIOCs on sequencers 0 1 (seq 0 is free)
      1 free seq on FOO -- 0 -- totalling 8K procs

3 processes waiting:
  dm[4146] waiting since 8:09:32 PM;wants i/f 0 ucc(s) 0 or 1.
  sam[4147] waiting since 8:23:36 PM; wants i/f 0 ucc(s) 1
  sophie[4148] waiting since 9:09:39 PM; wants i/f 0 ucc(s)
       0 and 1; or i/f 1 ucc(s) 0 and 1
```

In this case, the CM-2 called Foo has two front-end interfaces: interface (I/F) 0 on Wotan and interface 0 on Epicurus. The user named Karen is attached to sequencer 1 of Foo via Wotan's FEBI; this sequencer has 8K processors. (Note that the number of the front-end interface does not have to correspond to the number of the sequencer to which it attaches.) Karen is running a **cmattach** subshell; she has been idle for six minutes. No one is using the FEBI on Epicurus.

The information below the list of users provides more data about the CM system:

- The memory size of the processors in this CM is 1 megabyte; it has 32-bit floating point chips.
- Foo has a framebuffer and a CMIOC on both sequencer 0 and sequencer 1.
- Sequencer 0 of Foo is free for use.
- There are three users waiting for the CM. User Patrick will accept any sequencer but wants interface 0; user Sam wants interface 0 and sequencer 1; user Sophie will accept any interface but wants both sequencer 0 and sequencer 1.

Now let's look at another configuration:

```
CM        Seqs Size  Front end I/F User    Idle  Command
---------------------------------------------------------

CLOUSEAU  0-1  8K    wotan      0  kathy          "tests"

        cm2 with 256K memory, 32-bit floating point
        No free sequencers on CLOUSEAU
No process waiting for an interface or sequencer(s)

FOO         1  4K    wotan      1  root            "{CM}*"
FOO                                kyle           "simple"
FOO                                karen  0h 06m "cmattach"
FOO         0  4K    epicurus   0  krill
FOO        ---       thorlac    1  unknown

        cm2 with 1024K memory, 32-bit floating point
        framebuffers on sequencers 0 1
        CMIOCs on sequencers 0 1
        No free sequencers on FOO
No process waiting for an interface or sequencer(s)
```

In this case:

- User Kathy is attached to two sequencers on CM Clouseau and is running a program called **tests**.

- Sequencer 1 of Foo is operating under timesharing; the entry `{CM}*` under `Command` indicates this. Users Karen and Kyle are running timeshared programs on sequencer 1.
- User Krill is attached to sequencer 0 of CM Foo. **cmfinger** doesn't know what command Krill is running, or how long Krill has been idle. This suggests that Krill's front end is a Lisp machine; Lisp machines do not provide idle time or command information (because the concept of a command is meaningless on a Lisp machine).
- **cmfinger** reports that the front end Thorlac has no sequencers. It doesn't know if anyone is using interface 1 on Thorlac. This can happen if you issue **cmfinger** from one front end (for example, Wotan) to obtain information from other front ends. **cmfinger** prints the information it can obtain over the network; if the remote front end sends error or informational messages (for example, "Connection Refused"), **cmfinger** prints them as well. It also prints error messages received from the CM.
- There are no sequencers free.

3.1.1 Options

To obtain information about the CM attached to a particular front-end interface, use the **-i** option, followed by the number of the interface.

To obtain information about CMs to which your front end does not have an interface, list their names on the **cmfinger** command line; case does not matter. Similarly, you can list the names of front ends on the **cmfinger** command line to obtain information about the CMs attached to them; if a front end and a CM have the same name, **cmfinger** interprets the name as that of the CM.

3.2 Listing CMs: The cmlist Command

Use the **cmlist** command to list CMs in your system. The command has several options that make it especially helpful in locating a CM that has a particular resource:

- Use the **-d** option to list the CMs that have DataVaults.
- Use the **-f** option to list CMs with framebuffers.

- Use the **-v** option to list CMs connected to VME I/O computers.
- Use the **-h** option, along with one or more hostnames, to list the names of CMs connected to the front ends with these hostnames.
- Use the **-p** option to restrict the list of CMs to those having the specified number of processors or more. The number can be specified as an integer, or as an integer followed by **k** or **K** to specify thousands of processors.
- Use the **-0**, **-32**, or **-64** option to restrict the list of CMs to those having no floating-point accelerator, a 32-bit floating-point accelerator, or a 64-bit floating-point accelerator, respectively.

The options combine. Thus, to list the names of CMs with *both* a DataVault and a framebuffer, issue this command:

```
% cmlist -d -f
```

Another use of the **cmlist** command is with **cmfinger**. This **cmfinger** command:

```
% cmfinger 'cmlist'
```

prints the status of all CMs in your configuration. This is useful at sites with many CMs attached to many front ends.

3.3 Listing Timeshared Processes: The cmps Command

Use the **cmps** command to obtain information about processes currently running under timesharing. If you are attached to a sequencer running timesharing, **cmps** lists the processes running on that sequencer. If you are not attached, use this syntax:

```
cmps -C cmname -S seqset
```

where *cmname* is the name of a CM, and *seqset* is the number or numbers of the sequencers for which a timesharing daemon is running on that CM. You *must* issue the command from the front end where the timesharing daemon is running.

An example of the command's output is shown below:

```
% cmps
NAME       PID   OWNER      PGS   PRI  %-RT   AC     TSR     AGE
-----------------------------------------------------------------
hilbert  26440 daveg       3    *1   59.5  1391  0:00  187:45
./dvtest 26866 taylor     61     1   42.1  1391  0:42   67:00
walk     27126 daveg       0     1   0.0     49  0:21    0:21
cmps     27129 darm        0     1   0.0     49  0:00    0:00

{CM}* has been up 1:03
102/249 physical pages used; 59% free. 0/272 swap pages
in use.
Per process memory limit 248 pages.

pages shuffled:       10792 pgs   pages swapped out: 46 pgs
pages allocated:       1640 pgs   pages freed:      1456 pgs
process activations: 140 ave. shuffled/activation:  7:00
ave. swapped/activation: 0.00
```

where:

`NAME`	is the name of the program.
`PID`	is the UNIX process ID of the process.
`OWNER`	is the name of the user who owns the process.
`PGS`	is the number of 1024-bit pages that the process takes up on theCM.
`PRI`	is the current priority at which the process is running. In the current release, this is always 1. The asterisk indicates that the process wants to use the CM (that is, it has a CM operation pending or in progress).
`%-RT`	is the percent of real time that the process has received over its lifetime. Since processes start at different times, these percentages can add up to more than 100%.
`AC`	is the activation cost: a statistic that the scheduler uses in choosing the next process to run.

`TSR`	specifies how long it has been since the process has run on the CM (in minutes:seconds).
`AGE`	is the age of the process in minutes:seconds.

The remaining statistics are overall data gathered by the memory managers since the timesharing daemon was started.

3.4 Detaching Users: The cmdetach Command

Use the **cmdetach** command to detach a FEBI (and its user) from a sequencer, thus making the sequencer available. If you are issuing the command from the front end in which the FEBI is located, specify either its interface number or the login ID of the user who is attached via this interface. (You can obtain this information by issuing the **cmfinger** command.) For example, to detach user Karen from the sequencer she is using in the sample **cmfinger** output on page 48, type:

```
% cmdetach karen
```

or

```
% cmdetach -i1
```

if you are on the same front end. Both the interface and the sequencer become available. Any process running on the sequencer is aborted.

If you are on a different front end, you must specify the front end's hostname in addition to the interface number; you cannot simply specify a user's login ID. For example:

```
% cmdetach -i wotan:1
```

After you issue **cmdetach**, the command shows you the status of the system by displaying the **cmfinger** output, and then asks if you're sure you want to proceed with the operation. If you're sure, **cmdetach** detaches the interface from the sequencer and makes the sequencer available.

The user being detached receives a message like this one:

```
**CM** wotan:
(You have been detached from the CM by carl at Tue
Nov 12 22:21:02 1990)
```

(This message is not printed if you detach yourself.) A running process is sent a SIGURG signal when it is being detached.

Note that detaching a user isn't a particularly friendly thing to do if the user has an important program running on the CM. The **cmdetach** command is helpful if, for example, a user simply forgets to exit from a **cmattach** subshell before going home for the night. Detaching the forgetful user reclaims the CM processors. Your system administrator can configure the system so that users are automatically detached if they are idle for a specified period of time; this is a better way of handling the problem of forgetful or selfish users.

You can issue **cmdetach** without any arguments from within a **cmattach** subshell; this detaches your interface from the CM but leaves you in the subshell. This lets you preserve the UNIX environment of the subshell.

3.4.1 Under Timesharing

NOTE: Do not detach an interface or user attached to a sequencer under timesharing. If you do so, you abort *all* processes running on that sequencer and shut down timesharing.

If you try to do this, **cmdetach** notes that timesharing is running, and queries you an extra time to make sure this is what you want to do, as in the example below.

Note that the message displayed in response to the **cmdetach** specifies the owner of the timesharing daemon, even if you issue the **cmdetach** command to detach some other user.

If you wish to stop timesharing from running on a sequencer, talk to your system administrator, who can issue the **cmts-shutdown** command; this command shuts down timesharing in an orderly fashion.

```
% cmdetach -i0
CM        Seqs  Size  Front end I/F User  Idle      Command
-------------------------------------------------------------

JACQUES    1    8K     enki       0 root  0h 00m     "{CM}*"
                                    dm    0h 00m "cmattach"
JACQUES    0    8K     wotan      1 kathy              "tests"

            cm2 with 256K memory, 64-bit floating point
            no free sequencers on JACQUES

cmdetach: You are about to try to detach user ''dm''
from Connection Machine ''JACQUES''.

Proceed? [yn] y

cmdetach: Are you sure? root appears to be running
timesharing on Connection Machine JACQUES from front-end
enki.think.com, interface 0. Detaching or powering up a
timeshared interface will disrupt all the users of the
interface.

Really proceed? [yn] n

Function aborted.
```

3.5 Resetting the CM: The cmcoldboot Command

Use **cmcoldboot** to reset the state of the CM hardware to which you are attached. Issuing the command loads microcode into the control store of the sequencer, initializes system tables, and clears processor memory. You can issue **cmcoldboot** only from a **cmattach** subshell, or as part of a script file being submitted as an NQS batch request; see Chapter 2.

By default, the CM hardware is automatically cold booted when you first attach to it. Depending on the application, you might want to reset the CM between program runs. A warm boot is performed every time a program is executed; this warm boot does everything the cold boot does except clear processor memory and reload microcode into the sequencer. For some applications, not cold booting saves the time of reloading data into the CM processors for every single program run. If, however, you want to make sure that the current run of a program is not affected by the state left from a previous one, issue **cmcoldboot** between the runs. Also, if a computation is interrupted at some point and not continued, you should issue **cmcoldboot** and start the program over from the beginning.

As arguments to **cmcoldboot**, you can specify the geometry of a VP set in Paris programs; in programs written in high-level languages, the compiler does this for you. You do not need to specify **-g**, as you do for **cmattach**. For example,

```
% cmcoldboot 64 256
```

creates a VP set of 16,384 processors, arranged in a 64-by-256 geometry.

Another option **cmcoldboot** has in common with **cmattach** is **-u**, which you can use to specify which version of the CM microcode you want the CM to use. See page 28 for a discussion of this option.

3.5.1 Under Timesharing

Issuing **cmcoldboot** has an effect only if you have exclusive access to the CM. It is unnecessary if your program is executing under timesharing, since the system automatically performs a cold boot every time a program runs.

3.6 Timing a CM Program: The cmtime Command

Use the **cmtime** command to print information about the execution time of a program that uses the Connection Machine system. The results are displayed on the standard output. For example, to execute the program **myprog** and obtain timing information, issue this command:

```
% cmtime myprog
```

The output is slightly different depending on whether the program runs in exclusive mode or in timesharing mode. In exclusive mode, the output looks like this:

```
Elapsed time: 115.38 sec.; CM time 111.72 sec.
Front end virtual time (seconds): 10.39 user, 1.26 system
CM Utilization: 97%; Front end utilization: 10%
```

Under timesharing, the output looks like this:

```
Elapsed time: 14.25 sec.; CM time 7.53 (out of 13.63) sec.
Front end virtual time (seconds): 2.04 user, 4.40 system
CM Utilization: 55%; {CM*} share: 96%
```

The fields have these meanings:

- **Elapsed time** is the total elapsed wallclock time that the program took to run.
- **CM time** is the amount of time that the program used the CM. For timesharing, the amount of time during which the program had the CM available for use is also listed.
- **Front end virtual time** gives the amount of time the program used the front end. User time is the amount of front-end time spent in the program itself. System time is the time spent by the front-end operating system kernel on behalf of the program.
- **CM Utilization**, for exclusive mode, is the percentage of elapsed time represented by CM time. For timesharing, it is the percentage of time that the program used the CM out of the total time the CM was available to it.
- **Front end utilization** (exclusive mode only) is the percentage of elapsed time represented by front-end user and system time.
- **{CM*} share** (timesharing mode only) is the percentage of elapsed time during which the CM was available to the program.

Note these points in using **`cmtime`**:

- **`cmtime`** obtains its data from the CM's accounting system. If the accounting daemon is not running, **`cmtime`** can only display information about the use of the front end. Ask your system administrator if the accounting daemon is running on your system.

- The **cmtime** data is an estimate, obtained from polling the sequencer every .01 second. If you need more accurate information, use the timing routines discussed in Section 6.2 on page 99.
- As discussed in that section, to increase the accuracy of your timing, we recommend using a front end that is as unloaded as possible, and running the program several times; the minimum elapsed time reported will be the most accurate.

3.7 Obtaining Information about the cmattach Subshell: The cm Command

Use the **cm** command to obtain information about the CM to which you are attached via a **cmattach** subshell. The syntax is:

```
cm [-C] [-d] [-i] [-s]
```

where:

-C	prints the name of the CM to which this subshell is attached.
-d	prints the name of the CM device driver. This is always **/dev/cm**.
-i	prints the number of the interface to which this subshell is attached.
-s	prints the sequencer set to which this subshell is attached.

One way to use this command is to change your prompt while in the subshell. For example, issue this command (in the C shell) to set your prompt to the name of the CM to which you are attached:

```
% set prompt = 'cm -C'
```

3.8 Changing the Priority of Timesharing Jobs

Use the **cmnice** and **cmrenice** commands to change the priority of a process running under CM timesharing; the higher the priority, the more often the process is scheduled to run on the CM. Use **cmnice** to set the priority of the process when you first run it. Use **cmrenice** to change the priority of a process that is already running under timesharing.

The **cmnice** command takes as its argument a number from 0 to 5 (0 to 9 if you are the superuser), followed by the name of the program and any arguments to the program. The lower the number, the lower the priority; the default is 5. Note that therefore only the superuser can increase the priority of a process beyond this standard timesharing priority. Users can only lower their priority. For example,

```
% cmnice -0 simple
```

executes the program **simple** with a **cmnice** value of 0; this means that the process will run only when no other process in the system wants to.

The **cmrenice** command takes as its arguments the process ID of the process whose priority you want to change, and, optionally, **-p** *n*, where *n* is a number from 0 to 5 (0 to 9 if you are the superuser). If you omit the **-p** argument, the process's priority is reduced by 1. You can change the priority only of a process you own. (The superuser can change the priority of any process.)

You can obtain the process's process ID by issuing the **cmps** command; see Section 3.3 on page 51.

Note that you can use **cmrenice** to increase the priority of a process, but only if you originally ran the program with a lower-than-average priority, and only up to the standard timesharing priority of 5.

For example,

```
% cmrenice -p3 26440
```

changes to 3 the **cmnice** priority of the process with process ID 26440.

3.9 Displaying CM Manual Pages: The cmman Command

CM System Software contains a large number of on-line manual pages. Included in the 6.1 release are manual pages for:

- all CMost user and system administrator commands
- all CM Fortran utilities and intrinsics
- all CMFS commands and calls
- all Paris instructions

More manual pages will be added in the future.

Use the **cmman** command to display one of these pages. For example, issue this command to display the man page for the CM Fortran utility **CMF_ALLOCATE_ARRAY**:

```
% cmman CMF_ALLOCATE_ARRAY
```

You can use any of the options accepted by the UNIX command **man**. You can also issue **cmman** to display standard UNIX man pages; use it in the same way you would use **man**.

Note these points in specifying the names of CM functions and commands:

- If a function or command name has a prefix (such as **CM** or **CMFS**), make this prefix uppercase. Make the rest of the name lowercase, using underscores (not hyphens) as word-separators (except for the cases noted below). For example, **CMFS_read_file** is correct; **cmfs_read_file** is incorrect.

- Fortran names can also be specified using all uppercase. For example, **CMF_ALLOCATE_ARRAY** and **CMF_allocate_array** are both correct.

- Use Lisp syntax to specify a function used only in Lisp: for example, **CMFS:make-stat**.

- **For Paris functions,** you can omit the **_1L**, **_2L**, etc. suffixes, as well as the **_always**, **_constant**, and **_const** qualifiers. You can also specify a general name that matches the heading in the *Paris Reference Manual.* For example, to display the man page for **CM_c_add_2_1L** you can simply refer to it as **CM_c_add**.

- **For CMFS functions,** in general specify exact names when they end with **_always**; however, if a non-always version of the function exists, leave off the **_always**. For example, specify the function **CMFS_read_file_always** as **CMFS_read_file** (because there *is* a **CMFS_read_file**). However, you would specify **CMFS_transpose_always** as **CMFS_transpose_always** (because there is no function **CMFS_transpose**).

3.9.1 If You Don't Want to Use cmman

You can use the **man** command instead of **cmman** to display CM manual pages; you can also use **xman** to display these man pages under X. To do this, however, you have to add the CM man page directories to the path set by your **MANPATH** environment variable. One advantage of this approach is that it gives you somewhat more flexibility in how the search for a man page is carried out.

By default, the CM man-page directories are in the path **/usr/local/man**, with each subject area in its own directory. Note, however, that your system administrator may have put them somewhere else; check with your system administrator if you can't find them. These directories are currently available:

CMF	CM Fortran intrinsics and utilities
CMFS	CM file system calls and I/O commands
CMOST	User-level CM commands
PARIS	All Paris functions

Add these directories anywhere in your **MANPATH**. For example,

```
% setenv MANPATH /usr/man:/usr/local/man:/usr/local/man/CMF: \
  /usr/local/man/CMFS:/usr/local/man/CMOST:/usr/local/man/PARIS
```

If you add these directories before the standard directories (**/usr/man**, **/usr/local/man**), they are searched first; this may give you a slightly faster response time for CM man pages. You can also omit directories you aren't interested in. For example, if you don't use Fortran, you can omit the **CMF** directory; once again, this can speed up a search slightly.

Part III
Programming with the Connection Machine System

Chapter 4

Programming: The Basics

This chapter describes the basic process of programming for the Connection Machine system. In it, we assume that you are familiar with programming in a UNIX environment. See Chapter 6 for a discussion of tools you can use in programming.

Users of *Lisp and Lisp/Paris should read Part V of this manual for an introduction to programming in the Lisp environment.

4.1 Choosing a Language

The Connection Machine system offers several data parallel programming languages, which are discussed in this section. Complete information on these languages is available in separate manuals in the Connection Machine documentation set.

NOTE: Your Connection Machine system may not contain all the high-level programming languages discussed here; check with your system administrator.

4.1.1 Paris

Paris is the PARallel Instruction Set for programming the Connection Machine system. It is roughly similar to the machine-level instruction set of an ordinary computer. Interfaces to Paris are available from C, Lisp, and Fortran; the resulting "languages" are referred to as C/Paris, Lisp/Paris, and Fortran/Paris. These interfaces all call exactly the same Paris instructions; the only difference is that each interface conforms to the syntax and data types of its higher-level language.

The compilers for the data parallel languages described below (except for slicewise CM Fortran) generate code that makes direct calls to Paris. You can also include calls to Paris in programs written in these languages. You may be able to write faster code using Paris calls; the trade-off, of course, is that using Paris calls requires a deeper understanding of the Connection Machine architecture.

4.1.2 CM Fortran

CM Fortran implements the Fortran 77 programming language, extended with array-handling facilities from the Fortran 90 standard. CM Fortran supports all features of Fortran 77 that control allocation of or access to data residing on the front end; some restrictions are placed on the use of Fortran 77 features that would cause a program to depend on the storage order of data residing in CM memory. Most array data in CM Fortran is allocated in CM memory, one element per processor, and array operations on such data are performed by the CM processors in parallel.

A *slicewise* version of the CM Fortran compiler is available; this version generates special optimizations for programs running on systems with a 64-bit floating point accelerator.

4.1.3 C*

C* is a data parallel extension of the C programming language. C* programs are a mixture of familiar C code, which operates on data on the front end, and new C* code. C* provides new syntax for describing the size and shape of parallel data and for creating parallel variables. It also provides methods for choosing the parallel variables, and the specific data points within parallel variables, upon which C* code is to act.

4.1.4 *Lisp

*Lisp is an extension of the Common Lisp language. It allows you to write data parallel programs in Lisp that map simply onto Connection Machine hardware features. Most *Lisp functionality corresponds directly to underlying Paris instructions, making the execution speed of a *Lisp program predictable.

The CM system provides both a *Lisp interpreter and a *Lisp compiler; the *Lisp compiler executes as part of the Common Lisp compiler. Compiled

*Lisp runs more efficiently than interpreted *Lisp. There is also a *Lisp simulator that you can use to test and debug *Lisp code without using a CM. The simulator runs entirely on the front end and simulates the operations of an exclusive-access CM.

If you are going to program in *Lisp or Lisp/Paris, you can omit the remainder of this chapter and go instead to Part V, "In the Lisp Environment."

4.2 Overview of the Programming Process

The entire process of CM programming takes place on the front end. You need to be attached to a CM only to run the program and to debug it.

The remaining sections of this chapter discuss the basics of the programming process.

4.3 Developing a Program

You write source code for a data parallel program on the front end as you normally would for a serial program. The only difference is that C* and CM Fortran have new suffixes for the names of files containing source code: C* files must end in `.cs`, and CM Fortran files must end in `.fcm`. C/Paris and Fortran/Paris programs are standard C and Fortran programs that include the library of Paris operations and make calls to these operations.

4.3.1 Libraries and Include Files

This section lists CM libraries and include files that you are likely to use in your data parallel programs; check with your system administrator for the location of these files. You can also use standard libraries and include files—for example, `<stdio.h>` for C/Paris and C* programs. Consult the CM documentation for the relevant language for complete information—for example, you must explicitly link some libraries to your program, while other libraries are linked automatically. Also note that some languages and libraries may not be available at your site.

Table 6. CM software libraries

Library	Contents
`libckpt.a`	Checkpointing library
`libckpt-pg.a`	Checkpointing library for use when profiling
`libcmfs.a`	CM file system routines
`libcmfs-pg.a`	CM file system routines to use when profiling
`libcmsr.a`	*Render and Generic Display Interface graphics routines
`libcmsr-pg.a`	*Render and Generic Display routines to use when profiling
`libcmssl.a`	CM Scientific Software Library routines
`libparis.a`	Paris instructions for C
`libparis-pg.a`	Paris instructions to use when profiling
`libparisfort.a`	Paris instructions for Fortran

4.4 Compiling a Program

You compile and link data parallel programs on the front end as you normally do.

Use the command **`cs`** to invoke the C* compiler, which works in conjunction with the standard C compiler.

Use the command **`cmf`** to invoke the CM Fortran compiler; it can also invoke the standard Fortran and C compilers, as appropriate.

These commands and their options are discussed in detail in the user's guides for C* and CM Fortran.

Use the standard Fortran and C compilers to compile Fortran/Paris and C/Paris programs, respectively.

Table 7. CM include files

File	Include with:
`Xcm.h`	X Windows graphics routines
`attach.h`	Attaching routines for C
`attach-fort.h`	Attaching routines for Fortran
`ckpt.h`	Checkpointing routines for C/Paris
`ckpt-fort.h`	Checkpointing routines for Fortran
`cm_conf.h`	CM character-special device drivers
`cm_dir.h`	Certain CMFS directory routines
`cm_errno.h`	CMFS routines
`cm_file.h`	Certain CMFS file routines
`cm_mount.h`	`CMFS_statfs` routine
`cm_ioctl.h`	CMFS ioctl routines
`cm_param.h`	Certain CMFS routines
`cm_stat.h`	Certain CMFS statistics routines
`cmfb.h`	Framebuffer graphics routines
`cmsr.h`	*Render routines for C* and C/Paris
`cmsr-fort.h`	*Render routines for CM Fortran and Fortran/Paris
`cmssl-cmf.h`	CMSSL routines for CM Fortran
`cmssl-paris.h`	CMSSL routines for C/Paris
`cmssl-fort-constants.h`	Symbolic CMSSL constants for CM Fortran
`cmssl-fort-paris.h`	CMSSL routines for Fortran/Paris
`cmssltypes.h`	CMSSL routines for C* and C/Paris
`cmtypes.h`	CM data types, when `paris.h` is not included
`display-fort.h`	Generic Display Interface routines for CM Fortran and Fortran/Paris
`display.h`	Generic Display Interface routines for C* and C/Paris
`paris-configuration-fort.h`	Paris instructions in CM Fortran and Fortran/Paris
`paris.h`	Paris instructions in C* and C/Paris

4.5 Executing a Program

Chapter 2 discusses in detail the process of attaching to a CM and executing a program, using either batch or direct access. See also Chapter 5, which discusses how to attach to a CM from within a program, and Chapter 6, which discusses how to restart a program that has been checkpointed during execution.

4.6 Debugging a Program

We recommend that you use Prism, the CM's programming environment, to debug and analyze the performance of your program. For complete information on Prism, see the *Prism User's Guide*.

4.7 UNIX Utilities

You can use other standard UNIX utilities like **`make`** and **`gprof`** with a data parallel program. See Chapter 6 for information about how to use **`gprof`** to profile a data parallel program.

Chapter 5

Attaching and Detaching from within a Program

In their programs, users can include calls to a variety of functions that correspond to many of the commands discussed in Part II of this guide. Using these calls, a program can:

- Attach to a CM resource, specifying the interface, the sequencer(s), the memory size, and other characteristics of the resource.
- Detach from the CM.
- Detach other users from the CM.
- Power up and cold boot the CM.
- Obtain `cmfinger`-style information about who is using the CM.

The routines discussed in this chapter are available for C*, CM Fortran, C/Paris, and Fortran/Paris programs. For information on *Lisp and Lisp/Paris versions, see Part V.

5.1 Overview

Table 8 lists the routines discussed in this chapter. See the individual sections of the chapter for the specific C and Fortran versions of the routines, and for their arguments.

Table 8. Routines for attaching, detaching, and obtaining **cmfinger** data

Routine	Use
CM_attach	Attach to any CM resource.
CM_attach_to	Attach to the specified CM resource.
CM_preempt	Detach anyone else, then attach.
CM_detach	Detach from the CM.
CM_detach_cm	Detach users from the specified CM.
CM_detach_interface	Detach users from the specified FEBI.
CM_detach_user	Detach the specified user.
CM_detach_cm_by_seq	Detach users from the specified sequencer(s) on the specified CM.
CM_coldboot	Cold boot the CM.
CM_powerup	Power up the specified CM.
CM_finger	Print a complete **cmfinger** output.

There are also a number of C-only routines that provide more flexibility in obtaining **cmfinger** data; these routines are described in Section 5.5.1. Section 5.6 describes C routines for obtaining information about sequencers.

Section 5.8 describes a C-only mechanism for obtaining a user's command line arguments to determine the configuration of the desired CM resource.

To use the routines discussed in this chapter, include the file **<cm/attach.h>** (for C programs); for Fortran programs, include the file **/usr/include/cm/attach-fort.h**. Link with the Paris library, **libparis.a**, if your program does not do so automatically.

5.2 Attaching to a CM

Three routines are available for attaching to a CM:

- Call **CM_attach** to attach to any available CM resource.
- Call **CM_attach_to** to attach to a specific CM resource; the arguments to the routine specify the resource.
- Call **CM_preempt** to detach a process from the CM, then attach in its place.

In addition, the routine **CM_attached** is available to let you determine if the program is attached to a CM. **CM_attached** takes no arguments; it returns non-zero if the program is attached, and 0 if it isn't attached.

5.2.1 Attaching to Any CM Resource

Call the **CM_attach** routine to attach to any available CM resource.

The routine has this definition in C:

```
int CM_attach()
```

Call the routine from Fortran as follows:

```
call CM_attach()
```

CM_attach takes no arguments, and settles for any free CM resource it can get. If resources are available, by default it receives the highest-numbered sequencer that is free on the CM connected to the lowest-numbered interface that is also free.

If the process is already attached when it issues **CM_attach** (for example, because the program was executed from a **cmattach** subshell), **CM_attach** inherits the existing attachment.

CM_attach has these return values:

- 0 — There are no free CM resources available from this front end. The error message "No CM resources available" appears on your stderr.
- -1 — The request couldn't be satisfied for some other reason. An explanation of the failure is displayed on your stderr.
- >0 — If the return value is greater than 0, the process successfully attached, and the value represents the number of processors to which it attached.

5.2.2 Attaching to a Specific CM Resource

Call the **CM_attach_to** routine to attach to a specific CM resource.

The routine has this definition in C:

```
int CM_attach_to(char *cm_name, CM_bits bits)
```

Call the routine from Fortran as follows:

```
call CM_attach_to(cm_name, bits)
```

where:

cm_name	is the name of the CM to which you want the program to attach. Specify 0 to indicate that you will accept any CM.
bits	is a bit-mask that specifies the configuration of the CM resource to which you want to attach (see below). Specify 0 to indicate that any configuration is acceptable. The defaults are the same as those for the **cmattach** command: you are attached to the highest-numbered sequencer that is free on the lowest-numbered available interface.

Thus,

```
CM_attach_to(0, 0);
```

in C, is equivalent to **CM_attach()**.

Specifying the CM Resource

The bit-mask in **CM_attach_to** is also used in other routines. You can use this bit-mask to specify:

- Whether the process is to wait for the resource to become available.
- What memory size you want.
- The type of floating-point accelerator you want.
- Whether you want the resource to have a framebuffer.
- Whether you want to run in exclusive mode only, under timesharing only, or you don't care.
- Whether you want the resource to have a DataVault.
- The number of sequencers you want.

- The precise set of sequencers you want.
- The number of processors you want.

Table 9 lists the arguments and their meaning. Note that Fortran versions of the arguments all begin with *I* (so that Fortran implicitly assumes that the values are integral).

Table 9. Arguments to **CM_attach_to***

Argument	Meaning
CMA_WAIT	Wait for the resource.
CMA_CM*n*	Get a resource on the specified CM model.
CMA_M*memsize*	Get a resource with at least *memsize* memory size.
CMA_MEXACT	Get a resource with exactly **CMA_M***memsize*.
CMA_FPU_*fputype*	Get a resource with *fputype* floating-point accelerator.
CMA_FRAMEBUFFER	Get a resource with a framebuffer.
CMA_DATAVAULT	Get a resource with a DataVault.
CMA_TIMESHARED	Get a resource running under timesharing.
CMA_EXCLUSIVE	Get a resource running in exclusive mode.
CMA_UCCS_*n*	Get a resource with *n* sequencers.
CMA_UCC*n*[**_and_***n*]	Get a resource with the specified sequencer set.
CMA_P*n*	Get a resource with at least *n* processors.
CMA_PEXACT	Get a resource with exactly **CMA_P***n* processors.
CMA_I*n*	Attach to the CM via the specified interface.

*Fortran versions have an *I* added to the beginning.

You can specify any combination of requirements, as long as they are consistent. For example,

```
CM_attach_to("frodo", CMA_WAIT + CMA_EXCLUSIVE);
```

(in C) specifies that the process is to attach to Frodo in exclusive mode, and will wait for resources. But

```
CM_attach_to("frodo", CMA_TIMESHARED
                   + CMA_EXCLUSIVE); /* wrong */
```

incorrectly asks for both a timeshared resource and an exclusive resource.

Waiting for the resource. To specify that the process is to wait for the required CM resource to become available, include the argument **CMA_WAIT**

(ICMA_WAIT in Fortran). If you don't include this argument, CM_attach_to returns 0 if it can't obtain the specified resource.

Specifying the CM model. If you have more than one model of the CM available at your site, use the CMA_CM*n* argument (ICMA_CM*n* in Fortran) to specify the model to which you want to attach. Possible values are 2 and 200. Use 2 if you want to attach to a CM-2 series machine; use 200 if you want to attach to a CM-200 series machine.

If you omit this argument, you are attached to whatever is available, using the standard algorithm.

Specifying memory size. To request a memory size for the CM resource, use one of the arguments listed below:

CMA_M64K or CMA_M64	64K memory size
CMA_M256K or CMA_M256	256K memory size
CMA_M1M or CMA_M1	1M memory size
CMA_M4M or CMA_M4	4M memory size

(Fortran versions have an *I* added to the beginning.) These arguments indicate that you will accept *at least* the specified memory size, unless you also specify CMA_MEXACT (ICMA_MEXACT in Fortran); this indicates that you will accept *only* the specified memory size. Omit these arguments if you will accept any memory size.

Macros are available that let you convert a memory size constant into an actual memory size, and vice versa. These may be useful, for example, in converting between human input and the format required by CM_attach_to.

Use CMA_MSIZE (ICMA_MSIZE in Fortran) to convert a size constant into an actual memory size. For example, CMA_MSIZE(CMA_M1M) returns 1048576.

Use CMA_MBITS (ICMA_MBITS in Fortran) to convert an actual memory size to a constant that can be used in CM_attach_to. For example, CMA_MBITS(1048576) returns CMA_M1M.

Specifying the floating-point accelerator. To request a specific type of floating-point accelerator (or no floating-point accelerator at all), use one of these arguments:

CMA_FPU_32	32-bit floating-point accelerator
CMA_FPU_64	64-bit floating-point accelerator
CMA_FPU_NONE	No floating-point accelerator

(Fortran versions have an *I* added to the beginning.) Omit these arguments if you will accept any kind of floating-point accelerator. Note that this is different from specifying **CMA_FPU_NONE**, which specifically requests a resource with *no* floating-point accelerator.

Specifying attached devices. Include the argument **CMA_DATAVAULT** (**ICMA_DATAVAULT** in Fortran) to request a resource that has access to a DataVault.

Include the argument **CMA_FRAMEBUFFER** (**ICMA_DATAVAULT** in Fortran) to request a resource that has access to a framebuffer.

Specifying timeshared vs. exclusive access. Include the argument **CMA_EXCLUSIVE** (**ICMA_EXCLUSIVE**) if you want only a resource operating in exclusive mode.

Include the argument **CMA_TIMESHARED** (**ICMA_TIMESHARED** in Fortran) if you want only a resource operating under timesharing.

Omit these arguments if you will accept a resource operating either under timesharing or in exclusive mode.

Specifying the number of sequencers. Use one of these arguments to request a specific number of sequencers:

CMA_UCCS_1	1 sequencer
CMA_UCCS_2	2 sequencers
CMA_UCCS_4	4 sequencers

(Fortran versions have an *I* added to the beginning.) Omit these arguments if you will accept any number of sequencers.

Specifying the sequencer set. Use one of these arguments to request a specific sequencer or set of sequencers:

CMA_UCC0	sequencer 0
CMA_UCC1	sequencer 1
CMA_UCC2	sequencer 2
CMA_UCC3	sequencer 3
CMA_UCC0_and_1	sequencers 0 and 1
CMA_UCC2_and_3	sequencers 2 and 3
CMA_UCC0_to_3	sequencers 0 through 3

(Fortran versions have an *I* added to the beginning.) Omit these arguments if you will accept any sequencer or set of sequencers.

Specifying the number of processors. Use one of these arguments to request a specified number of processors:

CMA_P4K	4096 processors
CMA_P8K	8192 processors
CMA_P16K	16384 processors
CMA_P32K	32768 processors
CMA_P64K	65536 processors

(Fortran versions have an *I* added to the beginning.) These arguments indicate that you will accept *at least* the specified number of processors, unless you also specify **CMA_PEXACT**; this indicates that you will accept *only* the specified number of processors. Omit these arguments if you will accept any number of processors.

Macros are available that let you convert a processor-number constant into an actual number of processors, and vice versa:

Use **CMA_PCOUNT** (**ICMA_PCOUNT** in Fortran) to convert a processor-number constant into an actual number of processors. For example, **CMA_PCOUNT(CMA_P4K)** returns 4096.

Use **CMA_PBITS** (**ICMA_PBITS** in Fortran) to convert an actual number of processors to a constant that can be used in **CM_attach_to**. For example, **CMA_PBITS(4096)** returns **CMA_P4K**.

Specifying the interface. In C only, use the **CMA_I***n* argument to specify the number of the front-end bus interface by which you want to attach to the CM. For example, specify **CMA_I0** to attach via interface 0. Omit this argument if you will accept any interface.

To convert an interface number into a bit mask that can be passed to **CM_attach_to**, use the macro **CMA_interface_to_bits** or **CMA_I** (**ICMA_interface_to_bits** or **ICMA_I** in Fortran); its single argument is the interface number. For example, this Fortran code fragment asks for interface 0 in exclusive mode:

```
I = ICMA_I(0)
call CM_attach_to(ICMA_exclusive + I)
```

To convert the bits to an interface number, use the macro **CMA_bits_to_interface** (**ICMA_bits_to_interface** in Fortran).

Examples

This Fortran call requests a resource on CM Top with 8192 processors and a framebuffer:

```
call cm_attach_to("top", ICMA_P8K + ICMA_FRAMEBUFFER)
```

This C call requests a resource on CM Frodo with one sequencer and a 64-bit floating-point accelerator, running in exclusive mode; the process is willing to wait:

```
 CM_attach_to("frodo", CMA_WAIT
                          + CMA_UCCS_1
                          + CMA_FPU_64
                          + CMA_EXCLUSIVE);
```

Return Values

CM_attach_to has the following return values:

0	There are no free resources on this front end that match the requested configuration.
-1	The requested configuration is contradictory (for example, the CM you specified doesn't have the kind of floating-point accelerator you requested).
>0	The process is attached to this number of processors.

5.2.3 Preempting Another User

Call the **CM_preempt** routine to detach whoever is using the specified CM resource, and then attach to the resource in its place.

NOTE: Only the superuser or the owner of the timesharing daemon can detach timesharing from a CM resource.

The routine has this definition in C:

```
int CM_preempt(char *cm_name, CM_bits bits)
```

Call the routine from Fortran as follows:

```
call CM_preempt(cm_name, bits)
```

where *cm_name* and *bits* are the name of the CM and the bit-mask that specifies the resource, as discussed in the previous section.

The return values are the same as those for **CM_attach_to**; see the previous section.

Example

This C call preempts the user of sequencer 0 on CM Frodo:

```
CM_preempt("frodo", CMA_UCC0);
```

5.3 Detaching

These routines are available for detaching from a CM:

- Call **CM_detach** to detach the calling process from the CM resource to which it is attached.
- Call **CM_detach_cm** to detach anyone attached to a specified CM.
- Call **CM_detach_interface** to detach anyone attached to a CM on a specified interface.
- Call **CM_detach_cm_by_seq** to detach anyone attached to a CM on a specified sequencer set.
- Call **CM_detach_user** to detach a specified user from the CM resource to which he or she is attached.

NOTE: Only the superuser or the owner of the timesharing daemon can detach timesharing from a CM resource.

5.3.1 Detaching the Calling Process

Call the routine **CM_detach** to detach the calling process from its CM resource.

The routine has this definition in C:

```
void CM_detach()
```

Call the routine from Fortran as follows:

```
call CM_detach()
```

There are no return values. The process is detached from the CM resource it is using.

5.3.2 Detaching All Users from a CM

Call the routine **`CM_detach_cm`** to detach all users who are attached to the CM you specify via a single FEBI from the front end on which your process is running.

The routine has this definition in C:

```
int CM_detach_cm(char *cm_name, boolean confirm)
```

Call the routine from Fortran as follows:

```
call CM_detach_cm(cm_name, confirm)
```

where:

cm_name	is the name of the CM from which you want to detach users.
confirm	is either **`TRUE`** or **`FALSE`**. If it is **`TRUE`**, the routine asks for confirmation of the detach by printing a message on the standard error device. If it is **`FALSE`**, the routine proceeds with the detach without waiting for confirmation.

Note:

- Only the superuser or the owner of the timesharing daemon can detach timesharing from a CM resource.
- The routine works only if the CM is connected to the front end via one front-end bus interface; if there are multiple interfaces, use **`CM_detach_interface`** to choose which interface you want to detach; see below.
- If the calling process is attached to the specified CM, it too is detached.

- Users attached to the specified CM via an interface on another front end are not detached.

Return Values

CM_detach_cm has these return values:

0	The detach was successful.
-1	The detach failed. An explanation of the failure is printed on your stderr.
-2	The CM is attached to this front end on more than one interface. Use the **CM_detach_interface** routine to detach it from each interface individually.

5.3.3 Detaching Users from a Specific Interface

Call the routine **CM_detach_interface** to detach all users attached to a CM via the front-end bus interface that you specify.

The routine has this definition in C:

```
int CM_detach_interface(int iface, boolean confirm)
```

Call the routine from Fortran as follows:

```
call CM_detach_interface(iface, confirm)
```

where:

iface	is the number of the interface from which users are to be detached.
confirm	is either **TRUE** or **FALSE**. If it is **TRUE**, the routine asks for confirmation of the detach by printing a message on the standard error device. If it is **FALSE**, the routine proceeds with the detach without waiting for confirmation.

Note:

- Only the superuser or the owner of the timesharing daemon can detach timesharing from a CM resource.

- If the calling process is attached via the specified interface, it too is detached.
- Users attached to the specified CM via any other interface are not detached.

Return Values

`CM_detach_interface` has these return values:

0	The detach was successful.
-1	The detach failed. An explanation of the failure is printed on your stderr.

5.3.4 Detaching Users from a Specific Sequencer Set

Use the routine **`CM_detach_cm_by_seq`** to detach all users from the specified sequencer(s) on the specified CM.

The routine has this definition in C:

```
int CM_detach_cm_by_seq(char *cm_name, int seqs,
                        boolean confirm)
```

Call the routine from Fortran as follows:

```
call CM_detach_cm_by_seq(cmname, seqs, confirm)
```

where:

cm_name	is the name of the CM.
seqs	specifies the sequencer(s) from which users are to be detached. Possible values are **`CMA_UCC0`**, **`CMA_UCC1`**, **`CMA_UCC2`**, and **`CMA_UCC3`** for individual sequencers; **`CMA_UCC0_and_1`**, **`CMA_UCC2_and_3`**, and **`CMA_UCC0_to_3`** for sequencer sets. (Fortran versions have an *I* added to the beginning.)
confirm	is either **`TRUE`** or **`FALSE`**. If it is **`TRUE`**, the routine asks for confirmation of the detach by printing a **`cmfinger`** listing on

its standard error device and asking if you are sure you want to disrupt the listed users. If it is **FALSE**, the routine proceeds with the detach without waiting for confirmation.

Note:

- Only the superuser or the owner of the timesharing daemon can detach timesharing from a CM resource.
- If the calling process is attached via one of the specified sequencers, it too is detached.
- Users on other front ends attached to this sequencer are also detached.
- Users attached to the specified CM via any other sequencers are not detached.

Return Values

CM_detach_cm_by_seq has these return values:

0	The detach was successful.
-1	The detach failed. An explanation of the failure is printed on your stderr.

5.3.5 Detaching a Specific User

Use the routine **CM_detach_user** to detach the user you specify.

The routine has this definition in C:

```
int CM_detach_user(char *uname, boolean confirm)
```

Call the routine from Fortran as follows:

```
call CM_detach_user(uname, confirm)
```

where:

uname	is the name of the user whom you want to detach.

confirm	is either **TRUE** or **FALSE**. If it is **TRUE**, the routine asks for confirmation of the detach by printing a message on the standard error device. If it is **FALSE**, the routine proceeds with the detach without waiting for confirmation.

Return Values

CM_detach_user has these return values:

0	The detach was successful.
-1	The detach failed. An explanation of the failure is printed on your stderr.
-2	The request was ambiguous (for example, because the user is attached to more than one CM resource).

5.4 Cold Booting and Powering Up a CM

Call the **CM_cold_boot** routine to cold boot a CM resource.

Call the **CM_powerup** routine to power up a CM.

5.4.1 Cold Booting a CM Resource

Call the **CM_cold_boot** routine to cold boot the CM resource to which you are attached. For information on cold booting, see Chapter 3.

The routine has this definition in C:

```
void CM_cold_boot()
```

Call the routine from Fortran as follows:

```
call CM_cold_boot()
```

NOTE: **CM_cold_boot** has no effect if you are attached to a sequencer that is running under timesharing.

5.4.2 Powering Up a CM

Call the **CM_powerup** routine to initialize the CM you specify. It is equivalent to the **cmpowerup** command, which is discussed in the *CM System Administrator's Guide*. **CM_powerup** initializes the nexus of the CM and detaches any users who are currently attached.

The routine has this definition in C:

```
int CM_powerup(char *cm_name, boolean confirm)
```

Call the routine from Fortran as follows:

```
call CM_powerup(cm_name, confirm)
```

where:

cm_name	is the name of the CM you want to power up. If you specify a 0 for this argument, **CM_powerup** powers up the CM to which you are currently attached.
confirm	is either **TRUE** or **FALSE**. If it is **TRUE**, the routine requests confirmation of the powerup by printing the **cmfinger** output for the CM to the standard error device and asking if you really want to detach these users. If it is **FALSE**, the routine proceeds with the powerup without waiting for confirmation.

NOTE: **CM_powerup** will not power up a CM on which timesharing is running. You must first take down the timesharing daemon; see the *CM System Administrator's Guide* for information on how to do this.

Return Values

CM_powerup has these return values:

0	The powerup was successful.
-1	The powerup failed. An explanation of the failure is printed on your stderr.

5.5 Obtaining cmfinger Data

Call **CM_finger** to print the standard **cmfinger** display on the standard output; see Chapter 3 for examples of this display.

CM_finger has this definition in C:

```
void CM_finger()
```

Call the routine from Fortran as follows:

```
call CM_finger()
```

CM_finger prints on the standard output the **cmfinger** display for all front ends connected to the same CMs as the front end from which the routine was called.

5.5.1 C-Only cmfinger Routines

The routines discussed in this section are provided in C only. They give more flexibility in the use of the **cmfinger** data.

- **CM_finger_d** returns a list of **CM_finger_data** structures describing all the interfaces to all the CMs attached to this front end. The **CM_finger_data** structure is described below.

```
CM_finger_data *CM_finger_d()
```

- **CM_finger_delete** deletes the memory allocated to hold the list of **CM_finger_data** structures in **f**. Its definition is:

```
void CM_finger_delete(CM_finger_data *f)
```

- **CM_finger_print** prints the **CM_finger_data** structure on the process's I/O stream. Its definition is:

```
 CM_finger_data *CM_finger_print(FILE *stream,
                                 CM_finger_data *f)
```

- Call **CM_finger_banner** to display the standard **CM_finger** banner on the process's I/O stream. Its definition is:

```
void CM_finger_banner(FILE *fp)
```

- **CM_finger_cm** returns a linked list of **CM_finger_data** structures that describe the state of each sequencer of the specified CM. Its definition is:

```
CM_finger_data *CM_finger_cm(char *cm_name)
```

- **CM_finger_host_cm** returns a linked list of **CM_finger_data** structures that describe the state of each sequencer of the specified CM that is attached to the specified host. Its definition is:

```
 CM_finger_data *CM_finger_host_cm(char *host,
                                    char *cm_name)
```

- **CM_finger_host_interface** returns a **CM_finger_data** structure that describes the CM that is attached to the specified interface of the specified host. Its definition is:

```
 CM_finger_data *CM_finger_host_interface(char
           *host, int interface)
```

- **CM_finger_interface** returns a **CM_finger_data** structure that describes only this front end's connection to the CM on the specified interface. Its definition is:

```
CM_finger_data *CM_finger_interface(int iface)
```

- **CM_finger_all_interfaces** returns a list of **CM_finger_data** structures that describe all the interfaces on this front end (but does not query other hosts attached to this front end). It takes no arguments.

- **CM_waiters** returns the number of users waiting for a CM resource, and prints the list to the file structure you specify. Its definition is:

```
int CM_waiters(FILE *)
```

Example

The code below uses some of these routines to print a complete finger output onto the standard output:

```
CM_finger_banner(stdout);
CM_finger_delete(CM_finger_print(stdout,
                             CM_finger_d(NULL, 0 )));
```

The CM_finger_data Structure

The **CM_finger_data** structure is defined in **<cm/cm-interface.h>** and looks like this:

```
typedef struct CM_finger_data {
         struct CM_finger_data *next;
         struct CM_finger_data *prev;
         char *cm_name;    /* the name of the CM */
         char *host;       /* the front end */
         int interface     /* the interface (on host) to the CM.*/

/*
 * if f->udata is set, then f->user, f->last_event, and f->cmd can
 * be ignored (f->user may be set to ''unknown'' or ''nobody'',
 * the others have undefined values).
 */
  CM_UDATA_LIST *udata;    /* data about the users on the CM. */
                           /* This is a list because the */
                           /* interface may be timeshared. */

        /*
         * If f->user is ''nobody'', then the cmd and last-event
         * fields are undefined.
         */
  char *user;       /* the name of the user on the CM */
  char *cmd;        /* user's command */
  long last_event;  /* time_t.tv_sec of last significant event */
                    /* set to 0 for remote machines */
  int seqs;         /* the sequencer set the user's att'd to */
                    /* according to the using_cm server*/
  int nexus_seqs;   /* the sequencer-set the user's att'd to */
                    /* according to the nexus regs (this */
                    /* value is 0 when this front-end is */
                    /* not attached to the CM in question)*/
  char *msg;        /* a message (generally, an error message) */
  long nprocs;      /* count of processors in use */
} CM_finger_data;
```

CM_UDATA_LIST is also defined in **<cm/cm-interface.h>**, and looks like this:

```
#include <cm/cmioctl.h> {
typedef    struct CM_UDATA_LIST
           struct CM_UDATA_LIST *prev;
           struct CM_UDATA_LIST *next
           char *user;
           CM_UDATA ud
} CM_UDATA_LIST;
```

CM_UDATA is defined in **<cm/cmioctl.h>**, and looks like this:

```
typedef struct cm_user_data {
  u_short ud_state;      /* State flags for this indirect device */
  short  ud_intf;        /* Associated interface (or -1 if none) */
  short  ud_uid;         /* UID of device "owner" (first opener) */
  short  ud_detach_uid;  /* UID of whoever detached us (if we were) */
  long   ud_error_csr;   /* Saved CSR in case of hard error */
  u_long ud_mark;        /* Time of day of last significant event */
  char ud_command[CM_MAX_CMD_LENGTH+1]; /* User command, and null */
} CM_UDATA;
```

ud_state can have these flag values:

```
#define US_OPEN         0x0001  /* A process is connected to this slot */
#define US_WAITING      0x0002  /* A connected process is waiting to attach */
#define US_CONNECTED    0x0004  /* We are attached to a hardware interface */
#define US_EXCLUSIVE    0x0008  /* No further opens of this device allowed */
#define US_DISCONNECTED    0x0010  /* We once were attached, but no more */
#define US_HARD_ERROR   0x0020  /* Fatal hardware error occurred */
#define US_ATTACHING    0x0040  /* Process is trying to attach */
#define US_BOOTING      0x0080  /* Process is trying to cold-boot */
#define US_BLOCKED      0x0100  /* Process is blocked on IO */
#define US_SHARED       0x0200  /* Process is sharing this device */
#define US_BEQUEATHED   0x0400  /* Process has pass on exclusive rights */
#define US_DIRECT       0x0800  /* Process has opened direct device */
#define US_READING      0x1000  /* Process is in read */
#define US_WRITING      0x2000  /* Process is in write */
#define US_CM_ERROR     0x4000  /* Error detected on suspend */
```

In all cases the linked lists are terminated by a **NULL->next** pointer.

5.6 C-Only Routines for Sequencer Information

The routines discussed in this section provide ways of converting sequencer-set bit information to printable strings and vice versa.

Use **CM_sequencer_set** to convert a printable string to a bit mask that represents the sequencer set. Its definition is:

```
int CM_sequencer_set(char *str)
```

Legal strings for sequencer sets are 0, 1, 2, 3, 0-1, 2-3, and 0-3. The error value is 0.

Use **CM_sequencer_string** to convert the bit mask to a printable string. Its definition is:

```
char *CM_sequencer_string(int s)
```

If successful, the routine returns a string constant, so that you don't have to save the result before calling it again. If a bad sequencer set is specified, the routine returns a string that reports the error, using this format:

```
Bad sequencer mask: 0xhexstring
```

This string is a static and needs to be saved.

Two macros are provided that convert between the sequencer set returned by **CM_sequencer_set** and the format required by **CM_attach_to** (see Section 5.2.2):

Use **CMA_BITS_TO_UCCS** (**ICMA_BITS_TO_UCCS** in Fortran) to convert the **CM_attach_to** format to the **CM_sequencer_set** format.

Use **CMA_UCCS_TO_BITS** (**ICMA_UCCS_TO_BITS** in Fortran) to convert the **CM_sequencer_set** format to the **CM_attach_to** format.

Thus, to convert a string typed by the user to the **CM_attach_to** format, you could use this idiom:

```
CMA_UCCS_TO_BITS(CM_sequencer_set(string))
```

5.7 C-Only Methods for Error Handling

Error messages from the CM subsystem are sent to the file pointer **`CM_error_stream`**. By default, this is bound to stderr; you can change this default if, for example, you want error messages to be sent to a file.

CM errors are handled by **`CM_panic`**, which calls the function pointer **`CM_abort_function`** as the last thing it does. By default, this generates a core dump, but you may want to define your own abort function (for example, a **`longjmp`** to a top-level handler).

`CM_abort_function` has this definition:

```
void (*CM_abort_function)()
```

It can take an argument (**`CM_panic`** calls it with an argument 0). *It must not return;* if it returns, a core dump is generated.

To use **`CM_abort_function`**:

1. Declare your error-handling subroutine. For example:

```
foo() {
         int a;
         char *b;
}
```

2. Tell Paris about the error handler:

```
CM_abort_function = foo;
```

5.8 C-Only Methods for Attaching via Command-Line Arguments

You can use the C routines **`CM_getopt`** and **`CM_attach_getopt`** to parse command-line arguments and use them to determine the characteristics of the CM resource to which the user wants to attach. To use these routines, you should be familiar with the C routine **`getopt`**, on which they are based.

`CM_getopt` and **`CM_attach_getopt`** take the standard **`getopt`** arguments: **`argc`**, **`argv`**, and an option string. In addition, they take a pointer to a string pointer, which is where the routine stores the name of a CM (if the user supplies one via the `-c` option), and an integer pointer, which is where the routine stores the bit-mask that represents the characteristics of the CM resource. (The routines ignore the values of these last two arguments—they are used to only return values.) In every case, the syntax of the option is identical to the syntax used by **`cmattach`**.

The definition of **`CM_getopt`** is:

```
int CM_getopt(int argc, char  **argv, char *optstring,
                char *cmnamestore, int *cmbitstore)
```

The definition of **`CM_attach_getopt`** is identical.

`CM_getopt` understands these command-line options:

`-c` *name*	Selects a CM by name.
`-i` *number*	Selects a CM by interface.
`-s` *seq-set*	Selects a sequencer set.

In addition to these options, **`CM_attach_getopt`** understands these options:

`-64, -32, -0`	Selects CM by floating-point type.
`-cm` *name*	Selects CM architecture.
`-D`	The CM resource must have a DataVault.
`-e`	User wants exclusive access only.
`-F`	The CM resource must have a framebuffer.
`-p` *n*	Selects the number of processors to attach to.
`-t`	User wants timeshared access only.
`-w`	User will wait for resource to become available.

All other options are passed on to the application program.

Here is a sample program that uses **`CM_attach_getopt`**. It accepts the additional option **`-P`**, which causes the program to call **`CM_preempt`** to obtain the requested resource.

```
#include <stdio.h>

char *CM_progname;
extern char **environ;

void
main (argc, argv)
int argc;
char **argv;
{
    int c;
    extern char *optarg;
    extern int optind, opterr;
    extern char *CMA_attach_usage_string;
    extern char *CM_usage_string;

    boolean preempt = FALSE;
    char *cmname = NULL;
    int cmbits = 0;

    CM_progname = argv[0];

    while((c = CM_attach_getopt(argc, argv, "P", &cmname, &cmbits)) != EOF)
      switch(c) {
    case 'P':
      preempt = TRUE;
      break;
      fprintf(stderr, "%s usage: %s [-P]\n", CM_progname,
           CM_progname);
      fprintf(stderr, "%s", CMA_attach_usage_string);
      fprintf(stderr, "%s", CM_usage_string);
      exit(1);
      }

    if(preempt? CM_preempt(cmname, cmbits): CM_attach_to(cmname, cmbits))
   if(fork() == 0) {
        if(CM_attach()) {
             int i;

             CM_exec_coldboot_paris();

             execl("/bin/ksh", "-", 0);
             fprintf(stderr, "%s: Can't execl /bin/ksh; %s\n",
                 CM_progname, errormsg);
             exit(1);
        }
        else {
            fprintf(stderr, "%s (child): Can't attach!\n",
                    CM_progname);
            exit(1);
        }
    }
    else {      fprintf(stderr, "%s (child): Can't attach!\n",

                    CM_prognam
        int status;

        fprintf(stderr, "%s (parent): waiting for child...\n",
                CM_progname);
        wait(&status);
    }
    }
    else {
    fprintf(stderr, "%s (parent): can't attach\n", CM_progname);
    exit(1);
```

Chapter 6

Programming Tools

This chapter describes tools you can use in programming the Connection Machine system. Using these tools, you can:

- Perform safety checking of a program. See Section 6.1, below.
- Time a program, or parts of it. See Section 6.2 on page 99.
- Profile a program. See Section 6.3 on page 105.
- Checkpoint a program, so that it can restart from a specified point in its execution. See Section 6.4 on page 107.
- Visualize your data, using CM graphics display software. See Section 6.5 on page 125.

The discussions in this chapter apply to C*, CM Fortran, C/Paris, and Fortran/Paris programs. (Checkpointing is not available for C*.) For information on *Lisp and Lisp/Paris, see Part V of this guide.

6.1 Run-Time Safety Checking

The CM system provides a safety utility that checks for Paris-level errors and inconsistencies in programs. This utility can be used both for Paris programs and for programs written in high-level languages; see the user's guides for the high-level languages for information on additional safety checking available for these languages. Safety checking reduces execution speed, of course, but it can be useful in developing and debugging programs.

When turned on, the safety utility checks:

- Whether field IDs passed as arguments to Paris instructions refer to fields in the current VP set
- Whether field IDs passed as arguments to Paris instructions are valid field IDs (although not all invalid field IDs are caught)
- Whether the lengths passed to Paris instructions exceed the lengths of the respective field operands

(For information on field IDs and VP sets, see the *Paris Reference Manual.*) When the utility detects an error, it aborts the execution of the program and prints information about the error to your standard error device.

There are two ways of using this utility:

- By using the Paris instruction **`CM_set_safety_mode`** from within a program
- By issuing the command **`cmsetsafety`** from within a **`cmattach`** subshell

6.1.1 From within a Program

The safety utility is available as **`CM_set_safety_mode`**, a Paris instruction you can include in your program. To turn on safety, specify any non-zero integer as an argument to the instruction. To turn it off, specify zero as the argument.

If you call this routine in a C* or CM Fortran program, you must include the Paris header file. For C*, the file is **`<cm/paris.h>`**. For CM Fortran, the file is **`/usr/include/cm/paris-configuration-fort.h`** (you may need to specify a different path for this file; check with your system administrator).

6.1.2 From a cmattach Subshell

The CM command **`cmsetsafety`** performs the same function as the Paris instruction **`CM_set_safety_mode`**. Using the command rather than the instruction lets you turn safety checking on and off for a program without changing the source file. However, **`cmsetsafety`** does not let you limit safety checking to selected parts of a program.

To turn on Paris-level safety checking, issue this command from a **cmattach** subshell:

```
% cmsetsafety on
```

You can also put the command into a script file to be executed as an NQS batch request. See Chapter 2 for a discussion of the **cmattach** subshell and batch requests.

To turn off safety checking, use the argument **off** instead of **on**. Safety checking is initially off in a **cmattach** subshell. Turning it on causes it to stay on until you turn it off or exit the subshell. See below for a way to change the default behavior of safety checking.

6.1.3 Changing the Default Safety Behavior

As mentioned above, safety checking is initially turned off for a **cmattach** subshell. To enable safety checking by default for all CM program execution (including batch requests), set the environment variable **CM_DEFAULT_SAFETY** to **on.** For example, if you are running the C shell, put this line in your **.login** or **.cshrc** file:

```
setenv CM_DEFAULT_SAFETY on
```

If the variable is not set, or if it is set to any other value, safety is off for background execution and initially off in a **cmattach** subshell.

It is often convenient to set the defaults such that safety is off for background execution (that is, when you specify the name of your program on the **cmattach** command line) but on in a **cmattach** subshell. To accomplish this in the C shell, add this line to your **.cshrc** file:

```
if ($?CMDEVICE) cmsetsafety on
```

6.2 Timing a Program

The CM system provides a timing utility that lets you determine how much time any part of a program takes to execute on the CM. The timer consists of a set of Paris instructions that you insert at the appropriate places in your program.

NOTE: C* has its own versions of these routines, which C* users may prefer to use. See the *C* User's Guide* for information.

The timing utility has the following features:

- A timer calculates total elapsed (wall clock) front-end process run time and the total amount of time the CM is active. It provides times of up to 43 hours, with microsecond precision.
- Multiple timers can be active at the same time.
- Timers can be nested. This allows you, for example, to start one timer that will time the entire program, while using other timers to determine how different parts of the program contribute to the overall time.

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

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

In C, the call would be:

```
CM_timer_start(0);
```

In Fortran, the call would be:

```
CALL CM_timer_start(0)
```

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

```
CM_timer_stop(0);
```

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

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

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

```
CM_timer_print(0);
```

CM_timer_print prints information like the following to your standard output:

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

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

- **CM_timer_read_starts** returns an integer that represents the number of times the specified timer has been started.
- **CM_timer_read_elapsed** returns a double-precision value that represents the total elapsed time (in seconds) for the specified timer.
- **CM_timer_read_cm_idle** returns a double-precision value that represents the total CM idle time (in seconds) for the specified timer.
- **CM_timer_read_cm_busy** returns a double-precision value that represents the total time (in seconds) the CM was busy for the specified timer. CM busy time is the total elapsed time minus the CM idle time.
- **CM_timer_read_run_state** returns TRUE (1 in C) if and only if the specified timer is running.

If you call any of these timing routines in a C* or CM Fortran program, include the Paris header file. For C*, this file is **<cm/paris.h>**. For CM Fortran, the file is **/usr/include/cm/paris-configuration-fort.h**. (You may need to specify a different path for this file; check with you system administrator).

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

To clear the values maintained by timer 0, call **CM_timer_clear**:

```
CM_timer_clear(0);
```

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

When you run a program that contains timer routines, the timer first prints the CM's clock speed to your standard output device before displaying any timings. For example:

```
CM speed = 6.99714 MHz
```

As mentioned above, you can have up to 64 timers active. The maximum number of timers may change in future releases. You can check the maximum number of timers as follows:

- In C, check the **extern** unsigned variable **CM_number_of_timers** in **<cm/paris.h>**.
- In CM Fortran, use the external function **CM_number_of_timers()** in **/usr/include/cm/paris-configuration-fort.h.**

6.2.1 Interpreting the Results

In interpreting the results of a timer, it is important to understand something of how the timing utility works.

The elapsed time reported by a timer includes time when the process is swapped out on the front end. The more processes that are running on the front end, the more distorted this figure will be. Therefore, we recommend the following:

- Use a front end that is as unloaded as possible.
- Run the process several times; the minimum elapsed time reported will be the most accurate.

Similar considerations apply when the process is running on a CM under timesharing. To obtain the best results, run the process on a sequencer that is not timeshared. If that isn't possible, try to run the process when no other processes are using the same sequencer. Under timesharing, elapsed time is the amount of time your process used the CM (not elapsed wall clock time).

CM idle time includes only those cycles during which the CM is waiting for an instruction from the front end. Consequently, CM active time includes not only those cycles during which the CM is performing computations, but also those during which the CM is waiting for arguments to an instruction it has received. Therefore:

- Expect slightly different CM active times on different front-end models for code segments that do not keep the CM 100 percent active. The time the CM spends waiting for data to appear is counted as active, but front-end models differ in the speed with which they can move data over the FEBI to the sequencer.

- Avoid stopping a process that is being timed.

In addition, make sure that Paris safety checking is turned off, since safety checking slows down execution of a program; see Section 6.1 on page 97.

6.2.2 An Example

The following CM Fortran program uses several features of the timing utility:

```
program timing
integer A(100), B
parameter (N=20 000)

include '/usr/include/cm/paris-configuration-fort.h'

call cm_set_safety_mode(0)   ! Turn off safety
call cm_timer_start(0)   ! Start outer timer
call cm_timer_start(1)   ! Start inner timer
A = 0
do (N) times    ! Do an operation on CM array
   A = A + 1
end do
call cm_timer_stop(1)
print *, 'CM integer array addition:'
call cm_timer_print(1)     ! Print inner timer
call cm_timer_clear(1)
call cm_timer_start(1)    ! Restart inner timer
B = 0
do (N) times  ! Do an operation on front end
   B = B + 1
end do
call cm_timer_stop(1)
print *, 'Scalar integer addition:'
call cm_timer_print(1)    ! Print inner timer
call cm_timer_stop(0)
print *, 'Total process time:'
call cm_timer_print(0)    ! Print outer timer
end
```

The program's output is shown below:

```
CM speed = 7.04633 MHz

CM integer array addition:

Starts: 1
CM Elapsed time: 9.58206 seconds.
CM busy Time: 4.74322 seconds.

Scalar integer addition:

Starts: 1
CM Elapsed time: 0.0117976 seconds.
CM busy Time: 6.67014E-06 seconds.

Total process time:

Starts: 1
CM Elapsed time: 9.72201 seconds.
CM busy Time: 4.74331 seconds.
```

Note the following about this program:

- It explicitly calls **`CM_set_safety_mode`** to turn off run-time safety checking.
- It uses one timer (timer 0) to time the entire program, and another timer (1) to time the two DO loops within the program. The first DO loop uses the CM; the second doesn't.

6.3 Profiling

You can use the UNIX **gprof** command to generate a "call graph" profile of a data parallel program. This profile displays a summary of the amount of time spent in each routine, as well as a list of which routines call, and are called by, other routines. For complete information about **gprof**, consult your UNIX documentation.

The CM system provides special Paris and CM file system libraries that you can link with your data parallel program; by linking with these libraries and using **gprof**, you can see which routines are getting called most frequently, and which are using the most time during program execution.

NOTE: The profiling utility does not provide information about usage of the CM. To obtain that information, use the timing utility described in Section 6.2.

6.3.1 Effects of Using the Profiling Libraries

In the profiling libraries, all CM calls operate synchronously; this is not the case in the normal libraries. By synchronizing the CM and the front end, the profiling libraries enable **gprof** to obtain accurate information about both the front end and CM time for CM operations. This causes some loss of efficiency in the program as a whole, however.

6.3.2 Using the Profiling Libraries

From C/Paris and Fortran/Paris

To use the profiling libraries, do this when compiling the program:

- Use the **-pg** option with the compiler command.
- Use the **-l** option to link with the Paris library **libparis-pg.a** (using the syntax **-lparis-pg**).
- If the program uses the CM I/O system, also link with the CM I/O library **libcmfs-pg.a** (**-lcmfs-pg**).
- If the program uses *Render or Generic Display Interface graphics routines, also link with the *Render library **libcmsr-pg.a** (**-lcmsr-pg**).

Do not link with the standard versions of these libraries.

You can then run the program on the CM as you normally would. When the program has run, you can use the **gprof** command to profile it, as described below.

From C* and CM Fortran

To use the profiling libraries, use the **-pg** option to the **cs** or **cmf** command when compiling the program. The compiler automatically links with the profiling libraries. You can then run the program on the CM as you normally would. When the program has run, you can use the **gprof** command as described below.

Issuing the gprof Command

Issue **gprof** with the name of the program as its argument. If you are profiling a program called **simple**, for example, issue the command as follows:

```
% gprof simple
```

The **gprof** command produces a huge amount of output, so you might want to redirect the output to a file. For example:

```
% gprof simple > simple.profile
```

To help in interpretation, the output from **gprof** contains explanations of the various parts of the profile. Note, however, that the high-level languages are compiled into Paris, and the Paris calls are included in the profile. Thus, interpretation of these profiles is difficult without an understanding of Paris.

Note in particular that you may see many routines of the type **_CMI_read_rfifo_***xx_xx* in your output. These are calls inserted by the profiling libraries to synchronize the CM and the front end. You can therefore ignore the information about these routines.

For more information on **gprof** and its options, type

```
% man gprof
```

to read the on-line manual page for **gprof**.

6.4 Checkpointing a Program

The Connection Machine system's checkpointing package lets you save an executable copy of a program's state; you can subsequently issue a command to restart execution of the program from this state. This package is especially useful for programs with long execution times, where it is important that execution does not have to start over from the beginning because of a problem with the system. You can insert any number of checkpoints in a program, and you can restart a program from a particular checkpoint any number of times.

There are three basic methods of checkpointing:

- You can insert checkpoints at particular points in a program.
- You can have checkpoints occur periodically within a program.
- With some restrictions, you can have a checkpoint occur when a program is sent a particular signal (for example, during a planned shutdown of the system).

6.4.1 Features of CM Checkpointing

The CM checkpointing mechanism has the following features:

- It is callable from C/Paris, CM Fortran, and Fortran/Paris. (Note that it is not callable from C*.)
- It can be used from within a debugger like **dbx**.
- It does not require extensive modification of a program.
- It can be used on programs that execute only on the front end, as well as on programs that use the CM.

Limitations

You cannot use the checkpointing mechanism to restore communication links and pipes unless the program includes the code to do this itself. See Section 6.4.6 on page 113 for a further discussion of this issue.

If you initially run a program under timesharing, you must restart the checkpointed version under timesharing. Likewise, if you run the program in exclusive mode, you must restart it in exclusive mode.

6.4.2 Overview of CM Checkpointing

Programming a Checkpoint

You checkpoint a program by inserting calls to checkpointing routines in your program. These routines are listed in Table 10 and are described in detail in later sections.

Table 10. Checkpointing routines

Routine	Use
`ckpt`	Checkpoints a program.
`ckpt_hook_set`	Adds a routine to the list of routines to be executed during checkpointing.
`ckpt_hook_delete`	Deletes a routine from the list to be executed during checkpointing.
`ckpt_init`	Initializes the checkpointing package.
`ckpt_periodic`	Calls **`ckpt`** if the checkpoint bit is set.
`ckpt_periodic__start`	Starts the timer for periodic checkpointing; sets the checkpoint bit at the end of the period.
`ckpt_periodic_stop`	Stops the timer for periodic checkpointing.
`ckpt_periodic_with_return_value` (C only)	Calls **`ckpt`** if the checkpoint bit is set; otherwise returns.
`ckpt_print_error`	Prints an error message. (Fortran only)
`ckpt_restart`	Restarts a checkpointed program; for use with a debugger.

The Checkpoint Files

When a checkpoint occurs, the checkpointing package saves the state of the program in the following files:

- **`-core`** — This is a standard core file, containing the state of the program on the front end.
- **`-cm-core`** — This file contains the state of the program on the CM. This file is not created if the program is not using the CM.
- **`-file-list`** — This is a list of the files that the program had open when it was checkpointed.
- **`-program`** — This is a stored copy of the checkpointed program.

These files have prefixes added to them to create complete pathnames. The prefixes are specified by the routine that executes the checkpoint. The **`-core`**, **`-file-list`**, and **`-program`** files all share the same prefix, which specifies a path in the front-end file system; this is referred to as the *front-end prefix*. For example, if the routine specifies the prefix **`/jones/myprog`**, then the front-end core file is stored in **`/jones/myprog-core`**, and the list of I/O files is stored in **`/jones/myprog-file-list`**.

The **`-cm-core`** file is stored in the CM file system. Typically, its prefix would specify a pathname for a DataVault: for example, **`dva:/jones/myprog`**; this is referred to as the *CM prefix*. (See Chapter 7 for information on how to specify a DataVault pathname.)

Compiling a Program Containing Checkpoints

When compiling a program that contains checkpoints, link with the **`ckpt`** and **`cmfs`** libraries, in that order. For example:

```
% cmf myprog.fcm -lckpt -lcmfs
```

Restarting a Checkpoint

To run a checkpointed version of a program, use the CM command **`restart`**. For example,

```
% restart /jones/myprog
```

This restarts execution of the checkpointed version of the program **`myprog`**, using the front-end prefix **`/jones/myprog`** to identify the files that contain the checkpoint. (It obtains the CM prefix, if any, from the program being restarted.) For more details, see Section 6.4.13 on page 121.

6.4.3 Include Files for the Checkpointing Package

C Programmers: To use the CM checkpointing package in a C/Paris program, include the file **`<cm/ckpt.h>`** in your program. If your program does not contain code executed on the CM, include the following **`#define`** before including the checkpointing file:

```
#define CKPT_SKIP_CM_CODE
```

This loads a front-end-only version of the package, thereby avoiding the overhead of linking in the entire Paris library.

Fortran Programmers: To use checkpointing in a Fortran/Paris or CM Fortran program, include the file **/usr/include/cm/ckpt-fort.h**.

6.4.4 Initializing the Checkpointing Package

To initialize the checkpointing package, call the routine **ckpt_init**; *you must do this before calling any other checkpointing routine, and before any parallel operation or CM function.* Its only argument is the name of the program being run.

C Programmers: You can generally specify the name of the program in the **ckpt_init** call as follows:

```
main(argc, argv) int argc; char **argv;
{
     /* ... */
     ckpt_init(argv[0]);
     /* ... */
}
```

Fortran Programmers: You must include the name of the program as it is to be invoked by the user. For example, if the program is to be invoked as follows:

```
% simple
```

then you would call **ckpt_init** in the program as follows:

```
call ckpt_init("simple"//char(0))
```

NOTE: It is the responsibility of the Fortran programmer to ensure that all strings passed to the checkpointing package are null-terminated.

What ckpt_init Does

When the program first runs, this routine initializes the checkpointing package; the routine's arguments are examined, but nothing is done with them. If the environment variables **CKPT_ENV_FEPREFIX** and **CKPT_ENV_CMPREFIX** exist, however, **ckpt_init** restarts a checkpoint instead of initializing the package. These environment variables are created by the **restart** command. **ckpt_init** uses the prefixes stored in these environment variables to determine the pathnames of the files to use in restarting the checkpoint.

When **ckpt_init** successfully restarts a checkpoint, it does not return; instead, the program restarts at the return from the invocation of the **ckpt** routine that generated the checkpoint; see the next section. When restarting the checkpoint, **ckpt_init** prints a warning on the standard error device for any file that was open read-only, but that has changed since the checkpoint was created. Other errors and warnings are also printed on the standard error device.

6.4.5 Putting a Checkpoint in a Program

Use the **ckpt** routine to insert a checkpoint at a particular point in a program.

This routine creates a checkpointed version of the program in files with prefixes specified by the arguments to the routine. See below for the syntax in C and Fortran. See Section 6.4.2 on page 108 for more information on the files created by a checkpoint.

If the program checkpoints repeatedly using the same prefixes, **ckpt** ensures that all checkpoint data is safely stored to disk before removing the old checkpoint.

The call to **ckpt** must appear after the Paris call **CM_init** that initializes the CM hardware. In C/Paris and Fortran/Paris, this call is made explicitly in the program. In CM Fortran, the call appears in the object file produced by the compiler. In this language, it should be safe to insert the **ckpt** call after the first parallel operation in the program. If you're in doubt, however, check the object file.

In C

The **ckpt** routine has the following definition in C:

```
int ckpt(char *feprefix, char *cmprefix)
```

where:

feprefix	is the front-end prefix, which is added to the front-end core file, the file list, and the checkpointed program created by this checkpoint.

cmprefix is the CM prefix to add to the CM core file so that it is stored in the CM file system. If you omit a hostname, the default host is used; see Chapter 7 for a discussion of the default host. If the CM is not used, specify **NULL** for this argument.

See "Return Values," below, for a description of return values from **ckpt**.

In Fortran

You can use the syntax shown in this code fragment when calling the **ckpt** routine from CM Fortran or Fortran/Paris:

```
include '/usr/include/cm/ckpt-fort.h'

integer ckpt_result

[...]

ckpt_result = ckpt("fepref"//char(0), "cmpref"//char(0))
if(ckpt_result .lt. 0) then
   call ckpt_print_error()
end if
```

In this code fragment, **ckpt_print_error** prints an error message on standard error; see Section 6.4.10 on page 119.

In the call to **ckpt**, *fepref* is the front-end prefix to be add to the front-end core file, the file list, and the program created by this checkpoint, and *cmpref* is a prefix to add to the CM core file so that it is stored on a DataVault. If the CM is not used, specify **NULL** for this argument.

Return Values

The return values from the **ckpt** routine are:

-1	The checkpoint failed. The routine stores a detailed explanation of the failure in the string pointer **ckpt_errormsg**; see Section 6.4.10 on page 119.
0	The checkpoint succeeded.
1	The routine is returning from a restarted checkpoint.

6.4.6 Calling Routines to Execute as Part of a Checkpoint

In the process of creating or restarting a checkpoint, it may be necessary to call other routines to perform such tasks as

- reinitializing output windows or network connections (for example, X Window System connections)
- reattaching to databases or shared-memory segments (note, however, that reattaching to shared memory segments is done automatically on the Sun-4)
- updating environment variables

If such routines are to be used in creating a checkpoint, you can call them explicitly before the call to **ckpt**. If they are to be used in restarting a checkpoint, you can call them explicitly if **ckpt** returns 1.

The CM checkpointing package provides another mechanism for calling these routines, however. This "hook" mechanism, described below, is especially useful for library packages that have to take special measures to save or restart their state when checkpointed. By using it, library authors can avoid burdening library users with the need to know about these routines.

The Checkpoint Hook Mechanism

To call a routine to be executed as part of checkpointing, make it an argument to the **ckpt_hook_set** routine.

This routine has the following definition for C:

```
ckpt_hook_set(int (*hook_fn)(), ckpt_hook_id id,
    int order_hint,*char arg)
```

In Fortran, call the function as follows:

```
call ckpt_hook_set(hook_fn, id, order_hint, arg)
```

where:

hook_fn	is the name of the function to call.
id	is either **CKPT_SAVE_HOOK** (0 in Fortran) or **CKPT_RESTART_HOOK** (1 in Fortran). If you choose **CKPT_SAVE_HOOK**, the function is called as part of the checkpointing pro-

cess. If you choose the argument **CKPT_RESTART_HOOK**, it is called as part of restarting a checkpoint.

order_hint is an integer that the checkpointing package uses in determining the relative order of execution of hook functions. "Save hooks" are executed in decreasing order of *order_hint* values; "restart hooks" are executed in increasing order of *order_hint* values. If more than one function has the same *order_hint* value, no guarantee is made about the order of execution.

arg is an argument passed to the hook function when it is called. This lets you call the same hook function repeatedly with different arguments. In C, this argument is passed as a pointer.

Order hints are used to determine the relative order of execution between library packages. For example, one library that depends on another can require that the other library's restart hook be executed before its own restart hook.

The routine returns 0 if it successfully added the function to the list of checkpoint hooks, and -1 if an error occurred. A description of the error is stored in the string pointer **ckpt_errormsg**; see Section 6.4.10 on page 119.

To delete a function from the list of checkpoint hooks to be called as part of checkpointing, use the routine **ckpt_hook_delete**. This routine has the following definition for C:

```
ckpt_hook_delete(int (*hook_fn)(),
    ckpt_hook_id id, char *argument)
```

Call it from Fortran as follows:

```
call ckpt_hook_delete(hook_fn, id, argument)
```

where *hook_fn* is the hook function, as described above; *id* is either **CKPT_SAVE_HOOK** or **CKPT_RESTART_HOOK** (0 or 1 in Fortran); and *argument* is the argument that was passed to the hook function when it was invoked. In C, *argument* is a pointer.

The function returns 0 if it deleted the specified function from the list of checkpoint hooks, and -1 if an error occurred. A description of the error is stored in the string pointer **ckpt_errormsg**; see Section 6.4.10. The most probable error is that the function wasn't found in the list of functions.

6.4.7 Setting Up Periodic Checkpoints

The CM checkpointing package provides routines that let you essentially automate the checkpointing of a program. You can also use these routines to save a checkpoint in response to a signal; see Section 6.4.8 on page 117.

Setting the Period

After initializing the checkpointing package, you can set up a checkpointing period by issuing a call to **`ckpt_periodic_start`**. This routine requests a SIGALRM signal at specified intervals. When the signal arrives, a bit is set indicating that it is time to do a checkpoint.

The routine has this definition in C:

```
int ckpt_periodic_start(int H, int M)
```

Call it from Fortran as follows:

```
call ckpt_periodic_start(H, M)
```

where *H* and *M* specify hours and minutes. A SIGALRM is to be sent after *H* hours and *M* minutes. The timer will then be reset, and another period of the same length will begin.

Note that a checkpoint does not occur automatically at the end of the period; instead, a bit is set, which signals that it is time for a checkpoint. You can then check this bit as described in "Performing the Checkpoint," below. The checkpoint can't occur automatically because a Paris instruction may consist of several messages from the front end to the CM. A signal can arrive during these messages. If a checkpoint occurs in response to the signal, the new Paris instructions that it generates can end up being inserted as data to the Paris instruction that was interrupted by the signal. The mechanism of setting and checking a checkpoint bit ensures that the checkpoint occurs between Paris instructions.

Note that currently there is no way to change the period—that is, to have one time period between checkpoints in one part of the program, and another time period in another part of the program.

The routine returns the value of the old SIGALRM handler, or -1 if an error occurred.

Call **ckpt_periodic_end** (with no arguments) to turn off the SIGALRM signal and discontinue its use for triggering a checkpoint.

Performing the Checkpoint

There are two routines you can use to check the bit set by the **ckpt_periodic_start** routine: **ckpt_periodic** (a C preprocessor macro) and **ckpt_periodic_with_return_value** (a subroutine—not supported in CM Fortran and Fortran/Paris).

The definition of **ckpt_periodic** in C is:

```
void ckpt_periodic(char *feprefix, char *cmprefix)
```

The definition of **ckpt_periodic_with_return_value** in C is:

```
int ckpt_periodic_with_return_value(char
   *feprefix, char *cmprefix)
```

where the arguments and the return value are the same as for the **ckpt** routine.

If the checkpoint bit is set, **ckpt_periodic** calls **ckpt** to checkpoint the program, then clears the bit. The **ckpt** routine creates a checkpointed version of the program in files with prefixes specified by *feprefix* and *cmprefix*; **ckpt** always returns with the bit cleared, either from a restart or from a checkpoint. If the bit is not set, nothing happens.

ckpt_periodic_with_return_value also checks the bit and calls **ckpt** if it is set. In addition, it returns the return value from **ckpt**, or -2 if no call to **ckpt** was made.

ckpt_periodic is preferable if you don't care about the return value from the **ckpt** routine, since it requires fewer instructions than the **ckpt_periodic_with_return_value** routine.

Fortran Programmers: ckpt_periodic is a subroutine call in Fortran. To avoid the overhead of a subroutine call, use code like this (the variable must be declared as COMMON):

```
if(_ckpt_periodic_request_bit .ne. 0) then
   return_value=ckpt("fepref"//char(0),"cmpref"//char(0))
end if
```

ckpt_periodic_with_return_value is not supported in Fortran. If you want a return value from a periodic checkpoint, use the call to **ckpt** shown above, since **ckpt** returns a value.

Here is a code fragment that sets up a periodic checkpoint in Fortran; the period is one hour.

```
include '/usr/include/cm/ckpt-fort.h'
integer return_value
  [...]
call ckpt_init("prog_name"//char(0))
call ckpt_periodic_start(1,0)
  [...]
if(ckpt_periodic_request_bit .ne. 0) then
     return_value=ckpt("x"//char(0),"y"//char(0))
     if(return_value .eq. -1) then
            ckpt_print_error()
            stop 'stopped on failed checkpoint'
     else if (return_value .eq. 0) then
            stop 'stopped on successful checkpoint'
     end if
end if
```

6.4.8 Checkpointing in Response to a Signal

The **ckpt_periodic** and **ckpt_periodic_with_return_value** routines can also be used to respond to a SIGTERM or other signal from the kernel during an orderly shutdown of the CM.

Use the routine **ckpt_periodic_request** (without any arguments) in a signal handler to explicitly set the checkpoint bit. The next call to **ckpt_periodic** or **ckpt_periodic_with_return_value** then sees that this bit is set and checkpoints the program.

For this checkpointing mechanism to work during a shutdown of the CM, you must include frequent calls to either **ckpt_periodic** or **ckpt_periodic_with_return_value**, so that the program has enough time to checkpoint before the system comes down. We recommend that these routines be called at least once a minute if you are using them for this purpose.

C Programmers: The following code fragment is a signal handler in C that uses the **ckpt_periodic_request** mechanism:

```
signal_handler(){
    printf("Requesting checkpoint...\n");
    ckpt_periodic_request();
}
```

(Note that you can't call the **ckpt** routine in this signal handler, for the reason discussed in "Setting the Period" on page 115.)

Somewhere early in your program include the following code to specify which signals this routine handles (this example calls it to handle SIGTERM and SIGINT):

```
signal(SIGTERM, signal_handler);
signal(SIGINT, signal_handler);
```

At the top of your program include this line:

```
#INCLUDE <sys/signal.h>
```

6.4.9 Displaying Progress Reports

It can take several minutes to checkpoint a program or to restart a checkpoint. The checkpointing package provides a user-visible flag, **ckpt_verbose**; setting this flag to any non-zero value causes the checkpointing package to print progress reports while it is checkpointing or restarting. The flag is turned off by default. To turn it on, put lines like those shown below anywhere in your program.

In C/Paris:

```
ckpt_verbose = 1;
```

In Fortran/Paris or CM Fortran:

```
ckpt_verbose = 1
```

When checkpointing the program, the checkpointing package then prints messages like these on the program's standard error:

```
myprogram(ckpt/saving a checkpoint): copying
executable to /jones/myprogram-program.
myprogram(ckpt/saving a checkpoint): writing
CM state to dvvax:/jones/myprogram-cm-core.
```

When restarting the program, the checkpointing package prints a message like this:

```
myprogram(ckpt/restarting a checkpoint): restoring
CM state from dvvax:/jones/myprogram-cm-core
```

6.4.10 Errors

In C, the string pointer **ckpt_errormsg** contains a printable explanation of any error encountered by routines in the checkpointing package. The pointer is initialized to NULL.

For Fortran programs, call the routine **ckpt_print_error** to print the error message on your stderr.

If There Is a Timing Problem with Core Files

It is possible that the checkpoint will fail to create the front-end or CM core file; when you restart the program, you might then inadvertently use a core file that is older than the one you expected. The checkpointing package can detect this problem. When the package is finished with a checkpoint, in addition to checking the results of the write and file creation operations, it checks the creation time of the core files against the time the checkpoint began. If there is a discrepancy, it prints a message on stderr. The message points out, however, that a discrepancy of only a few seconds may be caused by the clock on the DataVault (for the CM core file) or an NFS file server (for a front-end core file) being slightly slower than the clock on the front end on which the program is running.

6.4.11 Debugging

You can do checkpointing from any object-code debugger, such as **dbx**. Call `ckpt` to generate a checkpoint; use as arguments a front-end prefix and a CM prefix, just as you do for `ckpt`. For example,

```
(dbx) call ckpt("/fetest", "dvvax:/cmtest")
```

checkpoints the program you are debugging, using the specified prefixes for the checkpoint files. Specify `NULL` for the CM prefix if the CM is not used.

Call the routine `ckpt_restart` to restart an earlier checkpoint. `ckpt_restart` requires only a front-end prefix as an argument; the CM prefix is restored with the rest of the program data when the program is restarted. For example,

```
(dbx) call ckpt_restart("/fetest")
```

restarts execution of the checkpointed version of the program, using the specified prefix to identify the checkpoint files to be used.

Breakpoints are preserved around these calls within a single **dbx** session, but a `ckpt_restart` executed in one **dbx** session does not restore the breakpoints from an earlier **dbx** session.

Note that you should *not* use this syntax to restart the checkpoint:

```
(dbx) print ckpt_restart(feprefix)
```

dbx notices the process exiting in the former case. In the latter case, the process does not return as **dbx** expects, and it doesn't leave a return value for **dbx** to examine; this may cause a memory fault in **dbx**.

6.4.12 Programming Hints

When writing a program that uses the CM checkpointing package, keep the following points in mind, in addition to suggestions made in previous sections:

- Be careful when using shared memory. Either simultaneously checkpoint all programs sharing the memory, or ensure that any changes to the data contained in shared memory will not be important.

- Checkpointing requires that two files be open at the same time. Therefore, make sure your program has at least two unused file descriptors to use the checkpointing package. (The program needs more if it uses the CM file system.)
- The checkpointing package knows the names of files that the program explicitly opens. It does not know the names of redirected standard input or output, nor the names of sockets and pipes.
- The entire contents of a program's memory is saved—including, for example, the random number seed.

6.4.13 Running a Checkpointed Program

Use the **restart** command to run a checkpointed version of a program. The syntax is:

```
restart feprefix
```

where *feprefix* is the front-end prefix used for naming the front-end core file and the list of I/O files. (**restart** obtains the CM prefix, if any, from the checkpointed program.) The front-end prefix is program-dependent; it can be hard-coded in the program or, in C, specified by a command-line argument when the program is first executed.

As mentioned earlier, if you initially run a program under timesharing, you must restart the checkpointed version under timesharing. If you initially run the program in exclusive mode, you must restart it in exclusive mode.

If you attached using a **cmattach** command, or if your program contains a **CM_attach_to** routine to specify the CM resource to which it is to attach, you must use the **cmattach** command to attach to the proper CM configuration before restarting the program. The number of processor, the memory size, and the type of floating-point unit (if any) must all be the same; in addition, if your program needs access to a DataVault or framebuffer, attach to a resource that is connected to one.

If you do not issue a **cmattach** command before restarting, **restart** will do an attach for you. Use this option only if all the CM resources to which you could be attached are identical to the one on which the program was initially run (for example, different sequencers on the same CM).

When you issue the **restart** command, the program begins execution from the point at which the files specified by *feprefix* were saved.

Note the following in using **restart**:

- It can take up to several minutes to restart a checkpointed program, depending on the size of the files. If a flag is set in the program, progress reports like this are displayed:

  ```
  /jones/myprog(ckpt/restarting a checkpoint):
  restoring CM state from
  dvvax:/jones/myprog-cm-core
  ```

  For information on this flag, see Section 6.4.9 on page 118.

- **Changing I/O files.** Checkpointing does not preserve the contents of I/O files; it is up to you to make sure these files haven't changed. If the files do change between the time the file is checkpointed and the time you restart the checkpoint, the results are unpredictable.

  The checkpointing package cannot detect changes to files that the program has open for writing. It does detect changes to files that are being read, and it reports such changes on the standard error device; it may, however, report a change when none has occurred.

- **Overwriting an I/O file.** If a file is open for writing, the restarted program continues output to that file at the point at which the checkpoint took place. This overwrites changes to the file that may have been made subsequently.

- **Output redirection**. Be careful when using output redirection with **restart**. You must *append* the output to the output file, not just redirect it; otherwise, **restart** overwrites what is already in the file. For example, a program that you originally executed as follows:

  ```
  % program_name [...] > myoutput
  ```

  must be restarted as follows:

  ```
  % restart feprefix [...] >> myoutput
  ```

- **Input redirection.** You can restart a checkpoint using the same input file used when the program was originally executed. However, the checkpointing package does not keep track of the name of this file (the string **#stdin** appears in the file-list file to specify this file). You can, if you like, substitute the real name of the file in the file-list file, since the checkpointing package does not read the file-list file.

- **UNIX pipelines.** You cannot put programs containing checkpoints in a UNIX pipeline. The UNIX kernel buffers data in the pipes, and this data is not saved by the checkpoints.

- **Restarting a checkpoint more than once.** You can restart a checkpoint more than once simply by renaming the checkpoint files and issuing the `restart` command with different prefixes as its arguments. If you move the checkpoint files to another directory, make sure that files used by the program are accessible from this directory with the same names they had when opened by the original invocation of the program.

- **Re-running the original program.** You can change a program, compile the new version, and execute it without affecting a checkpointed version of the program. Once again, however, files used by the program must not have been changed.

In a Debugger

To restart a previously checkpointed program in a debugger like **dbx**, follow these steps:

- Set the environment variable **CKPT_ENV_CMPREFIX** to the CM prefix, and set the environment variable **CKPT_ENV_FEPREFIX** to the front-end prefix. For example, if you are using the C shell, issue commands like these:

```
% setenv CKPT_ENV_CMPREFIX dvvax:/cmtest
% setenv CKPT_ENV_FEPREFIX /fetest
```

- Invoke the debugger for the program as you normally would, then run the program within the debugger as you normally would. For example:

```
% dbx myprog
(dbx) run
```

  If the environment variables are set, the debugger will run the checkpointed version of `myprog`.

- After you are finished in the debugger, be sure to "unset" these environment variables; in the C shell, you do this with the `unsetenv` command. If you don't unset them, and you try to run a program with a `ckpt_init` call in it, the program won't run; instead, `ckpt_init` will try to restart a checkpointed version of the program.

6.4.14 Sample Program

The simple CM Fortran program **ftest.fcm** shown below includes checkpointing routines. The program creates arrays on the front end and the CM, fills them, checkpoints, and then makes sure that the checkpointing and subsequent restarting didn't affect the contents of the arrays.

Remember that you must compile the program using the options **-lckpt** and **-lcmfs** (in that order), so that the program is linked with the checkpointing and CM file system libraries.

```
C   A test program for checkpointing.
C   Call checkpointing from fortran
    program ftest
    parameter (ncm = 65536, nfe = 1000)
    integer a_cm(ncm), a_fe(nfe)
    integer cm_errors, fe_errors, total_errors
    integer ckpt
    integer ckpt_result

    include '/usr/include/cm/ckpt-fort.h'

    call ckpt_init("ftest"//char(0))

    cm_errors = 0
    fe_errors = 0
    total_errors = 0

    print *, 'initializing cm array'
    a_cm = 0    ! Do an array operation to put it on the CM
    do i = 1, ncm      ! Give it some values
       a_cm(i) = i
    enddo

    print *, 'initializing front end array'
    do i = 1, nfe        ! Give the array some values
       a_fe(i) = i
    enddo

    print *, 'calling ckpt'
    ckpt_result = ckpt("ftestckpt"//char(0), "ftestckpt"//char(0))

    if (ckpt_result .lt. 0) then
       print *, 'ckpt failed, result ', ckpt_result
           call ckpt_print_error()
    endif
           print *, 'ckpt returned, checking CM data...'
```

```
do i = 1, ncm ! Check that array has correct contents
   if(a_cm(i) .ne. i) then
  print *, 'a_cm(', i, ') is ', a_cm(i)
      cm_errors = cm_errors + 1
   endif
enddo
print *, cm_errors, 'CM errors; checking FE data...'

do i = 1, nfe
   if(a_fe(i) .ne. i) then
      print *, 'a_fe(', i, ') is ', a_fe(i)
      fe_errors = fe_errors + 1
   endif
enddo

print *, fe_errors, 'errors on the front end'
total_errors = fe_errors + cm_errors
print *, total_errors, 'total errors'

end program ftest
```

6.5 Visualizing Data

The CM system provides software tools you can use to graphically display results of data parallel programs. There are two basic ways of displaying CM data:

- On a color monitor attached to a CM via a framebuffer I/O module—the combination of the monitor and the framebuffer I/O module is usually referred to simply as the *framebuffer*
- On a workstation that supports the X Window System, Version 11 interface (referred to as an *X11 window*), either locally or over a network

The framebuffer provides higher resolution and faster display, but you must be attached to a CM sequencer that contains a framebuffer module in order to use it. With the X Window System interface, you can display your output on any standard graphics workstation, either local or remote. You can also use the X interface to develop software on your local workstation that you can later display on a framebuffer.

The following software is available for visualization and graphic display:

- **Generic Display Interface.** The Generic Display Interface provides a single user interface to all CM framebuffers and X Window System servers available from your workstation. It allows an application to present a menu of the available displays. Its routines support the creation, initialization, and selection of a display, writing to and reading from the display, and the control of display offset parameters and color maps. With the appropriate image data, these routines let you visualize data through an X window when a framebuffer is not available, or to preview an image locally during development and then easily switch to the framebuffer for final revisions and viewing.

  The Generic Display also provides routines to support interactive applications through mouse support and text display. The mouse routines let you use your workstation's mouse to control and respond to a cursor on either the CM framebuffer or an X window. With the text routines you can label your image or prompt the user by displaying text strings on either generic display.

- ***Render.** *Render (pronounced "star-render") is a CM library containing routines that support graphics processing on the CM. Using these routines, you can draw simple graphics primitives that are placed in a buffer field in CM memory. You can then transfer this image to a framebuffer or X window for display. *Render is intended as a building block for more advanced visualization tools.

  *Render graphics math utilities help you create, manipulate, and transform coordinate vectors and matrices and to convert color vectors between different color spaces (for example, HSV or CMY to RGB).

  *Render dithering routines make it possible to move your image to displays with different color capabilities. These routines convert color (RGB) images to grayscale images and grayscale images to black and white images.

- **Image File Interface.** The Image File Interface lets you store images created on the CM in image files for later processing or display. These files are created in the TIFF format, a standard image file format that is widely supported by other display systems and software. You can move CM images in TIFF files into other graphics environments and read TIFF images created elsewhere into the CM system.

For complete information on CM visualization tools, see the volume *Connection Machine Visualization Programming* in the Connection Machine documentation set.

Part III

I/O on the Connection Machine System

Chapter 7

Using the CM File System

The Connection Machine system has a file system associated with it. This file system is separate from the front end's file system, although it is similar to a UNIX file system. You can use the CM file system (CMFS) to store data for the CM.

There are user commands associated with the CM file system. These commands let you perform typical tasks such as copying, moving, and deleting files, and making, deleting, and listing the contents of a directory. This chapter describes the CM file system and how to use these commands.

You create files in the CM file system either by using one of these commands to copy in data or a file from another file system, or by issuing library calls from within a program.

For information on I/O library calls that use the CM file system, consult the *CM I/O System Programming Guide.*

7.1 Overview of the CM File System

It is easiest to understand the CM file system by comparing it with the UNIX file system.

7.1.1 Similarities to the UNIX File System

Here are some of the similarities between the CM file system and the UNIX file system:

- The CM file system has a hierarchical directory structure, in which directories contain subdirectories and files, and directories and files are identified by unique pathnames. A directory has an owner, a group, and a mode. A user has a current working directory.
- Many of the user commands have counterparts in UNIX; the CMFS commands are similar in name and function to their UNIX counterparts. For example, the CMFS command **cmls** is comparable to the UNIX command **ls**.
- As in UNIX, an I/O device in the CM I/O system (except for CM-HIPPI) is simply a file in the CM file system, as far as the user interface is concerned. For example, you write data from the CM to a file in the same way, whether the "file" is really a file stored on disk or is in fact a tape drive.

7.1.2 Differences

This section lists the differences between the CM file system and the UNIX file system.

More Than One Directory Tree

Unlike a UNIX file system, a CM file system can contain more than one directory tree, each with its own root directory. Different components of the CM I/O system have their own directory trees. You distinguish among them by beginning a pathname with the component's hostname, followed by a colon; for example:

```
dva:/project/data
```

Obtain the hostnames for the components of your CM I/O system from your system administrator.

Thus, two CMFS files could have the same pathname, except for different hostnames: **dva:/project/data** could exist on one DataVault, for example, and **dvb:/project/data** could exist on another. You can use an environment variable to set a default hostname; see Section 7.5 on page 137.

As mentioned above, you have a current working directory in the CM file system; you do not have one per directory tree. You can use an environment variable to set the current working directory; see Section 7.5.

Since the colon (:) is required as part of the hostname, you cannot use a colon as part of a filename.

As a component of the CM I/O system, the front end can have its own directory tree within the CM file system; check with your system administrator to see if your front end does. The CM file system is logically separate from the UNIX file system in this case. For example, you have separate working directories within each file system.

Parallel and Serial Formats

There are two formats for files in the CM file system: parallel and serial. A parallel file consists of many streams of data, one per CM virtual processor; the file also reflects the size and shape of the data set. A serial file is a single stream of data. In general, a file must be parallel to be used by the parallel processing unit; a file must be serial to be used on a front end or other serial machine.

The basic way of creating a parallel file is by including a library call in a program to write data from the parallel processing unit to a file; the resulting file is automatically in parallel format.

7.2 Overview of CMFS User Commands

The rest of this chapter discusses user commands that operate on files and directories of the CM file system. See the *CM I/O System Programming Guide* for reference descriptions of these commands. Table 11 lists the commands.

7.2.1 CMFS Commands and UNIX Commands

As the table shows, many of the CMFS commands are simply UNIX commands with "**cm**" in front of them. In general, a CMFS command performs the same function as its corresponding UNIX command.

CMFS commands and UNIX commands are not interchangeable, however. CMFS commands operate *only* on files in the CM file system (except when copying a UNIX file into the CM file system); they have no effect on files in the UNIX system. Likewise, UNIX commands have no effect on files in the CM file system.

Table 11. CMFS user commands

Command	Use
`cmchgrp`	Change a file's group ownership.
`cmchmod`	Change a file's permissions mode.
`cmchown`	Change a file's owner.
`cmcp`	Copy files within the CM file system using serial I/O to a file server computer.
`cmdd`	Copy and convert data.
`cmdf`	Display free and used disk space.
`cmdu`	Summarize disk usage.
`cmdump`	Back up files to tape.
`cmfind`	Find files.
`cmftp`	Transfer files between DataVaults and remote systems.
`cmln`	Make links to files or directories.
`cmls`	List contents of a directory.
`cmmkdir`	Make a directory.
`cmmknod`	Make a CM character-special file.
`cmmv`	Move (rename) files or directories.
`cmrestore`	Extract files from a tape archive.
`cmrm`	Remove (unlink) files or directories.
`cmrmdir`	Remove (unlink) an empty directory.
`cmstat`	Print status information about a file.
`cmtar`	Archive tape (or other media) file.
`cmtruncate`	Truncate or extend a CM file.
`copyfromdv`	Copy files in the CM file system to the front-end file system.
`copytodv`	Copy files in the front-end file system to the CM file system.
`dvcp`	Copy files within the CM file system using the CM.

7.2.2 Where You Can Issue the Commands

You can execute all CMFS commands (except **cmftp**) on a front end, just as you would execute UNIX commands. One of the commands, **cmls**, can also be executed from a Lisp environment. Also, you must be attached to the CM to issue the **dvcp** command. The commands can also be executed from a DataVault file server computer; typically, only the system administrator would use this computer.

If your CM system has a VMEIO host computer as part of the I/O system, you can also issue commands from it. This is particularly useful when the command deals with an I/O device attached to this computer. You must have an account on the computer to issue commands from it; check with your system administrator if your commands don't work.

The easiest way to issue commands on the VMEIO host is to use the UNIX command **rsh** from the front end to open a remote shell on the VMEIO host; then issue the CMFS command on the **rsh** command line. For example,

```
% rsh vme1 cmls
```

issues the command **cmls** on the remote computer with hostname **vme1**.

There may be other machines available on your system, not front ends or DataVault file server computers, from which CMFS commands can be executed. Check with your system administrator.

7.3 Copying Files and Data

Often users have data that they need to bring into the CM file system for processing. The data may or may not be in a standard UNIX file. This section describes CMFS commands you can use to transfer data between the outside world and the CM file system. It also describes how to copy files within the CM file system.

7.3.1 Copying Files between the Front-End File System and the CM File System: The copytodv and copyfromdv Commands

In the most straightforward situation, you have a file in your front end's UNIX file system, and you want to copy it into the CM file system. To do this, use the **copytodv** command. For example, to copy the UNIX file **mydata** (in your working directory on the front end) to the file **mydata1** on the DataVault with hostname **dva**, issue the following command:

```
% copytodv mydata dva:mydata1
```

The copying takes place via the Ethernet connection between the front end and the DataVault. This means you do not have to be attached to the CM to issue the command; it also means that execution may be relatively slow.

The UNIX file that you copy may be in either parallel or serial format (it could be in parallel format if it had been previously copied into the UNIX file system from the CM file system). If it is in serial format, it is copied in serial format, and it must be transposed to parallel format before it can be processed by the CM; see "Parallel and Serial Formats" on page 131.

Use the **copyfromdv** command to copy a file from the CM file system to a UNIX file system. For example,

```
% copyfromdv dva:mydata1 mydata2
```

copies the file **mydata1** on the device named **dva** back to your working directory in UNIX, and names the file **mydata2**.

NOTE: If the file was in parallel format in the CM file system, it will be unusable in the UNIX file system. If you want to use the file, you must transpose it to serial format before copying; see "Parallel and Serial Formats."

If the UNIX File System Isn't on the Front End

If the UNIX file you want to copy isn't on your front end, but is reachable by network, you have a couple of choices:

- Use the UNIX command **rcp** to copy the file to your front end, then use **copytodv** to copy it into the CM file system from there.
- If the CM I/O system software has been loaded on the networked computer and you have an account on the computer, you can use the UNIX **rsh** command, followed by **copytodv**, to copy the file directly; see Section 7.2.2 on page 132.

7.3.2 Copying Files to and from a Tape Archive: The cmdump, cmrestore, and cmtar Commands

The **cmtar** command corresponds to the UNIX **tar** command. You can use it to write files to or read files from an archive of files stored on magnetic tape.

Most users, however, will find it easier to use **cmdump** and **cmrestore**, which provide interactive user interfaces to **cmtar**. Use **cmdump** to back up CMFS files to tape. Use **cmrestore** to extract files from the tape archive and place them into the current CM file system.

For complete information on **cmtar**, **cmdump**, and **cmrestore**, see their reference descriptions in the *CM I/O System Programming Guide*.

As with **copytodv** and **copyfromdv**, note the following:

- **cmtar** does not change the format of the files it copies; if necessary, you must explicitly transpose a file's format from parallel to serial or vice versa while it is in the CM file system. See "Parallel and Serial Formats" on page 131.
- The copying does not involve the CM, so you do not need to be attached to the CM to execute **cmtar**.
- If your CM system has a VMEIO host computer, it is faster to use a tape drive attached to it than one attached to a front end. Issue **cmtar** on the VMEIO host to write to or read from a tape drive attached to it; see Section 7.2.2 on page 132.

If you want to keep a copy of files from a tape archive in your UNIX file system, you can issue the standard UNIX **tar** command to copy the files to UNIX, then use **copytodv** to copy them from the UNIX file system to the CM file system.

7.3.3 Copying Unarchived Data from Tape: The cmdd Command

Scientific data collected in the field is often placed on tape just as a series of numbers, not in files. You can copy such data into the CM file system using the **cmdd** command, which corresponds to the UNIX **dd** command. For example, the following command copies data from the tape drive **/dev/rmt0** to the file **dva:/datafile** in the CM file system:

```
% cmdd -todv if=/dev/rmt0 of=dva:/datafile
```

You can also use **cmdd** to copy a file from the CM file system to a UNIX file system; it performs the conversions you specify in the options. Like **copytodv** and **copyfromdv**, **cmdd** does not affect the parallel or serial format of a file, and it does not involve the CM, so you do not have to be attached to the CM to execute it.

It is also possible to write a front-end program to read data into the CM file system; see the *CM I/O System Programming Guide*.

7.3.4 Copying Files within the CM File System: The cmcp and dvcp Commands

There are two commands available for copying files within the CM file system: **cmcp** and **dvcp**. They differ as follows:

- **dvcp** uses the CM and the CMIO bus to perform the copy; **cmcp** uses the Ethernet. Therefore, **dvcp** is much faster than **cmcp**. (However, **cmcp**, when issued on a VMEIO host, will use the CMIO bus.
- Because **dvcp** uses the CM, it can be issued only from a UNIX front end when you are attached to a CM. **cmcp** can be issued from any UNIX computer on the CM system.
- **dvcp** can copy only one file at a time; **cmcp** can copy multiple files into a directory. For example,

```
% dvcp dva:/mydata dvb:/mydata
```

copies **dva:/mydata** to **dvb:/mydata**.

```
% cmcp dva:/mydata dva:/mydata1 dvb:/
```

copies the files **dva:/mydata** and **dva:/mydata1** to the root directory of **dvb**. The files keep their original names.

7.3.5 Transferring Files between a DataVault and a Remote System via UltraNet: The cmftp Command

Use the **cmftp** command from a CM-HIPPI system to transfer files between one or more DataVaults and a remote system via an UltraNet network. You must either be logged into the CM-HIPPI or use an **rsh** command from another system that can reach it by network.

7.4 Other CMFS User Commands

Other CMFS user commands have corresponding UNIX versions and perform the same functions as their UNIX counterparts. See the reference descriptions for complete discussions of these commands. Note the following:

- The **cmls** command can be executed in a Lisp environment. There are, however, certain restrictions; see its reference description in the *CM I/O System Programming Guide.*
- The **cmmv** command cannot be used to move a file between CM file systems (that is, from one directory tree to another). To do this, use **cmcp** (or **dvcp**) to copy the file, then use **cmrm** to remove the original file.
- The **cmln** command creates only hard links; unlike **ln**, it does not create symbolic links.

7.5 Environment Variables

CMFS provides four new environment variables: **CMFS_DEBUG**, **DVWD**, and **DVHOSTNAME**, and **CMFS_VERIFY_AFTER_WRITE**.

If you are using the C shell, add the following entry to your **.cshrc** file to activate the printing of CMFS debugging messages on your screen:

```
setenv CMFS_DEBUG 1
```

When debugging is activated and a program is executing, useful information is printed. For example, turning on debugging lets you know if a call is receiving an invalid argument or producing an unexpected return value.

The environment variable **DVWD** stores your working directory in the CM file system. For example, if you are using the C shell,

```
setenv DVWD dva:/myproject/mydata
```

makes **dva:/myproject/mydata** the working directory.

If the setting of the **DVWD** environment variable includes a hostname, that is the default hostname. If it doesn't, you can set a default hostname with the **DVHOSTNAME** environment variable. For example,

```
setenv DVHOSTNAME dvb
```

sets the default hostname to **dvb**. You can omit **dvb:** from the pathname of files in this file system.

The setting of the **CMFS_VERIFY_AFTER_WRITE** environment variable determines whether data that was written is to be compared with the original data. If the setting is **ON**, this comparison takes place. If the data is not identical, up to 10 rewrites are attempted. If the setting is **OFF**, no verification is attempted; this is the default, since setting this environment variable to on makes writes take twice as long.

Part IV
In the Lisp Environment

Chapter 8

In the Lisp Environment

This chapter describes how to develop and execute data parallel programs written in *Lisp and Lisp/Paris on either UNIX or Symbolics Lisp machine front ends. *Lisp is a parallel extension of Common Lisp; we assume that readers are familiar with Common Lisp. In this chapter, *Lucid* refers both to Lucid Common Lisp (for VAXes) and Sun Common Lisp (for Sun Workstations).

For complete information on *Lisp, see the volume *Programming in *Lisp* and the **Lisp Dictionary* in the CM documentation set; see also *Getting Started in *Lisp*. For complete information on Lisp/Paris, see the volume *Parallel Instruction Set.*

8.1 The *Lisp Language

*Lisp is a parallel extension of the Common Lisp language, and has the same syntax and style as Common Lisp.

*Lisp adds one major data type to Common Lisp: the parallel variable, or *pvar.* This is an abstract data object that represents the concept of a value stored in the memory of each processor on the CM. *Lisp also adds a large number of functions and macros that operate exclusively on pvars. Among these operators are parallel equivalents for many of the operators available in Common Lisp, as well as special-purpose operators that perform such CM-specific tasks as processor selection, interprocessor communication, and scanning.

*Lisp is available in two versions: as an interpreter/compiler combination for the CM hardware, and as a stand-alone simulator.

The *Lisp interpreter and compiler are extensions of the existing interpreter and compiler in Common Lisp, and *Lisp programs are written and compiled no differently from Common Lisp programs.

The *Lisp simulator runs entirely on the front-end computer and simulates the operations of an attached CM. Code developed using the *Lisp simulator can be ported directly to the *Lisp interpreter/compiler on the CM hardware with few modifications. However, code compiled using the simulator must be recompiled to run on the hardware.

8.2 Lisp/Paris

Lisp/Paris is implemented as a language interface between Lisp and Paris. All the operations of Paris are available directly as function calls from Lisp. See Section 8.18 on page 174 for more information about programming in Lisp/Paris.

It is not necessary to use the *Lisp language to program in Lisp/Paris. However, *Lisp provides a useful level of abstraction and handles most of the details of Paris programming in a clean, readable manner. It is possible to write programs consisting mostly of *Lisp code that call Paris directly only for important, time-critical operations.

It is also possible to write code in *Lisp, compile it, and then examine the resulting Paris code to see how a given program can be written using Lisp/Paris alone. Section 8.10 on page 152 describes the process of compiling *Lisp code in more detail.

8.3 Loading *Lisp and Lisp/Paris

Before using either *Lisp or Lisp/Paris, it is necessary to load in the appropriate software. *Lisp users will want to load in either the *Lisp interpreter/compiler or the *Lisp simulator.

Lisp/Paris users will want to load in the *Lisp interpreter/compiler software, because this includes the Lisp/Paris language interface.

8.3.1 From the UNIX Prompt

NOTE: At some sites, *Lisp bands configured differently from the ones mentioned here may be in use, each with its own initialization command. If this is the case at your site, ask your system administrator what commands you can use instead of those described here.

The following examples also assume that your UNIX PATH variable includes the directory in which these bands have been stored. Check with your system administrator to make sure that your PATH variable includes the right directory.

On UNIX front ends, you can load the *Lisp interpreter/compiler by typing this command at the UNIX prompt:

```
% starlisp
```

This starts up Lucid Lisp and loads in a saved world that includes *Lisp. From within this environment, you can attach to the CM, cold or warm boot the CM, compile, execute, and debug code on the CM, and detach from the CM.

If you want to be able to run programs under timesharing on a CM, type:

```
% starlisp-ts
```

This allows you to run programs on either a sequencer running under timesharing, or a non-timeshared sequencer.

To load the *Lisp simulator software, type:

```
% starlisp-simulator
```

This starts up Lucid Lisp and loads in a saved world that includes the *Lisp simulator. From within this environment, you can compile, execute, and debug code just as you would from the interpreter/compiler, without having to attach to a real CM.

8.3.2 From Gmacs

You can also invoke *Lisp from within a Gmacs editor, using a set of Gmacs enhancements that Thinking Machines Corporation provides free of charge but does not support. These enhancements are known as the "TMC Gmacs Hacks." They are available from your applications engineer or from Thinking Machines Customer Support.

Once the Gmacs hacks have been loaded into your Gmacs session, type:

```
M-x run-lisp
```

(where **M-** is the Gmacs "Meta" key). You are then prompted:

```
Type name of Lisp to run:
```

At this point, type the name of the *Lisp software you wish to load. For example:

```
starlisp
```

will load in the *Lisp software and begin a Lisp session in a buffer named ***lisp***.

There are many advantages to running Lisp under Gmacs. For example, the full power of the Gmacs editor is available, as well as many features common to the Symbolics Genera environment, such as source finding, display of argument lists, macroexpansion, and incremental compilation and completion. Once the Gmacs hacks have been loaded, issue the command

```
M-x help-tmc-hacks
```

for a list and short description of the available features.

8.3.3 From a Lisp Machine

On Symbolics Lisp machine front ends, the *Lisp software is typically pre-loaded as part of the world load file used to boot the machine. To use *Lisp, the *Lisp simulator, or Lisp/Paris alone, it is necessary to boot the machine using the appropriate world. Ask your system administrator for assistance in finding the right world to use in booting your Lisp machine.

8.4 Using *Lisp — An Overview

Once you have *Lisp loaded on your system, the process of developing and executing *Lisp and Lisp/Paris code is the same for UNIX and Lisp machine front ends.

Here, in brief, are the steps involved in developing and executing *Lisp and Lisp/Paris programs:

(1) *Make *Lisp the current package* by typing the following at the Lisp prompt:

```
> (*lisp t)
```

or:

```
> (in-package '*lisp)
```

These forms change the current package to **`*lisp`**, making the functions and macros of the *Lisp language available.

(2) *Attach to a CM* by using the Lisp/Paris function **`cm:attach`**, as follows:

```
> (cm:attach)
```

You can also use the **`cm:finger`** function to find out if there is a sequencer available to which you can attach.

(3) *Initialize the CM hardware and the *Lisp software* as follows:

```
> (*cold-boot)
```

At this point, you can perform any Common Lisp, *Lisp, or Lisp/Paris operation.

(4) Load, edit, compile, execute, and debug your data parallel programs.

(5) *Detach from the CM* when you are finished using it. Type:

```
> (cm:detach)
```

(6) *Exit from the *Lisp environment* (on UNIX front ends only) by typing:

```
> (lcl:quit)
```

(If you are running Lucid Common Lisp 2.1 or 2.5, type **`(sys:quit)`** instead.) You are returned to your UNIX prompt.

Because the *Lisp software is part of the booted world file on a Lisp machine, it is not necessary to "quit" from *Lisp as on a UNIX front end. You can exit from the *Lisp package, however, by typing either

```
> (*lisp nil)
```

or

```
> (in-package 'user)
```

Both of these forms change the current package to **user**.

The following sections go into more detail about each of these steps. They also discuss some of the things you can do in the Lisp environment.

8.5 Entering the *Lisp Package

All the functions and macros of *Lisp reside in the ***lisp** package. To select the ***lisp** package, first make sure that the *Lisp software is loaded, as described in Section 8.3 on page 142. Next, at the top level, type the following form:

```
> (*lisp t)
```

This form displays a message that tells you that the ***lisp** package has been made current.

You can also type:

```
> (in-package '*lisp)
```

Once in the *Lisp package, you can invoke any Common Lisp or *Lisp function or macro and any Lisp/Paris instruction; Lisp/Paris instructions called from within the *Lisp package begin with **cm:**. The time during which front-end code is executed with the ***lisp** package selected is called a **Lisp session,* and code executed with the ***lisp** package selected is said to be running in the **Lisp environment.*

8.6 Attaching to a CM

To execute a *Lisp program on a CM, you must first *attach* to the CM. As described in Chapter 1, a front end connects to a CM parallel processing unit via a FEBI (front-end bus interface). A FEBI can be logically attached to one or more sequencers on the CM; a sequencer controls groups of processors within the CM.

To attach to a CM from within the *Lisp package, use the Lisp/Paris command **`cm:attach`**. By default, you are attached to the smallest number of physical processors associated with one sequencer, and to the highest-numbered FEBI and sequencer available. You can specify more processors with an argument to **`cm:attach`**. For example, to attach to 16,384 processors, type:

```
> (cm:attach 16384)
```

or

```
> (cm:attach :16kp)
```

Valid values for the latter argument include **`:4kp`**, **`:8kp`**, **`:16kp`**, **`:32kp`**, and **`:64kp`**.

You can also specify a specific sequencer, or group of sequencers. For example,

```
> (cm:attach :ucc2)
```

specifies that you want to attach to sequencer 2. Valid values include **`:ucc0`**, **`:ucc1`**, **`:ucc2`**, :ucc3, **`:ucc0-1`**, **`:ucc2-3`**, and **`:ucc0-3`**. You might specify a particular sequencer, for example, if it is connected to a framebuffer.

If successful, **`cm:attach`** returns the number of physical processors that were attached by the call, and either `:TIMESHARING` or `:SINGLE-USER`, depending on whether the sequencer(s) are running under timesharing. (You can run *Lisp programs on a timeshared sequencer only if you issued the **`starlisp-ts`** command to load *Lisp.) It signals an error if the requested number of processors was not available. The error message includes two especially helpful options: attaching to the number of available processors (instead of requesting a larger number), and waiting for the requested number of processors to become available.

If you have more than one CM connected to a Sun or VAX front end, you will want to select the interface that is connected to the CM you wish to use. You can directly specify which interface you want to use by the keyword argument **`:interface`**. For example,

```
> (cm:attach :64kp :interface 3)
```

specifies that you want to attach to 64K processors on the CM connected to interface 3.

Use the keyword argument **:wait-p** to specify that you want to wait for the processors to become available. For example,

```
> (cm:attach :16kp :wait-p t)
```

asks for 16,384 processors and specifies that the front end is to wait until they become available. To quit waiting, type Ctrl-C. (If you are running under Gmacs, type Ctrl-C Ctrl-C. From a Lisp machine front end, type Ctrl-ABORT.)

For a complete discussion of the **cm:attach** command and its options, see the dictionary portion of the *Paris Reference Manual.*

8.7 Finding Out about CM Use

Use the Lisp/Paris command **cm:finger** to find out the current status of CMs in your system. This command prints out a table that shows which front ends are connected to sections of a CM system, who is using that section, what command or program is being executed, and whether the CM has any free sequencers.

This last piece of information is typically the most important. Many people routinely issue **cm:finger** before **cm:attach** to see if any CM resources are available. Alternatively, you can try **cm:attach** first; if no resources are available, you can issue **cm:finger** to find out what's happening.

At the top level within a *Lisp environment, type:

```
> (cm:finger)
```

A table is displayed; an example is shown below.

```
CM       Seqs Size   Front end I/F   User    Idle    Command
------------------------------------------------------------

FOO        1   8K    wotan      0    karen  0h 06m "cmattach"
FOO       ---  ---   epicurus   0    nobody

        1024K memory, 32-bit floating point
        framebuffers on sequencers 0 1 (seq 0 is free)
        CMIOCs on sequencers 0 1 (seq 0 is free)
        1 free seq on FOO -- 0 -- totalling 8K procs
```

In this case, the CM called Foo has two front-end interfaces: interface (I/F) 0 on Wotan and interface 0 on Epicurus. The user named Karen is attached to sequencer 1 of Foo via Wotan's FEBI; this sequencer has 8K processors. (Note that the number of the front-end interface does not have to correspond to the number of the sequencer to which it attaches.) Karen is running a **`cmattach`** subshell; she has been idle for six minutes. No one is using the FEBI on Epicurus.

The information below the list of users provides more data about the CM system:

- The memory size of the processors in this CM is 1 megabyte; it has 32-bit floating-point chips.
- Foo has a framebuffer and a CMIOC on both sequencer 0 and sequencer 1.
- Sequencer 0 of Foo is free for use.

For more discussion of this kind of output, see Section 3.1 on page 48.

`cm:finger` takes an optional argument, which must be the name of a CM, the name of a front end, or a list of CM or front end names. This is useful at sites with more than one CM or with a front end that has more than one FEBI.

If the user is shown as "{CM*}", the attached sequencer (or sequencers) is operating under timesharing.

8.7.1 On Symbolics Lisp Machines

On Lisp machines, the CM usage information is displayed differently. For example:

```
Name of CM:  Spicerack  Physical size = 32k CM2.
betty   CURRY     Not attached
boop    FENNEL:0  Sequencer Ports (1 )<-- Attached to 8192
                                          physical processors
nobody  GARLIC:0  Not attached
nobody  GARLIC:1  Not attached
nobody  MORREL:2  Not attached
NIL
```

The CM named Spicerack has 32,768 (32K) physical processors and four front ends: Curry, Fennel, Garlic, and Morrel. Betty is logged on to Curry. Boop is logged on to Fennel. Nobody is logged on to either Garlic or Morrel. Only one front end is currently attached: Fennel has a FEBI that resides in its expansion board slot 0 and it is attached to 8,192 processors through sequencer 1. This leaves three free sequencers. Given this situation, it is possible to attach to one or two of the free sequencers. (You can only attach to 1, 2, or 4 sequencers at a time—never 3.)

8.8 Initializing and Resetting the CM

8.8.1 *cold-boot

Immediately after attaching to a CM, use the *Lisp macro **`*cold-boot`** to initialize *Lisp and reset the CM hardware. You can also use arguments to **`*cold-boot`** to specify the default geometry of the virtual processors you are going to use and the safety level of the interpreter. If you are not already attached, **`*cold-boot`** will automatically issue a **`(cm:attach)`** command for you.

Use the **`:initial-dimensions`** keyword argument to specify the default geometry of the virtual processors. Each dimension must be a power of two, and the total number of virtual processors must be the same as, or a multiple of, the number of physical processors. For example, if you have attached to 8192 processors:

```
> (cm:attach :8kp)
8192
```

you could call **`*cold-boot`** as follows:

```
> (*cold-boot :initial-dimensions '(16 32 32))
```

The system responds:

```
8192
(16 32 32)
```

You have specified a default 3-dimensional geometry of 16 by 32 by 32, and allocated the 8192 physical processors as 8192 virtual processors (a virtual processor ratio of 1). For more information on the virtual processor mechanism, see Chapter 5 of the *Supplement to the *Lisp Reference Manual*, and the *Paris Reference Manual*. The dimensions that you specify are bound to a global variable called **`*default-vp-set*`**.

If you don't specify a value for **`:initial-dimensions`**, the system returns a default VP set that has a 2-dimensional geometry with a VP ratio of 1.

Use the **`:safety`** keyword argument to set the safety level of the interpreter. Specify a safety level of 3 to enable all interpreter error checking. For example:

```
> (*cold-boot :initial-dimensions '(128 256) :safety 3)
```

We highly recommend this setting when you are debugging *Lisp code. Specify a safety level of 0 to turn off most of the run-time error checking that the *Lisp interpreter would otherwise do.

NOTE: The **`:safety`** argument does not affect compiler safety. See Section 8.10 on page 152 for a discussion of the *Lisp compiler.

Another useful argument to **`*cold-boot`** is **`:undefine-all`**, which causes *Lisp to deallocate all VP sets and pvars.

For complete information on **`*cold-boot`**, consult the **Lisp Dictionary*.

8.8.2 *warm-boot

Use **`*warm-boot`** whenever a *Lisp program has an error and you abort back to top level. **`*warm-boot`** clears all CM error conditions and clears CM stack memory, but does not alter the contents of CM heap memory. We also recommend calling **`*warm-boot`** at the beginning of stand-alone *Lisp programs, in case previously run and aborted code has left the CM hardware in an inconsistent state.

8.9 Developing and Executing *Lisp and Lisp/Paris Code

You can develop and execute *Lisp and Lisp/Paris code in a file on the front end, in the same way you would normally develop and execute Lisp code. The simplest method is to type code directly at the Lisp prompt. For example, the following code defines a function called **hypotenuse!!**, which calculates the hypotenuse of a right triangle in each processor on the CM:

```
> (defun hypotenuse!! (x y)
    (declare (type single-float-pvar x y))
    (sqrt!! (+!! (*!! x x) (*!! y y)))
    )
```

To call the **hypotenuse!!** function, you must supply pvars as arguments. For example:

```
> (hypotenuse!! pvar1 pvar2)
```

You can also use the editor on your front end to edit and save files of *Lisp code. For example, if a file called **/user/lisp/my-file** contains *Lisp code, you can load it for interpreted execution by typing

```
> (load "/user/lisp/my-file")
```

If the main function in **my-file** is called **start-here**, for example, you can then execute the program by typing

```
> (start-here)
```

8.10 Using the *Lisp Compiler

The *Lisp compiler is an extension to the Common Lisp compiler as implemented on your front end. Invoking the Common Lisp compiler on any *Lisp file or function definition automatically invokes the *Lisp compiler. The *Lisp compiler translates *Lisp code into Common Lisp code with calls to Paris. Then the Common Lisp compiler translates the code into native machine instructions.

Compiled *Lisp runs more efficiently than interpreted *Lisp, but in order for *Lisp code to compile completely, it must be properly declared. For a discussion and examples of type declarations in *Lisp, see Chapter 4, "*Lisp Type Declaration," of the **Lisp Dictionary* in the Connection Machine documentation set.

To compile a *Lisp function, use the Common Lisp **compile** function. For example, the following compiles the **hypotenuse!!** function shown in Section 8.9:

```
> (compile 'hypotenuse)
HYPOTENUSE
```

To compile all definitions within a file containing *Lisp code, use the Common Lisp **compile-file** function. For example, to compile the file **/user/lisp/my-file** from Section 8.9, type

```
> (compile-file "/user/lisp/my-file")
```

Some front-end editors include special keystrokes that incrementally compile code. For example, in the Lisp machine Zmacs editor, the keystroke Ctrl-Shift-C compiles the function definition surrounding the cursor.

For further information on using the *Lisp compiler, consult the **Lisp Compiler Guide*, in the Connection Machine documentation set.

8.11 Debugging

*Lisp and Lisp/Paris code can be debugged using your Lisp system's debugger, just as with Common Lisp code. There are also, however, a number of *Lisp functions that you may find useful in debugging the programs you create.

The basic debugging tool for *Lisp is **ppp** ("pretty-print pvar"). It allows you to print out the values of a pvar in all processors or in any subset of processors, and to control the format with which these values are displayed.

For example, the call to **ppp** in the following code prints out the results of **hypotenuse!!** for all processors whose send addresses are less than 5.

```
> (ppp
   (hypotenuse!!(float!! (self-address!!))
                 (float!! (self-address!!)))
   :end 5)
0.0 1.4142134 2.8284268 4.2426405 5.6568537
```

The function **self-address!!** returns a pvar whose value in each processor is the send address of that processor. The values calculated by the above call to **hypotenuse!!** are therefore the hypotenuse lengths for triangles with sides x=0, y=0; x=1, y=1; and so on.

(Note that because the arguments to the **hypotenuse!!** function have been declared to contain floating-point numbers, it is necessary to use the *Lisp operation **float!!** to convert the integer pvars returned by the two calls to **self-address!!** to the floating-point pvars expected by **hypotenuse!!**.)

The **ppp** function has a large number of arguments that control which values are displayed and the format in which they are printed. There are also a number of specialized functions similar to **ppp** that you can use to display the values of pvars. For more information about **ppp**, its arguments, and other related functions, refer to the **Lisp Dictionary* in the Connection Machine documentation set.

Another useful function is ***room**. This function displays the amount of CM memory remaining, as well as the amount currently being used in the pvars you have created. A sample call to ***room** looks like:

```
> (*room :how :totals)

*Lisp system memory utilization

Stack memory usage   : 0
Heap memory usage    : 0
*Defvar memory usage: 0
Overhead             : 328
Total                : 328
```

You can call the Common Lisp function **describe** with a pvar argument, but the information it displays is not very useful. There is a *Lisp equivalent, **describe-pvar**, which accepts only pvar arguments and provides a more detailed and informative display.

For example:

```
> (describe-pvar (!! 2))

Pvar name: NIL
  Location: 4
  Field Id: 65536
  Length: 2
  Type: :FIELD
  Vp Set Name: *DEFAULT-VP-SET*
  Vp Dimensions: (32 16)
  Constant Value: 2
```

For more information about **`*room`**, **`display-pvar`**, and other functions available in *Lisp, refer to the **Lisp Dictionary* in the Connection Machine documentation set.

8.12 Timing *Lisp Code

The CM system provides a timing utility that lets you determine how much time any part of a program takes to execute on the CM. The timer consists of a set of Paris instructions that you insert at the appropriate places in your program. These functions allow you to:

- Calculate total elapsed front-end process run time and the total amount of time the CM is active.
- Run multiple timers—up to 64—at the same time.
- Nest timers. This allows you, for example, to start one timer that will time the entire program, while using other timers to determine how different parts of the program contribute to the overall time.

8.12.1 Timing Your Code with CM:TIME

The simplest way to time a piece of *Lisp or Lisp/Paris code is to wrap the macro **`CM:TIME`** around it. For example,

```
(defun test-fun (loops)
  (declare (type fixnum loops))
   (*let ((temp-pvar loops)) ;; define a pvar with loops in
                             ;; every proc.
    (declare (type (field-pvar 32) temp-pvar))
    (dotimes (i loops) ;; loop until pvar decrements to zero
      (*decf temp-pvar))))

(compile 'test-fun)

(cm:time (test-fun 100000))
Evaluation of (TEST-FUN 100000) took 9.572998 seconds of elapsed
time, during which the CM was active for 9.572998 seconds or
100.00% of the total elapsed time.
```

The arguments to **CM:TIME** are:

```
(CM:TIME FORM &KEY RETURN-STATISTICS-ONLY-P)
```

where:

FORM is a single Lisp or *Lisp form, which is timed using the CM timer mechanism.

RETURN-STATISTICS-ONLY-P
is an optional keyword argument that controls whether **CM:-TIME** prints its statistics or simply returns them as multiple values. Refer to the *Paris Reference Manual* for an explanation of what the numbers mean. By default, **CM:TIME** displays its results on the standard output.

Note that the *FORM* argument must be a single Lisp expression. If you wish to time more than one form, enclose them in a **PROGN**, as in:

```
(CM:TIME (PROGN (+!! 2 3) (-!! 3 2))
         :return-statistics-only-p t)
0.001455431565570317 4
0.001455431565570317 4
100
```

Calls to **CM:TIME** can be nested, as in the following example:

```
(cm:time (progn (cm:time (test-fun 5000)) (cm:time (test-fun
5000))))

Evaluation of (TEST-FUN 5000) took 0.473473 seconds of elapsed
time, during which the CM was active for 0.473473 seconds or
100.00% of the total elapsed time.

Evaluation of (TEST-FUN 5000) took 0.480044 seconds of elapsed
time, during which the CM was active for 0.480044 seconds or
100.00% of the total elapsed time.

Evaluation of (PROGN (CM:TIME (TEST-FUN 5000)) (CM:TIME (TEST-
FUN 5000))) took 1.036253 seconds of elapsed time, during which
the CM was active for 1.036253 seconds or 100.00% of the total
elapsed time.
```

8.12.2 Using Timers in *Lisp Code

It is also possible to gain access to the timing mechanism of the CM directly. You can have up to 64 timers running in a program. (The actual limit on the number of active timers is given by the value of **cm:*number-of-timers***.)

Each timer is referenced by a unique unsigned integer (from 0 to 63) that is used as an argument to the Paris timing instructions. Instructions with a given timer number as an argument affect only the timer with that number.

Each timer maintains its own copy of the following values:

- **Total Elapsed Time**. This is the total time the timer has been running since it was last cleared.
- **Total CM Idle Time**. This is the total time the CM has been idle while the timer was active.
- **Number of Starts.** This is the number of times the timer has been started since it was last cleared.

Starting, Stopping, and Printing the Values of a Timer

To start timer 0, put a call to the following function in your program:

```
(cm:timer-start 0)
```

The first time you start a timer, the timing mechanism is initialized, and a message like this is displayed:

```
Calibrating CM idle timer... Calculated CM clock
speed = 6.99714 MHz
```

You can subsequently stop timer 0 by calling the following function later in your program:

```
(cm:timer-stop 0)
```

This function increments the total elapsed time and total CM busy time for this timer. You can subsequently call **(cm:timer-start 0)** again to restart timer 0; the timing starts at the values currently held in the timer. This is useful for measuring how much time is spent in a frequently called subroutine. The timer keeps track of the number of times it has been restarted.

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

To get the results from timer 0, call the following function after you have called **cm:timer-stop**:

```
(cm:timer-print 0)
```

cm:timer-print prints information like that shown below to your standard output:

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

Clearing Timers and Initializing the Timer System

To clear the values maintained by timer 0, call the following function:

```
(cm:timer-clear 0)
```

The **`cm:timer-clear`** function zeroes the total elapsed time, the total CM idle time, and the number of starts for the specified timer.

Other Timer Operations

The following functions return specific information from the timer for use in a program. (Note that the *timer* argument for any of these functions must be the number of a timer for which you have previously called **`cm:timer-start`**.)

- (**`cm:timer-read-starts`** *timer*) returns an integer that represents the number of times the specified timer has been started.
- (**`cm:timer-read-elapsed`** *timer*) returns a double-precision value that represents the total elapsed time (in seconds) for the specified timer.
- (**`cm:timer-read-cm-busy`** *timer*) returns a double-precision value that represents the total time (in seconds) the CM was busy for the specified timer.
- (**`cm:timer-read-cm-idle`** *timer*) returns a double-precision value that represents the total CM idle time (in seconds) for the specified timer. (CM idle time is the total elapsed time minus the CM busy time).
- (**`cm:timer-read-run-state`** *timer*) returns `t` if and only if the specified timer is running.
- (**`cm:timer-set-starts`** *timer value*) takes a timer number and an integer value as arguments. It sets the number of starts for the specified timer to the specified value.

8.12.3 Interpreting the Results

In interpreting the results of a timer, it is important to understand something of how the timing utility works.

The elapsed time reported by a timer includes time when the process running the program is swapped out on the front end. The more processes that are running on the front end, the more distorted this figure will be. Therefore, we recommend the following:

- Use a front end that is as unloaded as possible.
- Run the program several times; the minimum elapsed time reported will be the most accurate.

CM idle time includes only those cycles during which the CM is waiting for an instruction from the front end. Consequently, CM active time includes not only those cycles during which the CM is performing computations, but also those during which the CM is waiting for arguments to an instruction it has received. Therefore:

- Expect slightly different CM active times on different front-end models for code segments that do not keep the CM 100 percent active. The time the CM spends waiting for data to appear is counted as active, but front-end models differ in the speed with which they can move data over the FEBI to the sequencer.
- Avoid stopping a process that is being timed.

In addition, note that the timer turns Paris safety checking off; see Section 8.19 on page 175.

8.12.4 An Example

The following *Lisp program uses several features of the timing utility:

```
((defun timing-example (&optional (loops 1000))
       (*let ((a 0))
          (let ((b 0))
             (format t "~%Integer pvar addition~%")
             (cm:timer-start 0)
             (cm:timer-start 1)
             (dotimes (i loops)
                (*set a (+!! a 1)))
             (cm:timer-stop 1)
             (format t "Timer 1 results:")
             (cm:timer-print 1)
             (cm:timer-clear 1)
```

```
                  (format t "~2%Scalar integer addition~%")
                  (cm:timer-start 1)
                  (dotimes (i loops)
                  (setq b (+ b 1)))
                  (cm:timer-stop 1)
                  (format t "Timer 1 results:")
                  (cm:timer-print 1)
                  (format t "~2%Total process time:~%")
                  (format t "(Timer 0 results)")
                  (cm:timer-stop 0)
```

A sample call to this function is shown below.

Note that the program uses one timer (0) to time the entire program, and another timer (1) to time the two **dotimes** loops within the program. The first **dotimes** loop uses the CM; the second executes on the front end alone.

```
> (timing-example)

Integer pvar addition
Calibrating CM idle timer... Calculated CM clock speed =
6.99866 MHz
Timer 1 results:
Starts: 1
CM Elapsed time: 2.32033 seconds.
CM busy Time: 0.177360 seconds.

Scalar integer addition
Timer 1 results:
Starts: 1
CM Elapsed time: 0.409085 seconds.
CM busy Time: 6.904490E-6 seconds.

Total process time:
(Timer 0 results)
Starts: 1
CM Elapsed time: 3.04659 seconds.
CM busy Time: 0.177455 seconds.
```

8.13 Detaching from the CM

When you are finished using the CM, you must explicitly release the sequencers to which you are attached. To do this, use the Lisp/Paris command **cm:detach**:

```
> (cm:detach)
```

All your attached sequencers are detached.

You can call **cm:detach** to release one set of sequencers, and then immediately call **cm:attach** to attach to the same set of sequencers or to a different set. You can call **cm:detach** and **cm:attach** in this way as many times as you like during your *Lisp session. However, you must *always* call **cm:detach** at the end of each session if you still have sequencers attached, so that those sequencers can be made available to other users.

8.14 Exiting *Lisp

On UNIX front ends, to exit *Lisp you must quit your Lucid Lisp session. To exit from Lucid, type:

```
> (lcl:quit)
```

(On VAX front ends, if you are using Lucid Common Lisp 2.1 or 2.5, you must type **(sys:quit)** instead.)

This command ends the Lisp session.

Users on Lisp machine front ends do not need to explicitly exit from *Lisp. Detaching from the CM is sufficient. You may, however, wish to use either the ***lisp** command or **in-package** to make some other package current instead of the ***lisp** package. For example, either

```
> (*lisp nil)
```

or

```
> (in-package 'user)
```

will make the **user** package the current package.

8.15 Using the CM Batch System from *Lisp

Using the CM batch system (NQS) from *Lisp is similar to using it from UNIX; see Chapter 2 for a complete discussion of how to submit a batch request from UNIX. Note the following similarities:

- From *Lisp, as from UNIX, you use the **qsub** command to submit a batch request to a queue; the queue can be either a batch queue or a pipe queue.

- You can specify the name of a script file that contains the program to run as an argument to **qsub**, or you can submit the program to **qsub** from the standard input.

- You can include options to **qsub** that, for example, specify the queue to which the request is to be submitted, and whether you want mail sent to you when the request starts running. For example,

```
% qsub -q cm1 -o starlisp.out starlisp.script
```

  submits the script file **starlisp.script** to queue **cm1**; the output is to go to the file **starlisp.out**.

- You can use other NQS commands such as **qstat** (to check on the status of your request) and **qdel** (to delete a request).

Note the following important differences, however:

- The request you submit (either from the standard input or a shell script) must be the name of a *Lisp executable band (generally **/usr/local/starlisp**).

- You must ensure that the *Lisp system attaches to the same sequencer(s) and interface that the batch queue is using.

These differences are discussed in more detail below. Following the discussions is a sample *Lisp program you can use as a template for your own batch requests.

8.15.1 Submitting the Name of a *Lisp Executable Band

As mentioned above, the request you submit must be the name of a *Lisp executable band. As options, use the following:

- **-l** Use this option to specify the *Lisp file you want the band to load.
- **-n** This option tells Lucid Common Lisp not to load your **lisp-init.lisp** file after it has loaded the specified file.
- **-q** This option tells Lucid Common Lisp to terminate execution instead of entering an interactive session, once it has done all its initializations.

For example:

```
/usr/local/starlisp -l run-main.lisp -n -q
```

executes a *Lisp band and loads the file **run-main.lisp**.

For complete information on options to a Lisp executable band, see the Lucid 3.0 *Advanced User's Guide.*

8.15.2 Attaching to the Correct Sequencer and Interface

Your *Lisp program must attach to the same CM, sequencer, and front-end bus interface as the batch queue in which it runs. Rather than hardcoding this information in your program, we recommend obtaining the information from the environment. The batch queue sets the UNIX environment variables **CMSEQUENCERS**, **CMINTERFACE**, and **CMNAME** when it runs a request. The sample program shown below demonstrates how to use these values to attach correctly.

8.15.3 Sample Program

The file **run-main.lisp** is shown below. As discussed above, it uses the environment variables **CMSEQUENCERS**, **CMINTERFACE**, and **CMNAME** in attaching to the correct sequencer and interface.

Once the CM has been successfully attached and cold-booted, user code (in this case, a sample program included with the 6.0 release) is loaded, and then a user-defined program is executed. Various informative messages are printed out as execution proceeds.

This is just an example of a file used to control the Lisp batch system. You can adapt it to suit your needs.

```
(in-package '*lisp)
;;; The following sequence submits a starlisp batch job
;;; to the batch queue called FOO_1.  Output is directed
;;; to the file ~/starlisp-output.text.  Mail is sent to
;;; user 'massar' when the job begins and when the job
;;; ends.

;;; The starlisp executable /usr/local/starlisp
;;; is invoked, and told to load the file ~/run-main.lisp
;;; when it starts up via the -l flag.  The -n flag
;;; says not to load user massar's lisp-init.lisp file,
;;; and the -q file says to terminate the Lisp process
;;; once the ~/run-main.lisp file has been loaded.

;;; The ^D (CONTROL-D) tells qsub to submit the job
;;; and exit.
;;; massar% qsub -q foo_1 -me -mb -o ~/starlisp-output.text

;;; /usr/local/starlisp -l ~/run-main.lisp -n -q
;;; ^D
;;;
;;; Sample output from a run is shown below the code.

;;; BEGIN LISP EXECUTION.
;;;  (LOAD YOUR OWN LISP-INIT.LISP FILE HERE IF YOU WISH)

;;;  (load "/u/massar/lisp-init.lisp")


(format t "~%;;; Lisp batch system beginning execution")
(format t "~%;;; Beginning attach/cold-boot sequence.")
(finish-output)

(defun attach-parameters-from-batch-environment ()
  (flet
      ((oops (&rest strings)
        (dolist (string strings) (format t "~%*** ~A" string))
        (format t "~%*** Fatal error.  Lisp batch system terminating")
        (sys::quit)
  ))
    (let ((sequencer-environment-string
            (sys::environment-variable "CMSEQUENCERS"))
      (interface-environment-string
            (sys::environment-variable "CMINTERFACE"))
  )
      (when (null sequencer-environment-string)
         (oops "Batch system is apparently not attached to a CM!!!"
      "CMSEQUENCERS environment variable is not bound."
      ))
      (when (null interface-environment-string)
        (oops "Batch system didn't set CMINTERFACE env. variable!!!")
)
 (let ((sequencer
     (cond
   ((string-equal "0" sequencer-environment-string) :ucc0)
   ((string-equal "1" sequencer-environment-string) :ucc1)
   ((string-equal "2" sequencer-environment-string) :ucc2)
   ((string-equal "3" sequencer-environment-string) :ucc3)
   ((string-equal "0-1" sequencer-environment-string) :ucc0-1)
       ((string-equal "2-3" sequencer-environment-string) :ucc2-3)
((string-equal "0-3" sequencer-environment-string) :ucc0-3)
   (t (oops "CMSEQUENCERS value, ~S, is not valid!!!"))
   ))
    (interface
     (cond ((string-equal "0" interface-environment-string) 0)
   ((string-equal "1" interface-environment-string) 1)
   ((string-equal "2" interface-environment-string) 2)
   ((string-equal "3" interface-environment-string) 3)
   (t (oops "CMINTERFACE value, ~S, is not valid!!!"))
   )))
(values sequencer interface cmname)
))))

;;; ATTACH AND COLD BOOT

(multiple-value-bind (sequencer interface cmname)
    (attach-parameters-from-batch-environment)
  (format t "~%;;; Attaching to sequencer(s) ~S, interface ~D"
sequencer interface)
  (finish-output)
  (cm:attach sequencer :interface interface)
  (format t "~%;;; Cold-booting Connection Machine ~A" cmname)
  (finish-output)
```

```
    (*cold-boot)
   )
;;; EXECUTE USER CODE HERE

;;; (LOAD ALL YOUR FILES AND CALL YOUR MAIN ROUTINE)

(progn
   (format t "~%;;; Beginning execution of user code")
   (finish-output)
   (dfs:load-n
    "/cm/starlisp/interpreter/f6100/text-processing-example")
   (*lisp::do-text-processing "This is some text to process")
   )

;;; ALL DONE

(finish-output)
(format t "~%;;; Lisp batch system execution terminating.")
(finish-output)
(terpri)
```

Output from the batch request is shown below:

```
;;;
;;; Sun Common Lisp, Development Environment 3.0.5 (Rev 01),
;;; 30-Aug-90
;;; Sun-4 Version for SunOS 4.0
;;;
;;; Copyright (c) 1985, 1986, 1987, 1988 by Sun Microsystems, Inc.,
;;; All Rights Reserved
;;; Copyright (c) 1985, 1986, 1987, 1988 by Lucid, Inc., All Rights
;;; Reserved
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems. It may not be copied
;;; for any reason other than for archival and backup purposes.

;;; Loading source file "/cm/patch/initializations.lisp"
;;; *Lisp Patch Level 0
;;; Loading source file "/cm/patch/tmc-initializations.lisp"

;;; Connection Machine Software, Release 6.0
;;;
```

```
;;; Copyright (C) 1990 by Thinking Machines Corp.  All rights
;;; reserved.
;;;
;;; Loading source file "/u7/massar/run-main.lisp"
;;; Lisp batch system beginning execution.
;;; Beginning attach/cold-boot sequence.
;;; Attaching to sequencer(s) :UCC1, interface 1
;;; Loading source file "/cm/configuration/configuration.lisp"

;;; Cold-booting Connection Machine FOO
;;; Beginning execution of user code
;;; Loading binary file
;;;"/cm/starlisp/interpreter/f6000/text-processing-example.sbin3"

Processor 0. Length: 4.  Word: This
Processor 1. Length: 2.  Word: is
Processor 2. Length: 4.  Word: some
Processor 3. Length: 4.  Word: text
Processor 4. Length: 2.  Word: to
Processor 5. Length: 7.  Word: process
;;; Lisp batch system execution terminating.
logout
```

8.16 Running *Lisp Programs under Timesharing

As described in Section 8.3.1, you use the **starlisp-ts** command to load the version of *Lisp that runs under timesharing. Programs can run either under timesharing or in exclusive mode; use the **cm:finger** command to determine which sequencer(s) are running timesharing. Programs compiled under this version of *Lisp can run under the non-timesharing version of *Lisp, and vice versa. Some error messages are different under timesharing.

Generally, programs run somewhat more slowly under timesharing.

For the most part, code compiled under timesharing should be able to run without recompilation under exclusive mode, and vice versa. Exceptions are noted below.

8.16.1 Restrictions

This section describes existing problems, restrictions, and workarounds for *Lisp timesharing. In general, the timesharing system is an interface to documented Paris; the restrictions apply only to undocumented Paris and to CMIS.

The cm:attach Command

As mentioned in Section 8.6 on page 146, the **cm:attach** command now returns as a second value either **:TIMESHARING** or **:SINGLE-USER**, depending on whether the sequencer to which you have attached is running timesharing or not.

Under timesharing, the syntax of **cm:attach** has been augmented, as follows:

```
(CM:ATTACH &KEY INTERFACE SEQS PROCESSORS  CM)
```

where **INTERFACE**, **PROCESSORS**, and **SEQS** have the values described in Section 8.6. **CM**, if provided, must be a string.

The old syntax still works.

Paris Floating-Point Instructions

In Lisp, many Paris floating-point instructions allow the mantissa and exponent arguments to be optional; they default to 23 and 8. (This is undocumented.) *The timesharing system requires that the mantissa and exponent arguements be provided explicitly* in all Paris instructions, as the documentation states. If your code relies on the undocumented behavior, it must be changed to run under timesharing.

Undocumented CMI:: Functions

Many undocumented **CMI::** functions do exist and can be called under timesharing. If your code uses one that does not exist under timesharing, you should first see if it can be replaced by a call to a documented Paris function. If not, contact Thinking Machines Customer Support to see if it can be included in the next release of the software, or if a workaround or fix can be provided.

Undocumented CMI:: Variables

Due to the nature of the timesharing interface, it is not possible to provide access to most **CMI::** variables. You should determine whether the accessing of an internal **CMI::** variable can be replaced by a documented Paris function or variable.

If this is not possible, the function

```
(*LISP-I::PORTABLE-VARIABLE-ACCESSOR cmi::symbol)
```

returns the value of the variable, accessed from the C world. NOTE: The value returned for boolean variables is 0 or 1, instead of **NIL** or **T**.

This function exists in the regular *Lisp band, so it is a completely portable construct.

Undocumented CMI:: Macros

In general, **CMI::** macros do not work under timesharing, especially macros that are used to access very low-level primitives, such as writing to the FIFO. Also, the field decoding macros do not work, as discussed below.

Field Decoding Macros

Field decoding macros do not work under timesharing. However, in the **IMP::** package we have defined a set of portable field decoding macros with the same names. These include:

- **IMP::WITH-VP-FIELDS**
- **IMP::WITH-ANY-VP-FIELDS**
- **IMP::WITH-TRANSLATED-FIELDS**

If you replace uses of **CMI::** field decoding macros with these macros, your code should be portable.

Error Messages

The error messages generated by the *Lisp interpreter and *Lisp compiler at safet;y level 1 (which uses the **cmi::error-if-location** mechanism) look somewhat different under timesharing. Here is an example:

```
> (setq *interpreter-safety* 1)
1
> (/!! 3 0
#<FLOAT-Pvar 5-32 *DEFAULT-VP-SET* (32 16)>
> (*sum (!! 1))
>>ERROR:    Delayed error from ERROR-IF-LOCATION.
            *** An error occurred in your code between the
            *** last time a value was read out of the CM and now.
            One of the following occurred:
The result of a (two argument) float /!! overflowed, or
Divide by zero in float (two argument) /!!
            *** Once you abort, remember to cm:warm-boot or
*warm-boot***

CMI::PRETTY-PRINT-ERROR-IF-LOCATION-ERROR-MESSAGE:
            Required arg 0 (LOCATION): 3568
            Required arg 1 (TAG): 1553
:A 0: Abort to Lisp Top Level

->.
```

Messages that are generated by Paris also look different. Here is an example:

```
> (setq x (cm:allocate-stack-field 32))
65536
> (cm:move x x 33)
>>Error:

     An error occurred within C/Paris.

     Trying to access off of the end of field 65536. The
passed field has a length of 32, and the length passed to
this instruction is 33.
     *** Once you abort, remember to cm:warm-boot or
*warm-boot ***

CMI::PRETTY-PRINT-C-ERROR-MESSAGE:
     Required arg 0 (STRING): "Trying to access off of the
end of field 65536. The passed field has a length of 32,
and the length passed to this instruction is 33."
:A 0: Abort to Lisp Top Level

->
```

Being Detached

If you are detached or the timesharing daemon goes down, you should see this message:

```
Febi interrupt. This probably means you were just
detached.
```

8.16.2 Conditionalizing Code

You can, if necessary, conditionalize your code for compile time as follows:

```
#+C-PARIS
(code for *Lisp timesharing)

#-C-PARIS
(code for standard *Lisp)
```

You can also, if necessary, conditionalize your code at run time as follows:

```
(if cmi:: *timesharing*
   (code for *Lisp timesharing)
   (code for standard *Lisp)
    )
```

8.17 Using the *Lisp Simulator

The *Lisp simulator runs entirely on the front-end computer, and simulates the operations of a permanently attached CM. You can port code developed using the *Lisp simulator directly to the *Lisp interpreter/compiler with few modifications. However, code compiled using the *Lisp simulator must be recompiled to run on the CM hardware.

The performance of the simulator is much slower than that of an actual CM, but the simulator is a useful development tool when CM resources are scarce. One important advantage of the simulator is that it can be run on any available machine. It does not have to be run on the front end attached to your CM. The *Lisp simulator will run on any of a number of platforms. Consult the **Lisp Reference Manual* for more information.

Another advantage of the simulator is that it allows you to define VP sets of any size, so that you can test out your code on a small number of processors and print out the results for all of the processors at once, before recompiling it and running it on the thousands of processors available on a real CM.

To use the *Lisp simulator, load in the simulator software as described in Section 8.3 on page 142. The simulator emulates a *Lisp interpreter with a CM that is permanently attached, so it is unnecessary to call either **cm:attach** or **cm:detach** when using the simulator. (In fact, functions whose names begin with **cm:** don't exist in the *Lisp simulator. Calling one of these functions will signal an error.)

Operation of the simulator is identical to that of the *Lisp interpreter/compiler. Make ***lisp** the current package by typing

```
> (*lisp)
```

or

```
> (in-package '*lisp)
```

(Note that these operations will report that the package ***sim** has been made current. This is the actual package in which the *Lisp simulator software resides. However, no errors will result from referring to symbols in the ***lisp** package; in the *Lisp simulator the ***lisp** package is defined as a nickname for the package ***sim**.)

Initialize *Lisp by calling

```
> (*cold-boot)
```

The default VP set on the simulator consists of 32 processors in an 8 by 4 array, but you can use ***cold-boot** to define VP sets of any size you wish. For example,

```
> (*cold-boot :initial-dimensions '(16 16))
```

initializes the *Lisp simulator with a VP set of 256 processors in a 16 by 16 array.

At this point you can edit, execute, and debug *Lisp code as you normally would. When you are finished, there is no need to call **cm:detach**. Simply exit from *Lisp (or switch to another package) as described in Section 8.14 on page 162.

8.18 Lisp/Paris Programming

Lisp/Paris is implemented as a language interface between Lisp and Paris. All the operations of Paris are available directly as function calls from Lisp. For example, a call to the function **cm:latch-leds** would look like:

```
> (cm:latch-leds field-id)
```

where **field-id** is a variable bound to the field ID of a one-bit field on the CM. (For more information about fields and field IDs, refer to the "Concepts" section of the *Paris Reference Manual* in the Connection Machine documentation set.)

It is not necessary to use the *Lisp language to program in Lisp/Paris. However, *Lisp provides a useful level of abstraction and handles most of the details of Paris programming in a clean, readable manner.

It is possible to write programs consisting mostly of *Lisp code that call Paris directly only for important, time-critical operations. To call a Paris operation from *Lisp, it is necessary to convert the pvar data structures created by *Lisp code into the field IDs and field lengths required by Paris functions. It is also necessary to convert VP set objects created in *Lisp to the geometry IDs and VP-set IDs required by Paris functions.

*Lisp provides two utility functions to perform this conversion on pvars. The function **pvar-location** takes a single pvar argument, and returns the field ID of that pvar. The function **pvar-length** takes a single pvar argument, and returns the field length in bits of that pvar.

*Lisp also provides a utility function to perform this conversion on VP sets. The function ***lisp-i::vp-set-internal-id** takes a single VP set argument and returns its Paris VP-set ID. You can then call the Paris operation **cm:vp-set-geometry** to obtain the Paris geometry-id associated with that VP set.

As an example, the following *Lisp code uses ***defvar** to create a pvar with a negative value in every processor, calls the Paris operation **cm:s-negate-1-1l** to perform a signed negation on the pvar, and then calls the function **ppp** to show that the pvar's value in every processor has been negated.

```
(*defvar x (!! -6))
(ppp x :end 12)
-6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6

(cm:s-negate-1-1l (pvar-location x) (pvar-length x))

(ppp x :end 12)
6 6 6 6 6 6 6 6 6 6 6 6
```

Of course, this example can be performed much more easily in *Lisp by the expression `(*set x (-!! x))`, but for some applications calling Paris directly may be just what you need to increase the speed and efficiency of your programs.

For more information about Paris and the Lisp/Paris interface, consult the *Paris Reference Manual* in the Connection Machine documentation set.

8.19 Paris Run-Time Safety Checking

The CM system provides a safety utility that checks for Paris-level errors and inconsistencies in programs. Safety checking reduces execution speed, of course, but it can be useful in developing and debugging programs.

When turned on, the safety utility checks the following:

- Whether field IDs passed as arguments to Paris instructions refer to fields in the current VP set
- Whether field IDs passed as arguments to Paris instructions are valid field IDs (although not all invalid field IDs are caught)
- Whether the lengths passed to Paris instructions exceed the lengths of the respective field operands

(For information on field IDs and VP sets, see the *Paris Reference Manual.*) When the utility detects an error, it aborts the execution of the program and prints information about the error to your standard error device.

To use this utility, call the Paris function `cm:set-safety-mode` from within your program. To turn on safety, specify any non-zero integer as an argument to the instruction. For example,

```
(cm:set-safety-mode 1)
```

turns on complete Paris safety. To turn Paris safety off, specify zero as the argument:

```
(cm:set-safety-mode 0)
```

Note that you can also turn Paris safety on and off with the `:safety` keyword argument to `*cold-boot`; see Section 8.8 on page 150.

8.20 The *Lisp Library

An additional set of useful *Lisp functions and macros is available in the form of an on-line software library. Please note that all code included in the library is experimental. Users are welcome to make use of the library code at their own risk, with the understanding that some or all of these functions and macros may not be supported in future releases.

The *Lisp library code is available in the directory

```
/cm/starlisp/library/f6000/*
```

on UNIX front ends, and in the directory

```
host:>**>cmoptional>starlisp>library>f6000>*.*.*
```

on Lisp machine front ends.

On-line documentation for the library functions and macros is available in the file **documentation.text** in the *Lisp library directory. Ask your system administrator to help you locate the library files at your site.

All functions in the library are defined to autoload on demand. When any one function in a given interface file is autoloaded, the rest of the functions in that interface file are also autoloaded.

Consult the latest edition of the **Lisp Release Notes* for information about the current contents of the *Lisp library.

8.21 Visualization of Data in *Lisp

*Lisp also gives you access to software tools you can use to visually display the results of data parallel programs. There are two basic ways of displaying CM data:

- On a color monitor attached to a CM via a framebuffer I/O module—the combination of the monitor and the framebuffer I/O module is usually referred to simply as the *framebuffer*
- On a workstation that supports the X Windows System, Version 11 interface (referred to as an *X11 window*), either locally or over a network

The framebuffer provides higher resolution and faster display, but you must be attached to a CM sequencer that contains a framebuffer module in order to use it. With the X Windows interface, you can display your output on any standard graphics workstation, either local or remote. You can also use the X Windows interface to develop software on your local workstation that you can later display on a framebuffer.

The following software is available for visualization and graphic display:

- ***Graphics.** *Graphics (pronounced "star-graphics") is the *Lisp interface to the Generic Display and *Render utilities described below. It also includes display functions that take care of positioning, rescaling, and dithering data automatically, as well as a symbolic interface for defining color maps. The rendering functions provide *Lisp wrappers for many of the Paris *Render functions.
- **Generic Display Interface.** The Generic Display Interface provides a single user interface to all CM framebuffers and X Windows servers available from your workstation. It allows an application to present a menu of the available displays. Its routines support the creation, initialization, and selection of a display, writing to and reading from the display, and the control of display offset parameters and color maps. With the appropriate image data, these routines let you visualize data through an X11 window when a framebuffer is not available, or to preview an image locally during development and then easily switch to the framebuffer for final revisions and viewing.
- ***Render.** *Render (pronounced "star-render") is a CM library containing routines that support graphics processing on the Connection Machine. Using these routines, you can draw simple graphics primitives that are placed in a buffer field in CM memory. You can then transfer this image to a framebuffer or X11 window for display. *Render is intended as a building block for more advanced visualization tools.

- **Graphics Display Library.** The graphics display library contains routines for using the framebuffer. It includes facilities for displaying images, panning and zooming the screen, double buffering, and loading the color maps.
- **Xcm.** Xcm provides routines that support the display of image data from CM memory in an X11 window. Normally, you use these routines via the Generic Display Interface, rather than calling them directly.

For complete information on CM visualization tools, see the volume *Connection Machine Graphic Display System* in the Connection Machine documentation set. Documentation for *Graphics is contained in the volume *Programming in *Lisp* in the Connection Machine documentation set.

8.22 CM I/O Programming from *Lisp

The Connection Machine system has a file system associated with it. This file system is separate from the front end's file system, although it shares many similarities with a UNIX file system. You can use the CM file system (CMFS) to store data for the CM.

You create files in the CM file system either by using one of these commands to copy in data or a file from another file system, or by issuing library calls from within a program. All the DataVault I/O operations may be called directly as functions from within *Lisp programs. For information on I/O library calls that use the CM file system, consult the *CM I/O System Programming Guide* in the Connection Machine documentation set.

8.23 CM Scientific Software

*Lisp also provides access to the Connection Machine Scientific Software Library (CMSSL), a library of specialized scientific and mathematical routines. For more information on the CMSSL, and for information on calling CMSSL routines from *Lisp, refer to the *CMSSL for *Lisp* portion of the volume *Connection Machine Scientific Software Library* in the Connection Machine documentation set.

8.24 Managing Large File Sets

An additional feature of Connection Machine System Software, accessible through *Lisp, is DFS, the Def File Set system. This system allows you to define groups of files as belonging to "file sets." All the files in a file set can then be compiled and/or loaded together in a single operation. Individual files can contain either Lisp code or plain text, and the order in which the files are compiled and/or loaded can be determined by the user. DFS is portable among Lisp machines, VAX Lucid, and Sun-4 Lucid.

DFS is described in detail in Appendix B. NOTE: DFS is unsupported software and is subject to change at any time. Users are welcome to use DFS functions, with the understanding that the functions may be changed without notice in future releases.

Lisp machine users already have access to a more robust, supported system of this kind in the Symbolics System Construction Tool system. We recommend that Lisp machine users use SCT rather than DFS to manage large sets of files, if portability is not an issue.

Appendixes

Appendix A

Back-Compatibility Mode

Starting with Version 5.0, CM System Software has allowed user programs to control the number of virtual processors used and the geometry in which they are laid out. Different sets of virtual processors, each with its own geometry, can coexist in a program, and they can be created and destroyed as execution proceeds.

Programs developed under previous CM System Software releases, as well as programs written subsequently in certain languages, require that users configure the number of virtual processors and their configuration when they cold boot the CM. You must execute such programs in *back-compatibility mode*. This appendix describes how to do this.

A.1 Executing in Back-Compatibility Mode

Your program executes in back-compatibility mode if you specify any of the following options to the **`cmattach`** command:

- **`-b`**

  Use **`-b`** to specify the amount of memory to be allocated for back-compatibility mode. A certain amount of memory must be always reserved for the overhead needed to run in back-compatibility mode. The value is a decimal fraction between .1 and .9; the default is .75. See Section A.2 on page 184 for more information.

- **`-v`**

  Use the **`-v`** option to configure the CM to have a specified number of virtual processors. The number can never be less than the number of physical processors and must be a power of 2. You can specify this number either as an integer (for example, **`32768`**) or in the form **`32k`**.

The processors are laid out in a 2-dimensional grid. If the number of virtual processors is the same as the number of physical processors, the processors have the following layout:

4K processors	64x64
8K	64x128
16K	128x128
32K	128x256
64K	128x512

- **-x** and **-y**

  Use the **-x** and **-y** options instead of the **-v** option to configure the virtual processors in a 2-dimensional grid. Each dimension must be a power of 2 and must be at least as large as the number of physical processors in that dimension, as shown above.

If you want to use back-compatibility mode from a batch request, you must use a batch queue that does not automatically attach you to the CM. See your system administrator if such a queue does not exist. If it does exist, you can put the appropriate **cmattach** command in your script file and submit the file to this queue via the **qsub** command.

To change the number of virtual processors or their geometry, you can issue the **cmcoldboot** command, using the same options as described above for **cmattach**. You can, if you wish, omit the **-v** or the **-x** and **-y**, and simply specify the number of virtual processors as either a single integer or two integers.

A.2 Memory Allocation in Back-Compatibility Mode

Certain features of a program may affect the value that you should supply as the **-b** option to **cmattach** or **cmcoldboot** when the program is executed.

This section assumes that you have some familiarity with the virtual processor mechanism described in the *Paris Reference Manual.*

Back-Compatibility and VP Sets

When cold booted in back-compatibility mode, the system creates a VP set with the geometry implied by the command line options `-x` and `-y` or `-v`. It then allocates the amount of memory specified by the `-b` value to a single field in that VP set. The single VP set and the one allocated field that are created for back-compatibility mode are called "VP set 0" and "field 0" for the purposes of this discussion.

The user can, however, create additional VP sets—of any legal geometry—by means of direct calls to the appropriate Paris instructions. For these VP sets, storage is handled in the normal way: that is, a single stack is shared among all user-created VP sets. Only VP set 0 has memory specifically allocated for it.

The Size of the Back-Compatibility Field

Creating new VP sets, or using certain Paris instructions from within VP set 0, may require the user to adjust the fraction of memory allocated for back-compatibility—that is, the relative size of field 0. Some considerations in deciding on the value of the `-b` option are:

- All automatic and scalar variables allocated by C* and CM Fortran programs running in back-compatibility mode are allocated *within* field 0. A C* or CM Fortran program that contains no direct calls to Paris can be executed with the `-b` value set relatively high.
- All Paris instructions called while a user-defined VP set is the current VP set use storage *outside* field 0. A C* or CM Fortran program that creates additional VP sets may require that the `-b` value be lowered.
- Most Paris instructions called while VP set 0 is the current (or only) VP set allocate variables *within* field 0. A C* or CM Fortran program that calls Paris instructions exclusively from within VP set 0 can be executed with the `-b` value set relatively high.
- There are exceptions, however. The following instructions allocate temporary storage *outside* field 0, regardless of which VP set is current when they are called:

  `CM_get_1L`
  `CM_read_from_news_array`
  `CM_write_to_news_array`
  Some **CM_send_** instructions when the message sent exceeds 128 bits in length

- A program that calls these instructions—either from within VP set 0 or from within a user-defined VP set—may require that the `-b` value be lowered.

Most Paris instructions have modest requirements, if any, for temporary storage. However, a few instructions—`CM_get_1L` in particular—are comparatively demanding of temporary storage. If at any time the available temporary memory is exhausted, the system signals a run-time error and program execution terminates. If this event occurs, you can make more temporary space available by lowering the `-b` value.

A.3 Back-Compatibility Mode and Timesharing

Programs that must be executed under back-compatibility mode can execute under timesharing. However, the overhead required for running back-compatibility mode can cause a considerable decrease in performance.

Appendix B

DFS: Defining File Sets

NOTE: The system described in this document is *not* officially supported and is subject to change at any time. Users are welcome to make use of the functions described below, with the understanding that the features described herein may be modified without notice in future releases.

B.1 DFS — Defining File Sets

This document describes DFS, the Def File Set system. This system allows you to define groups of files as "file sets." All the files in a file set can then be compiled and/or loaded together in a single operation. Individual files can contain either Lisp code or plain text, and the order in which the files are compiled and/or loaded can be determined by the user.

The basics of file sets and file set definition files are described in this document. For more information, execute the function **`(dfs:dfs:help)`**, or read through the file **`/cm/dfs/documentation.lisp`**, in the DFS system file set.

The following DFS operations are documented:

`dfs:def-file-set`	Define a file set.
`dfs:def-file-set-directory`	Define the directory in which a file set is stored.
`dfs:load-file-set`	Loads all files in a file set.
`dfs:compile-load-file-set`	Compiles and loads all files in a file set.
`dfs:load-n`	Loads a single file from a file set.

The purpose and use of file set definition files is also described.

B.2 Defining File Sets

A "file set" is simply a group of files stored in the same directory that must be loaded and/or compiled as a group in a specific order.

The **dfs:def-file-set** macro defines a file set.

```
dfs:def-file-set ( name (:directory default-pathname ) )
              &rest filespecs
```

A sample call to **dfs:def-file-set** looks like:

```
(dfs:def-file-set ( my-file-set (:directory
                  "~username/my-code/") )
     "definitions"
     "macros"
     "other-defs"
     "main-program")
```

This form defines a file set that contains four files: **definitions**, **macros**, **other-defs**, and **main-program**. All of these files are located in the directory **~username/my-code/**.

The **dfs:def-file-set** macro has many options to control file compilation and loading, including options that allow simple conditional compilation of files. These options are described in the next section.

B.2.1 Arguments to dfs:def-file-set

The *name* argument must be a symbol. It specifies the name of the file set for DFS. (The package of the symbol is ignored; only the name of the symbol is seen by DFS.)

The *default-pathname* must be a string containing a directory pathname. This pathname is used as the default path for the files specified by the *filespecs* arguments. Note that DFS currently requires that all files in the file set must be stored in this directory.

The *filespecs* arguments specify the files in the file set. The simplest way to specify a file is by a string that contains just the name of the file *without* any extension such as **.lisp** or **.bin**. (DFS automatically adds appropriate type extensions where needed.)

Files that are to be compiled and/or loaded only in specific circumstances may be specified as a list of the form (*filename conditions*). In this case, *filename* is a string containing the name of a file as described above, and *conditions* is any number of keywords that conditionalize the compilation and/or loading of that file.

Files specified with no conditions are compiled and loaded in order. Files with *conditions* may or may not be compiled or loaded, as determined by the *conditions.*

The permissible *conditions* are:

`:compile`	Compile the file only if it has been modified since it was last compiled. Note that **`:compile`** only compiles the file; the **`:load`** option must be specified as well to load the file.
`:load`	Load the file only if it has been modified and/or recompiled since it was last loaded.
`:compile-load`	This option is the same as specifying both **`:compile`** and **`:load`**, and is the default if no *conditions* are specified.
(**`:compile`** *filenames*)	Compile the file if any of the files specified by the *filenames* strings have been recompiled.
(**`:load`** *filenames*)	Load the file if any of the files specified by the *filenames* strings have been reloaded.
(**`:compile-load`** *filenames*)	This option is a combination of the preceding two, and specifies that the file is compiled and loaded conditionally.
`:always`	This keyword may replace *filenames* in the above options, and specifies that the file is always loaded or compiled.
`:read`	Read only the source version of the file. (Useful for files of code that are never compiled.)
`:no-load`	Do not compile or load the file. (Useful for text files and documentation files that are part of a file set but contain no compilable code.)

(**:external** *function-name*)

Define a function with the specified function-name that will return the full pathname of this file when called. (Useful for files specified as **:no-load** that may conditionally be loaded by other files in the file set.)

For example, here is the above **dfs:def-file-set** call modified to make use of conditions:

```
(dfs:def-file-set ( my-file-set (:directory "~username/my-code/") )
    ("definitions" :load :always)
    ("macros" :compile-load)
    ("other-defs" :no-load (:external other-defs-filename))
    ("main-program" (:compile "macros") :load)
```

This **dfs:def-file-set** call defines **my-file-set** with the following conditions:

- The **definitions** file is never compiled, but is always loaded, even if its source file has not been modified.
- The **macros** file is compiled and loaded only if it has been modified.
- The **other-defs** file is never loaded, but **dfs:def-file-set** defines a function named **other-defs-filename** that, when called, will return the pathname of this file. This function can be used elsewhere to load the file (within **main-program** itself, for example).
- The **main-program** file is compiled and loaded whenever the **macros** file has been compiled.

B.3 File Set Definition Files

The **dfs:def-file-set** macro is almost never called directly. Instead, it is stored in a special file known as a *definition file.*

The definition file for a file set is typically stored in the same directory as the files in the file set, and is typically named **def-file-set.lisp**. (This file can be located elsewhere, however, and can have a different name. See Section B.4 on page 191.)

The definition file is used by DFS to determine which files are part of the file set, and also to determine the order in which those files are to be compiled and/or loaded.

B.4 Defining File Set Directories

Another important macro is **`dfs:def-file-set-directory`**. This macro defines the directory in which the definition file of a file set is located.

```
dfs:def-file-set-directory file-set-name def-file-pathname
```

The *file-set-name* argument is the name of the file set, and must be a symbol.

The *def-file-pathname* argument is a string containing the pathname of the definition file. This may be a complete pathname that specifies the directory and name of the file, or a partial pathname that specifies only the directory. In the latter case, DFS assumes by default that the name of the file is **`def-file-set.lisp`**.

A call to this macro for **`my-file-set`** might look like:

```
(dfs:def-file-set-directory my-file-set
 "~username/my-code/")
```

This form informs DFS that the file set **`my-file-set`** is defined by a definition file named **`def-file-set.lisp`**, located in the directory **`~username/my-code/`**.

B.5 File Set Directory Definition Files

Like **`dfs:def-file-set`**, the **`dfs:def-file-set-directory`** macro is almost never called directly. Instead, it is stored in your system's site directory in a file that is named after the file set and that has the extension **`.dfs`**.

For example, the **`dfs:def-file-set-directory`** call shown above would be stored in a file named **`my-file-set.dfs`** in your system's site directory.

B.6 Finding Your Site Directory

The variable **`dfs::*site-file-directory*`** contains the pathname of your local site directory, and can be either a single string or a list of strings.

For UNIX users the site directory is typically **`/cm/dfs/site`**.

For Lisp machine users, the site directory is typically **`>cm>dfs>site`**.

B.7 How DFS Handles File Sets

When a DFS function is called to load a particular file set, DFS first checks to see whether that file set is known to it already. (This is the case, for example, if a DFS function has previously been called to compile or load the file set.)

If the file set is known, DFS reads the definition file for the file set and uses the information contained in it to load the files of the file set.

If the file set is not known to DFS (for example, if it has not been loaded before), then DFS searches the site directory for a **`.dfs`** file named after the requested file set. If there is more than one site directory (that is, if the value of **`dfs::*site-file-directory*`** is a list of strings), the directories are searched in order.

If a **`.dfs`** file is found, DFS loads it, and uses the **`dfs:def-file-set-directory`** form within to locate the definition file for the file set. DFS then loads the file set as usual. If no **`.dfs`** file is found, DFS cannot load the file set, and will signal an error.

B.8 Loading and Compiling File Sets

The DFS functions used to compile and load file sets, and to compile and load individual files of those file sets, are described in the following sections.

All files compiled by DFS are compiled using the Common Lisp compiler.

NOTE: When using DFS, never load or compile any of the files in a file set by any other means than the DFS operators described here (for example by using the Common Lisp **`load`** or **`compile-file`** functions). DFS maintains its own information about whether the files in a file set have been compiled and/or loaded. Compiling or loading files by other means can invalidate this information, causing DFS to perform incorrectly.

If it is necessary to recompile or reload a single file, use the **dfs:load-n** operator described in Section B.9 on page 195.

B.8.1 Loading File Sets

The DFS function to load a file set is **dfs:load-file-set**.

```
dfs:load-file-set file-set-name &key :reload
```

This function loads all of the files in a file set that are not currently loaded.

A sample call to **dfs:load-file-set** looks like:

```
(dfs:load-file-set 'my-file-set)
```

This example causes the three files in **my-file-set** to be loaded in order.

Keyword Arguments to dfs:load–file–set

The **:reload** keyword argument controls whether files are loaded if they have been loaded previously. It can have either of the following values:

:if-not-loaded	Only load files if they have not been loaded previously.
t	Load all files unconditionally.

The default value for the **:reload** argument is **:if-not-loaded**.

B.8.2 Compiling/Loading File Sets

The DFS function to compile and/or load a file set is **dfs:compile-load-file-set**.

```
dfs:compile-load-file-set file-set-name
    &key :recompile :reload :source-only :selective
```

This function compiles and loads each of the files in a file set that have been recently modified.

A sample call to this function looks like:

```
(dfs:compile-load-file-set 'my-file-set)
```

This example causes any of the files in **`my-file-set`** that have been recently modified to be compiled and loaded.

Keyword Arguments to dfs:compile–load–file–set

The **`:recompile`** keyword argument controls whether files are compiled if they have been compiled previously. It can have any one of the following values:

`t`	Compile all files unconditionally.
`:if-changed`	Compile files only if they have not been compiled previously.
`nil`	Do not compile any files.

The default value for the **`:recompile`** argument is **`:if-changed`**.

The **`:reload`** keyword argument controls whether files are loaded if they have been loaded previously. It can have either of the following values:

`t`	Load all files unconditionally.
`:if-not-loaded`	Load files only if they have not been loaded previously.
`nil`	Do not load any files.

The default value for the **`:reload`** argument is **`:if-not-loaded`**.

The **`:source-only`** keyword argument controls whether source or object files are loaded:

`t`	Load only source versions of files.
`:when-no-object`	Load compiled files if available, otherwise load source files.
`nil`	Load only compiled versions of files.

The default value for the **`:source-only`** argument is **`:when-no-object`**.

The **`:selective`** keyword argument controls whether files are loaded selectively.

t	Query user whether to compile and/or load each file.
nil	Compile/load all files without querying.

The default value for the **:selective** argument is nil.

The **:verbose** keyword argument controls whether files are compiled/loaded verbosely.

t	Display messages as files are processed.
nil	Process files silently.

The default value for the **:verbose** argument is **t**.

B.9 Loading Individual Files

The DFS function to compile and load a single file from a file set is **dfs:load-n**.

```
dfs:load-n filename &key :recompile :reload
```

This fuction compiles and loads a single file from a file set.

A sample call to **dfs:load-n** looks like:

```
(dfs:load-n "~username/my-code/main-program")
```

This example compiles the file **main-program** from **my-file-set** (if it was recently modified), and then loads the file.

B.9.1 Keyword Arguments to dfs:load–n

The **:recompile** keyword argument controls whether the file is compiled if it has been compiled previously.

The **:reload** keyword argument controls whether the file is loaded if it has been loaded previously.

The **:verbose** keyword argument controls whether files are loaded verbosely.

The legal values and default for these arguments are the same as for the corresponding arguments of **`dfs:compile-load-file-set`**.

Appendix C

Paris Functions Affecting Timesharing Performance

The Paris functions listed in this appendix cause the CM and the front end to resynchronize, potentially causing slower performance when running under timesharing from a VAX front end. If your program will be running under timesharing from a VAX, you should minimize calls to these functions.

```
/*
 * PARIS REL3 functions
 */

CM_complex_t CM_c_read_from_processor_1L _AP((CM_sendaddr_t
send_address_value,
 CM_field_id_t source,
 unsigned s,
 unsigned e));
double CM_f_read_from_processor_1L _AP((CM_sendaddr_t
send_address_value,
 CM_field_id_t source,
 unsigned s,
 unsigned e));


CM_complex_t CM_global_c_add_1L _AP((CM_field_id_t source,
 unsigned s,
 unsigned e));
unsigned CM_global_count_bit _AP((CM_field_id_t source));

unsigned CM_global_count_bit_always _AP((CM_field_id_t
source));

unsigned CM_global_count_context _AP((void));

unsigned CM_global_count_overflow _AP((void));

unsigned CM_global_count_test _AP((void));

double CM_global_f_add_1L _AP((CM_field_id_t source,
 unsigned s,
 unsigned e));

double CM_global_f_max_1L _AP((CM_field_id_t source,
 unsigned s,
 unsigned e));

double CM_global_f_min_1L _AP((CM_field_id_t source,
 unsigned s,
 unsigned e));

int CM_global_logand_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_logand_bit _AP((CM_field_id_t source));

unsigned CM_global_logand_bit_always _AP((CM_field_id_t
source));

unsigned CM_global_logand_context _AP((void));

unsigned CM_global_logand_overflow _AP((void));

unsigned CM_global_logand_test _AP((void));
```

```
int CM_global_logior_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_logior_bit _AP((CM_field_id_t source));

unsigned CM_global_logior_bit_always _AP((CM_field_id_t
source));

unsigned CM_global_logior_context _AP((void));

unsigned CM_global_logior_overflow _AP((void));

unsigned CM_global_logior_test _AP((void));

unsigned CM_global_logxor_1L _AP((CM_field_id_t source,
 unsigned len));

int CM_global_s_add_1L _AP((CM_field_id_t source,
 unsigned len));

int CM_global_s_max_1L _AP((CM_field_id_t source,
 unsigned len));

int CM_global_s_min_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_u_add_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_u_max_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_u_max_s_intlen_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_u_max_u_intlen_1L _AP((CM_field_id_t source,
 unsigned len));

unsigned CM_global_u_min_1L _AP((CM_field_id_t source,
 unsigned len));

int CM_s_read_from_processor_1L _AP((CM_sendaddr_t
send_address_value,
 CM_field_id_t source,
 unsigned len));


double CM_timer_read_cm_busy _AP((unsigned timer));

double CM_timer_read_cm_idle _AP((unsigned timer));

double CM_timer_read_elapsed _AP((unsigned timer));

int CM_timer_read_run_state _AP((unsigned timer));

int CM_timer_read_starts _AP((unsigned timer));

unsigned CM_u_read_from_processor_1L _AP((CM_sendaddr_t
send_address_value,
 CM_field_id_t source,
 unsigned len));


/*
 * Back compatibility functions.
 */
unsigned CM_enumerate_and_count _AP((CM_memaddr_t, unsigned));
double CM_f_read_from_processor _AP((CM_cubeaddr_t,
CM_memaddr_t, unsigned, unsigned));
long CM_global_add _AP((CM_memaddr_t, unsigned));
long CM_global_count _AP((CM_memaddr_t));
long CM_global_count_always _AP((CM_memaddr_t));
double CM_global_f_add _AP((CM_memaddr_t, unsigned, unsigned));
double CM_global_f_max _AP((CM_memaddr_t, unsigned, unsigned));
double CM_global_f_min _AP((CM_memaddr_t, unsigned, unsigned));
unsigned CM_global_logand _AP((CM_memaddr_t, unsigned));
unsigned CM_global_logand_always _AP((CM_memaddr_t, unsigned));
unsigned CM_global_logior _AP((CM_memaddr_t, unsigned));
unsigned CM_global_logior_always _AP((CM_memaddr_t, unsigned));
long CM_global_max _AP((CM_memaddr_t, unsigned));
long CM_global_min _AP((CM_memaddr_t, unsigned));
unsigned CM_global_u_add _AP((CM_memaddr_t, unsigned));
unsigned CM_global_u_max _AP((CM_memaddr_t, unsigned));
unsigned CM_global_u_min _AP((CM_memaddr_t, unsigned));
long CM_read_from_processor _AP((CM_cubeaddr_t, CM_memaddr_t,
unsigned));
CM_timeval_t * CM_stop_timer _AP((int));
unsigned CM_u_read_from_processor _AP((CM_cubeaddr_t,
CM_memaddr_t, unsigned));
```

Appendix D

The UNIX System for CM Users

This appendix presents brief explanations of features of the UNIX operating system that are important to users of the Connection Machine system. For a more comprehensive discussion of the UNIX system, consult *The UNIX Programming Environment*, by Brian W. Kernighan and Rob Pike (Prentice-Hall, 1984), or one of the many other books written about UNIX.

absolute pathname	See *pathname*.
background process	A process that runs "in the background," allowing you to issue other commands while it is executing.
`.cshrc`	In the C shell, a script file run after login to set up the characteristics of the shell.
Bourne shell	See *shell*.
C shell	See *shell*.
current directory	See *directory*.
directory	A node in the UNIX file system. A directory can contain files and other directories. The *current* or *working* directory is the directory to which relative pathnames refer.
environment variables	Variables whose settings are available both to a shell and to programs called from within the shell. You can change the settings of these variables to provide information about your environment to programs. CM

	system software provides various environment variables for use with CM commands. For example, the setting of the environment variable **CMINTERFACE** specifies a default front-end bus interface for use with the **cmattach** command. Compare *shell variables.*
filename	The name of a UNIX file. See also *pathname.*
group ID	The name of a class of users to which a user is assigned.
hostname	The name assigned to a computer running the UNIX system.
kernel	The program that controls the resources of the computer. A user interacts with the UNIX kernel by means of a *shell.*
Korn shell	See *shell.*
login ID	The name by which a user is known to the system.
make utility	A utility that provides a mechanism for maintaining programs by ensuring that the files constituting a program all exist and are up-to-date.
pathname	A name that includes all the directories that have to be traversed to reach a given file or directory. An *absolute* pathname starts with root—that is, at the beginning of the file system hierarchy: for example, **/usr/bin.** A *relative* pathname starts with the working directory: for example, **mydirectory/my_subdirectory**.
permissions	Attributes associated with a file that determine who can do what with the file.
pipeline	A sequence of commands in which the output of one command is the input of another.

process	An instance of a running program. Each process in a system has a unique process-id. Multiple processes can be assigned to the same process group, so that a single signal can be sent to them all at the same time.
prompt	A symbol that indicates that the system is ready to accept commands. You can use a shell variable to set what your prompt will be. In this guide, the prompt is displayed as a percent sign (%).
`rcp`	See *remote operations.*
redirection	A method of designating that the source (or destination) of input to a command (or output from a command) is to be a named file or device.
relative pathname	See *pathname.*
remote operations	Commands that involve interaction with UNIX systems other than the local system to which you are logged in. The **`rlogin`** command allows you to log in to a remote UNIX system; the **`rsh`** command allows you to execute a UNIX command on a remote system without logging in; and the **`rcp`** command allows you to copy a file to or from a remote system.
`rlogin`	See *remote operations.*
root	The beginning directory in the hierarchy of the UNIX file system—specified as `/`.
`rsh`	See *remote operations.*
script file	A file that contains commands or programs to be executed. You can submit a script file for execution by the CM batch system. Also called *shell script.*
`setenv`	The C shell command for setting an environment variable.

shell	A command interpreter that lets you issue commands to be executed by the kernel. There are different shells that provide slightly different features. The C shell, the Bourne shell, and the Korn shell are popular UNIX shells. You can create *subshells* within a shell; for example, the **`cmattach`** subshell is created as a subshell within the shell from which you issued the **`cmattach`** command.
shell script	See *script file.*
shell variables	A set of predefined variables whose values you can change to customize your shell. For example, you can set the **`prompt`** variable to change your UNIX prompt. Compare *environment variables.*
signal	A communication device that informs a process of an event. For example, NQS may send a SIGTERM signal to processes that are executing when a queue is about to detach from its CM resource. Processes may contain *signal handlers* that determine what to do when a particular signal is received.
standard input, output, and error	*Standard input* is the input device for commands. *Standard output* is the device to which commands send their results. *Standard error* is the device to which commands send error messages. Typically, all three are defined to be your terminal. You can change this—for example, by using redirection to send output to a file instead of to your terminal.
subshell	See *shell.*
superuser	A special user on a UNIX system who can read or modify any file in the system.
symbolic link	An entry in a directory that points to an already existing file on a different file system. This allows a user to gain access to a file without specifying an absolute pathname.

user ID	A number associated by the system with a login ID.
working directory	See *directory.*

Appendix E

Glossary

This is a glossary of CM-specific terminology used in this manual.

batch access	Access to the CM obtained by submitting a batch request via the **qsub** command. Compare *direct access.*
batch queue	In NQS, a queue for batch requests.
batch request	A job submitted via the **qsub** command to an NQS batch queue.
C*	A data parallel extension of the C programming language.
CM	Used loosely for the Connection Machine system. Also refers specifically to the *parallel processing unit.*
cmattach subshell	An interactive UNIX shell created when **cmattach** is issued without the name of an executable program. A user can issue CM and UNIX commands and run programs from this shell.
CM Fortran	An implementation of the Fortran 77 programming language, extended with array-handling facilities from Fortran 90.
CMFS	The Connection Machine file system.

CMIO bus	An I/O bus that provides high-speed data transfer among the components of the CM system.
CMIOC	Connection Machine I/O Controller. A board that connects a section of the parallel processing unit to the CMIO bus.
C/Paris	The C-language interface to the Paris instruction set.
CMSSL	CM Scientific Software Library. A library of routines that perform data parallel versions of standard mathematical operations.
CM system	An integrated combination of hardware and software designed for high-speed data parallel computing.
DataVault	A mass storage system for data in the CM system.
direct access	Access to the CM via the `cmattach` command. Compare *batch access.*
exclusive access	Access to the CM in which only one user can be attached to a FEBI and a sequencer at a time.
FEBI	Front-end bus interface. A board that provides an interface between a front end and a CM parallel processing unit.
Fortran/Paris	The Fortran interface to the Paris instruction set.
framebuffer module	A board that connects an I/O channel of a parallel processing unit to a high-resolution graphic monitor. *Framebuffer* is often used loosely to refer to the *graphic display system.*
front end	A standard serial computer that provides the user's link to the CM system.
front-end bus interface	See *FEBI.*

Generic Display Interface	A CM software product that provides a single user interface to all CM framebuffers and X Window servers available on a system.
graphic display system	Part of the CM system that lets users quickly visualize large data sets. It consists of the framebuffer module and a high-resolution color monitor.
Graphics Display Library	A CM library that contains routines for using the framebuffer.
grid communication	Communication among CM processors in which processors communicate with their neighbors in a multidimensional grid.
interface	See *FEBI.*
*Lisp	A data parallel extension of Common Lisp.
Lisp/Paris	The Lisp interface to the Paris instruction set.
NEWS communication	An alternative term for *grid communication.* "NEWS" refers to the four directions—North, East, West, and South—of a 2-dimensional grid.
nexus	A bidirectional switch that enables any front end to be attached to any section, or valid combination of sections, of a parallel processing unit.
NQS	Network Queueing System. The batch system on which the CM batch system is based.
Paris	The CM parallel instruction set. Users can call Paris instructions from Fortran, C, Lisp, or the high-level data parallel languages.

parallel processing unit	The part of the Connection Machine system that contains the parallel processors. Also referred to loosely as the *CM*.
pipe queue	In NQS, a queue that sends batch requests to other queues.
pvar	A Lisp object that represents a collection of values stored one per processor in the CM.
*Render	A CM software library whose routines support graphics processing on the CM.
request-id	In NQS, an identifier for a batch request, consisting of a sequence number and a hostname (or STDIN).
router	A high-speed communication device that interconnects processors in the CM.
section	Part of a CM that can be treated as a separate parallel processing unit. Separate sections have their own sequencers, routers, and I/O channels.
sequence number	In NQS, a number assigned to a batch request.
sequencer	A device that controls the individual processors within a parallel processing unit.
VMEIO computer	A VMEbus computer that contains a special interface board that connects it to a CMIO bus. Other devices, such as magnetic tape drives, can be connected to this computer, and thereby to the CM system.
virtual processors	"Processors" created by dividing up the memory of physical CM processors. The system automatically creates virtual processors if a program requires more processors than are actually available in the parallel processing unit.

Xcm	A CM software library containing routines that support the display of image data from CM memory in an X11 window.

Appendix F

Man Pages

This appendix contains UNIX man pages for some of the user commands discussed in this manual.

To obtain on-line documentation for a CM command or functional call, issue a command with this format:

```
% cmman name
```

NAME

cm – obtain information about the CM to which you are attached via a **cmattach** subshell

SYNOPSIS

cm [–C] [–d] [–i] [–S]

OPTIONS

–C	prints the name of the CM to which this subshell is attached.
–d	prints the name of the CM device driver. This is always **/dev/cm**.
–i	prints the number of the interface to which this subshell is attached.
–S	prints the number(s) of the sequencer(s) to which this subshell is attached.

IDENTIFICATION

Connection Machine System Software Release 6.1.

SEE ALSO

Thinking Machines Corporation, *CM User's Guide*.

NAME

cmattach – allocate Connection Machine processors

SYNOPSIS

cmattach [*flags, CM options, and front end options*] [*program* [*args* ...]]

DESCRIPTION

cmattach allocates a Connection Machine resource to a user. The various options are described below. When *cmattach* is executed without options specifying the characteristics of the CM resource to which you want to attach, it attaches to the highest-numbered sequencer that is free on the lowest-numbered interface available. If the *cmattach* succeeds, a *cold boot* (an industrial-strength reset of the allocated portion of the system) is automatically performed.

If no program name appears on the *cmattach* command line, the requested processors are allocated and configured, a cold boot is performed, and an interactive shell is spawned from which the user can run programs that access the Connection Machine system. The processors are freed and made available to other users when the subshell is exited (typically by typing control-D or "exit"). Note that if you are already in a *cmattach* subshell when invoking *cmattach* without a program argument, no further subshells are created; this lets you change the allocation of physical processors without relinquishing the interface.

If you give a program name, and possibly arguments for the program, the requested processors are allocated and configured, a cold boot is performed, and the program is run. The processors are deallocated when the program exits.

Flags that affect *cmattach*'s behavior are:

–e requires that the requested CM resource be operating in exclusive mode--that is, not under timesharing.

–h prints a help message. All other arguments provided are checked for legality, but otherwise ignored.

–I*<account-ID>*
indicates the ID under which the program(s) are accounted. <account-ID> is a string of up to 8 characters. The value specified by -I overrides the setting of the CM_ACCOUNT_ID environment variable (which the autoattaching mechanism uses to obtain a job's account ID).

–n means "don't cold boot." Normally the Connection Machine system is *cold booted* after being attached; this switch causes the cold boot to be skipped.

–q is for "quiet." All informational (non-error) messages are suppressed. This is particularly useful in conjunction with **–w**, when you run the program in the background and you don't really want it writing all over your terminal.

–t requires that the requested CM resource be operating under timesharing.

–w means "wait for resources." Normally *cmattach* will return an error if either the front-end interface to the Connection Machine system or the desired number of Connection Machine processors is not available; this option causes the *cmattach* program to wait (possibly forever) for the desired resources to be freed.

Options to *cmattach* affecting the choice of Connection Machine system are:

–p *nprocs*
Attach a particular number of Connection Machine processors. Legal values are **4096**, **8192**, **16384**, **32768**, and **65536** (which may be expressed as **4k**, **8k**, **16k**, **32k**, and **64k**, respectively), depending on your CM's physical configuration. The **–p** option may not be used in conjunction with the **–S** option.

–S *seqspec*
Attach a particular *sequencer* set. *seqspec* must be one of the following: **0**, **1**, **2**, **3**, **0–1**, **2–3**,

0–3. The **–S** option may not be used in conjunction with the **–p** option.

Options to *cmattach* affecting the initial configuration of the Connection Machine environment are:

–b *fraction*
: Set up the Connection Machine system in *back compatibility mode.* The fraction of memory allocated for back compatibility mode is specified by *fraction*, which must be a decimal fraction less than 1; if no value is specified, the default is 0.75.

–g *axis-length[,axis-length...]*
: Configure the Connection Machine system to have a NEWS-ordered geometry having the number of axes specified, of the lengths specified. A VP set is created in this geometry and made current. Each axis length must be a power of 2, and the total number of processors allocated must be an integer multiple of the number of physical processors. If **–g** is not given, the default depends on the number of physical processors, as follows: 4K processors, 64x64; 8K processors, 64x128; 16K processors, 128x128; 32K processors, 128x256; 64K processors, 128x512.

–u *ucode*
: Load the specified version of the CM microcode. If you omit this option, the latest version of the microcode is loaded.

Options to *cmattach* to help select among multiple interfaces (when more than one hardware interface is installed) are:

–C *CM-name*
: Attach to the Connection Machine system named *CM-name*. Only useful if your front-end machine is connected to more than one Connection Machine system. You need not specify the full Connection Machine name; a leading substring is acceptable.

–cm*n*
: Attach to the specified model of Connection Machine system. Specify **–cm2** to attach to a CM-2 series Connection Machine system. Specify **–cm200** to attach to a CM-200 series CM.

–i *interface*
: Attach to the Connection Machine system via a particular front-end interface. Only useful if you have more than one interface in your front end.

None of these selection options may be combined with one another.

The following arguments are valid only for back compatibility. They cause back compatibility mode to be entered, and 75% of CM memory to be reserved for back compatibility mode.

–v *nvprocs*
: Configure the Connection Machine system to have *nvprocs* virtual processors. The number of virtual processors can never be less than the number of physical processors allocated. The **–v** option may not be used in conjunction with the **–x/–y** option.

–x *xdimension* **–y** *ydimension*
: Configure the Connection Machine to have (*xdimension* * *ydimension*) virtual processors arranged in an *xdimension* by *ydimension* **NEWS** grid. Each dimension must be a power of 2 and must be no less than the number of physical processors in that dimension, as described below. The **–x/–y** option may not be used in conjunction with the –v option.

Virtual processors in back-compatibility mode are always configured in a two-dimensional grid
: called the **NEWS** grid. When the number of virtual processors is the same as the number of physical processors (the default case), the grid dimensions are as follows: 4K processors, 64x64; 8K processors, 64x128; 16K processors, 128x128; 32K processors, 128x256; 64K processors, 128x512.

SAFETY

There is a mode where extra error and consistency checking is performed during the execution of Connection Machine programs. This can be very useful for debugging, but reduces execution speed. *cmattach* establishes a default safety mode for all Connection Machine programs executed in the *cmattach* subshell. If the environment variable CM_DEFAULT_SAFETY has the value 'on' or 'ON', the default safety mode will be to perform the extra checking. In all other cases the extra checking is disabled. The default safety mode can be changed inside the *cmattach* subshell using the *cmsetsafety* command.

EXAMPLES

The following are examples of *cmattach* usage:

cmattach -p16K -x256 -y256 life 1024

Attach 16,384 processors, configuring 65,536 virtual processors in back compatibility mode, in a 256 by 256 grid, and run a program named *life* with one argument, 1024.

cmattach -C fig-newton -p 32k

Allocate 32,768 physical processors on a Connection Machine named **fig-newton**, using the default configuration (128x256), and enter a *cmattach* subshell.

DIAGNOSTICS

All output from the *cmattach* program itself is directed to *stderr.* If a program name is given, the exit code from *cmattach* will be one of the following: 255 (or –1, if you prefer) if the *cmattach* itself failed, 100 if something was killed by a signal, or the actual exit code of the program that was run within the *cmattach.* Note that the 255 and 100 exit codes, along with any incidental messages to *stderr*, are intended to be the only non-transparent aspects of running a program prefixed by *cmattach.*

FILES

/dev/cm??

The Connection Machine interface devices.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cmcoldboot (1), cmdetach (1), cmusers (1), cmfinger (1), cmsetsafety(1).
Thinking Machines Corporation, *CM User's Guide.*
CM System Administrator's Guide.

RESTRICTIONS

If you run a Connection Machine Lisp world (for example, *starlisp*) from within a *cmattach* subshell, CM allocation and configuration information is not passed into the Lisp world–you *must* execute the **cm:attach** and **cm:cold-boot** or ***lisp:*cold-boot** functions from within the Lisp world before attempting to access the Connection Machine system.

NAME

cmcoldboot – reset the Connection Machine system

SYNOPSIS

cmcoldboot [-h] [-g *axis-length*[,*axis-length*...] [-b *fraction*] [-u *ucode*] [[-v] *nvproc* | [-x] *xdimension* [-y] *ydimension*]

DESCRIPTION

cmcoldboot completely resets the state of the hardware allocated to the executing front end, loads microcode, initializes system tables, and clears user memory. It can only be run from inside a *cmattach* subshell (see *cmattach*(1) for details).

cmcoldboot has no effect when the allocated hardware is operating under timesharing. In that case, the system automatically clears memory before allocating it to a process.

If a single number is provided as an argument, it specifies the total number of virtual processors that will be configured in back compatibility mode. If two integers are provided, they specify the desired dimensions of the virtual **NEWS** grid. Each of these dimensions must be a power of two and must be no less than the number of physical processors in that dimension.

If no arguments are provided, the state of back compatibility mode and the number of virtual processors remains unchanged (that is, whatever was configured by a previous *cmattach* or *cmcoldboot* remains in effect).

When the number of virtual processors is the same as the number of physical processors (the default case), the grid dimensions are as follows: 4K processors, 64x64; 8K processors, 64x128; 16K processors, 128x128; 32K processors, 128x256; 64K processors, 128x512. Note that usable memory per virtual processor decreases with the number of virtual processors configured.

Options are:

–h Print a help message. All other arguments are ignored.

–g axis-length,*axis-length*...
Configure the CM system to have a NEWS-ordered geometry having the number of axes specified, of the lengths specified. A VP set is created in this geometry and made current. Each axis length must be a power of 2, and the total number of processors allocated must be an integer multiple of the number of physical processors.

–b *fraction*
Set up the Connection Machine system in *back compatibility mode*. The fraction of memory allocated for back compatibility mode is specified by *fraction*; if no value is specified, 0.75 is the default.

–u *ucode*
Load the specified version of the microcode.

In addition, if only one dimension is provided it may be preceded by a **–v**, or if two dimensions are provided they may be written as "**–x** *xdim* **–y** *ydim*" to be compatible with *cmattach* option format. Note that this format is valid only for back compatibility.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cmattach (1).
Thinking Machines Corporation, *CM User's Guide*.

NAME

cmdetach – detach a Connection Machine user

SYNOPSIS

cmdetach [[-i] *interface* | *user-ID* | *front-end*[:*interface*]] [-c | -C *CMname*] [-s | -S *seqno*]

DESCRIPTION

cmdetach detaches a user from the Connection Machine system.

If you invoke *cmdetach* in a *cmattach* subshell without arguments, you are detached from the Connection Machine system, but remain in the subshell. This is an alternative to exiting from the shell, and may be useful to preserve environmental state. Re-executing *cmattach* will then reconnect you to the Connection Machine system. An error results if you issue *cmdetach* with no arguments and you are not currently in a *cmattach* subshell.

If you provide an integer as an argument, then the user who is attached to the Connection Machine system via the local front end interface that corresponds to the given integer is detached. Use the *cmfinger* command to determine the interface number. The interface number may be preceded by a **–i** for compatibility with other Connection Machine commands.

If the argument provided is the login ID of a user on this front-end machine, that user will be detached from the Connection Machine system. This only works for users attached from this front end; to detach users on other front ends you must specify the name of that front-end system.

If the given argument is the name of another front end, and that host is connected to the same Connection Machine system as the local host, then that front end is logically and forcibly detached from the Connection Machine system, possibly disrupting any ongoing interaction with the Connection Machine system. By appending a colon and an interface number to the host name, you can specify a particular interface on that front end to detach. For example:

```
% cmdetach binky:1
```

Use **-c** or **-C** to detach the specified CM from your front end. Use the **-s** or **-S** option in addition to specify a particular sequencer or sequencer set within the CM; if only one CM is attached, **-s** or **-S** is sufficient. If your front end has more than one interface to the specified CM, you are prompted to use the **-i** option instead.

In all cases of detaching another user, you are asked to confirm the action, and are thereby given the chance to abort the function. *cmfinger* output is displayed to let you know the status of the system. If you are detaching a timesharing user, you are asked twice to confirm the action.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cmattach (1), cmfinger (1).
Thinking Machines Corporation, *CM User's Guide.*

RESTRICTIONS

When users on other Lisp machine front ends are detached they should be so notified, but they aren't. Instead they learn the hard way, when their programs crash in flames.

NAME

cmfinger – show Connection Machine users

SYNOPSIS

cmfinger [–h] [*–in*] [*cm-name* [*cm-name* ...]] [*hostname* [*hostname* ...]]

DESCRIPTION

cmfinger prints a table that provides information about who is using a Connection Machine system. If you do not specify any options, the information is reported for all front ends connected to the same Connection Machine system as the front end from which you issue the command. For example:

```
% cmfinger

CM          Seqs   Size   Front end      I/F    User    Idle     Command
----------------------------------------------------------------------------
GEMSTONE    1      8K     clytemnestra   1      fred    0h 18m  "cmattach"
GEMSTONE    2      8K     christopher    0      barney  0h 00m  "tests"
GEMSTONE    3      8K     alaska         0      betty   0h 27m  "matmul"
GEMSTONE    ---    ---    alaska         1      nobody
                   256K memory, 32-bit floating point
                   framebuffers on sequencers 0 1 (seq 0 is free)
                   CMIOCs on sequencers 0 1 (seq 0 is free)
       1 free seq on GEMSTONE -- 0 -- totalling 8K procs
```

Here the Connection Machine system in question is named Gemstone. It has four sequencers, each with 8K processors. It is connected to three front ends named Clytemnestra, Christopher, and Alaska, which are being used by users named Fred, Barney, and Betty, respectively. The second interface on Alaska is unused. Fred is attached, via interface 1 on Clytemnestra, to sequencer 1 on Gemstone, and is running a *cmattach* subshell; he has been idle for 18 minutes. Barney and Betty are running programs called "tests" and "matmul" on sequencers 2 and 3 of Gemstone; sequencer 0 is available for use from interface 1 on Alaska.

OPTIONS

The –h option prints a help message.

Specify –i, followed by the number of a front-end interface, to obtain information about that interface.

To obtain information about CMs to which your front end does not have an interface, list their names on the *cmfinger* command line.

To obtain information about CMs connected to hosts other than your front end, list their names on the command line. If a hostname is the same as a CM name, *cmfinger* chooses the CM name.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

Thinking Machines Corporation, *CM User's Guide*.

NAME

cmlist – list Connection Machine parallel processing units

SYNOPSIS

cmlist [–d] [–f] [–h *hostname* ...] [–p *nprocs*] [–v] [–0 | –32 | –64]

DESCRIPTION

cmlist lists all CMs listed in the CM configuration file associated with the front-end computer from which the command is issued.

OPTIONS

–d	lists CMs that have DataVaults.
–f	lists CMs that have framebuffers.
–h	lists CMs connected to front ends with the specified hostnames.
–p	lists CMs with the specified number of processors or more. Specify the number as an integer; to indicate thousands of processors, follow the integer with *k* or *K*.
–v	lists CMs connected to VME I/O computers.
–0	lists CMs with no floating-point accelerators.
–32	lists CMs with 32-bit floating-point accelerators.
–64	lists CMs with 64-bit floating-point accelerators.

The options combine, so that specifying **-d** and **-f** lists the names of CMs with *both* a DataVault and a framebuffer.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

Thinking Machines Corporation, *CM User's Guide.*

NAME

cmman – display CMost and UNIX manual pages

SYNOPSIS

cmman [*man-options*]

DESCRIPTION

cmman is a shell script that calls the UNIX *man* command, first temporarily adding to the user's MANPATH the directories containing CMost on-line man pages. This allows users to display CMost man pages without having to edit their MANPATH. *cmman* can also be used to display any UNIX man page in the user's MANPATH.

OPTIONS

cmman passes to *man* any *man* option specified on the *cmman* command line.

IDENTIFICATION

Connection Machine System Software Release 6.1.

SEE ALSO

Thinking Machines Corporation, *CM User's Guide*.

NAME

cmnice – run a command with low CM timesharing priority

SYNOPSIS

cm [*–number*] *command* [*arguments*]

DESCRIPTION

By default, the command is run with a CM scehduling priority of 4, one less than the default priority.

A priority may be specified in the range of 0 to 5 (0 to 10 for the superuser. Useful priorities are: 0 (the affected processes will run only when nothing else in the system wants to), 5 (the "base" scheduling priority) and values greater than 5 (to make things go very fast).

DIAGNOSTICS

cmnice returns the exit status of the subject command.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991-1992 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cmps(1), cmrenice(1)
Thinking Machines Corporation, *CM User's Guide*.

NAME

cmps – list timeshared processes

SYNOPSIS

cmps [–C *cm-name* -S *seqset*]

DESCRIPTION

cmps prints a table that provides information about processes running under Connection Machine timesharing. If you are attached to a sequencer running timesharing, *cmps* lists the processes running on that sequencer. If you are not attached, use -C and -S to specify the correct sequencer(s). However, you must issue the command from the front end that is running the timesharing daemon to obtain information about the processes running under that daemon.

OPTIONS

Use –C to specify the name of a CM.

Use -S to specify the sequencers on the CM. *seqset* can be 0, 1, 2, 3, 0-1, 2-3, or 0-3.

DISPLAY FORMAT

NAME	is the name of the program
PID	is the UNIX process ID of the process.
OWNER	is the name of the user who owns the process.
SIZE	is the number of 1024-bit pages that the process takes up on the CM. Some of these may be swapped out.
RSS	is the number of 1024-bit memory pages that the process is occupying at the moment.
BASE	is the page number in CM memory of the first page of this process when it is running.
PRI	is the current priority at which the process is running. Priority values vary from 0 to 9. A process with a priority of 0 will run only when no other processes want to use the CM. An asterisk indicates that the process is ready to use the CM.
%-RT	is the percent of real time that the process has received over its lifetime. Since processes start at different times, these percentages can add up to more than 100%.
Q	is the current quantum of the process. For the currently scheduled process, this is the amount of time left in this processes quantum. While this is generally a small number (0.25 - 1 second), if the timesharing system has to swap jobs to disk it will increase the scheduling quantum in order to reduce the overhead involved in swapping.
TSR	specifies how long it has been since the process has last run on the CM.
AGE	is the age of the process in minutes:seconds.

The remaining statistics are overall data gathered by the memory managers since the timesharing daemon was started.

IDENTIFICATION

Connection Machine System Software Release 6.1.

SEE ALSO

Thinking Machines Corporation, *CM User's Guide.*

NAME

cmrenice – change the CM timesharing system priority of a running process.

SYNOPSIS

cmrenice [**\-p** *priority*] *process-ID*

DESCRIPTION

cmrenice changes the CM timesharing system priority of a process. Processes with higher priorities will be scheduled more often than processes with low priorities.

OPTIONS

By default, the specified process-id has its priority reduced by one.

–p priority
Set the scheduling priority of the process to *priority*.

Users other than the superuser may alter only the priority of processes they own, and can only change their "nice value" within a range of 0 to 5. (This prevents overriding administrative fiats.) The superuser may alter the priority of any process and set the priority to any value in the range 0 - 9. Useful priorities are: 0 (the affected processes will run only when nothing else in the system wants to), 5 (the "base" scheduling priority) and values greater than 5 (to make things go very fast).

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991-1992 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cmps(1), cmnice(1)
Thinking Machines Corporation, *CM User's Guide.*

NAME

cmsetsafety – set Paris safety mode

SYNOPSIS

cmsetsafety on/off

DESCRIPTION

cmsetsafety is used to set the Paris safety mode. It can only be run from inside a *cmattach* subshell (see *cmattach*(1) for details).

This command sets the initial safety mode for all Connection Machine programs executed in the *cmattach* subshell. If the safety mode is "on," various extra error and consistency checks are performed at the Paris-level interface. The price of these error checks is substantially reduced execution speed at low virtual processor ratios. If the safety mode is "off," minimal error checking is performed at the Paris-level interface.

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cmattach (1)

Thinking Machines Corporation, *CM User's Guide.*

NAME

cmtime – time a command

SYNOPSIS

cmtime *command*

DESCRIPTION

cmtime prints execution timing information about a program using the Connection Machine system. Timing information is displayed on the diagnostic output stream.

cmtime displays information gathered by the CM accounting facility. If the CM accounting daemon is not running, **cmtime** will report only on front-end timing information.

EXAMPLES

The output of **cmtime** is slightly different under timesharing and exclusive modes.

Exclusive mode:

```
% cmtime myprog
Elapsed time: 115.38 sec; CM time: 111.72 sec.
Front end virtual time (seconds): 10.39 user, 1.26 system.
CM utilization: 97%; Front end utilization: 10%
```

Timesharing mode:

```
% cmtime myprog
Elapsed time: 14.25 sec; CM time: 7.53 (out of 13.63) seconds.
Front end virtual time (seconds): 2.04 user, 4.40 system.
CM utilization: 55%; {CM}* share: 96%
```

IDENTIFICATION

Connection Machine System Software Release 6.1.
Copyright © 1991 by Thinking Machines Corporation, Cambridge MA.

SEE ALSO

cm-acctd (8)
Thinking Machines Corporation, *CM User's Guide.*

RESTRICTIONS

CM time is an estimate from the accounting system, obtained by polling the status of the CM sequencer every 1/100th of a second. Users desiring really accurate timings should use the PARIS timing facility: CM_timer_start(), CM_timer_stop(), and CM_timer_print().

NAME

qdel – delete or signal NQS request(s).

SYNOPSIS

qdel [**–k**] [*–signo*] [**–u** *username*] *request-id*

DESCRIPTION

qdel deletes all queued NQS requests whose respective *request-id* is listed on the command line. Additionally, if the flag **-k** is specified, then the default signal of **SIGKILL** (-9) is sent to any running request whose *request-id* is listed on the command line. This will cause the receiving request to exit and be deleted. If the flag *–signo* is present, then the specified signal is sent instead of the **SIGKILL** signal to any running request whose *request-id* is listed on the command line. In the absence of the **-k** and *–signo* flags, *qdel* will **not** delete a *running* NQS request.

To delete or signal an NQS request, the invoking user *must* be the owner--that is, the submitter of the request. The only exception to this rule occurs when the invoking user is the *superuser*, or has NQS operator privileges as defined in the NQS manager database. Under these conditions, the invoker may specify the **–u** username flag, which allows the invoker to delete or signal requests owned by the user whose account name is *username*. When this form of the command is used, **all** *request-ids* listed on the command line are presumed to refer to requests owned by the specified user.

An NQS request is always uniquely identified by its *request-id*, no matter where it is in the network of the machines. A *request-id* is always of the form: *seqno* or *seqno.hostname*, where *hostname* identifies the machine where the request was originally submitted, and *seqno* identifies the sequence number assigned to the request on the originating host. If the *hostname* portion of a *request-id* is omitted, then the local host is always assumed.

The *request-id* of any NQS request is displayed when the request is first submitted (unless the *silent* mode of operation for the given NQS command was specified). The user can also obtain the *request-id* of any request through the use of the *qstat*(1) command.

CAVEATS

When an NQS request is signalled by the methods discussed above, the proper signal is sent to *all* processes comprising the NQS *request* that are in the same *process group*. Whenever an NQS request is spawned, a new *process group* is established for all processes in the request. However, should one or more processes of the request successfully execute a *setpgrp*() system call, then such processes will **not** receive any signals sent by the *qdel*(1) command. This can lead to "rogue" request processes, which must be killed by other means--such as the *kill*(1) command. For the UNIX implementations that support the ability to "lock" a process and all of its progeny into a *process-group*, NQS will exploit this capability to prevent processes from "escaping" in this manner.

SEE ALSO

qdev(1), qlimit(1), qpr(1), qstat(1), qsub(1), kill(2), setpgrp(2), signal(2), qmgr(1M)

NAME

qlimit – show supported batch limits and shell strategy for the named host(s).

SYNOPSIS

qlimit [*host-name* ...]

DESCRIPTION

qlimit displays the set of batch request resource limit types that *can* be directly enforced on the implied local host or named hosts, and also the *batch request shell strategy* defined for the implied local host or named hosts.

If no *host-names* are given, then the information displayed is only relevant to the local host. Otherwise, the supported batch request limits and *batch request shell strategy* for each of the named hosts is displayed.

NQS supports many batch request resource limit types that can be applied to an NQS batch request. However, not all UNIX implementations are capable of supporting the rather extensive set of limit types that NQS provides.

The set of limits applied to a batch request is always restricted to the set of limits that can be directly supported by the underlying UNIX implementation. If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type.

When an attempt is made to queue a batch request, each *limit-value* specified by the request (that can also be supported by the local UNIX implementation), is compared against the corresponding *limit-value* as configured for the destination batch queue. If the corresponding batch queue *limit-value* for all batch request *limit-values is defined as unlimited*, or is *greater than* or *equal to* the corresponding batch request *limit-value*, then the request can be successfully queued, provided that no other anomalous conditions occur. For request *infinity limit-values*, the corresponding queue *limit-value* must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the *qsub* (1) command, or by the indirect placement of a batch request into a batch queue via a *pipe* queue. It is impossible for a batch request to be queued in an NQS batch queue if *any* of these resource limit checks fail.

Finally, if a request fails to specify a *limit-value* for a resource limit type that is supported on the execution machine, then the corresponding *limit-value* as configured for the destination queue, becomes the *limit-value* for the unspecified request limit.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent *qmgr* (1M) commands that alter the limits of the containing batch queue.

As mentioned above, this command also displays the *shell strategy* as configured for the implied local host, or named hosts. In the absence of a *shell specification* for a batch request, NQS must choose which shell should be used to execute that batch request. NQS supports three different algorithms, or *strategies* to solve this problem that can be configured for each system by a system administrator, depending on the needs of the user's involved, and upon system performance criterion.

The three possible shell strategies are called:

fixed,
free, and
login.

These shell strategies respectively cause the configured *fixed* shell to be exec'd to interpret all batch requests, cause the user's login shell as defined in the password file to be exec'd, which in turn chooses and spawns the appropriate shell for running the batch shell script, or cause only the user's login shell to be exec'd to interpret the script.

A shell strategy of *fixed* means that the same shell as chosen by the system administrator will be used to execute all batch requests.

A shell strategy of *free* will run the batch request script *exactly* as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of *fixed* and *login* exist for host systems that are short on available free processes. In these two strategies, a single shell is exec'd, and that same shell is the shell that executes all of the commands in the batch request shell script.

When a shell strategy of *fixed* has been configured for a particular NQS system, then the "fixed" shell that will be used to run *all* batch requests at that host is displayed.

SEE ALSO

qdel(1), qdev(1), qpr(1), qstat(1), qsub(1), qmgr(1M) n.TH QSTAT 1 July 1992 "Thinking Machines Corporation"

NAME

qstat – display status of NQS queue(s)

SYNOPSIS

qstat [**–a**] [**–l**] [**–m**] [**–u** *user-name*] [**–x**]
[*queue-name* ...] [*queue-name@host-name* ...]

DESCRIPTION

qstat displays the status of Network Queueing System (NQS) queues.

If no queues are specified, then the current state of each NQS queue on the local host is displayed. Otherwise, information is displayed for the specified queues only. Queues may be specified either as *queue-name* or *queue-name@host-name*. In the absence of a *host-name* specifier, the local host is assumed.

For each selected queue, *qstat* displays a *queue header* (information about the queue itself) followed by information about requests in the queue. Ordinarily, *qstat* shows only those requests belonging to the invoker. The following flags are available:

-a Shows all requests.

-l Requests are shown in a long format.

-m Requests are shown in a medium-length format.

-u *user-name*
Shows only those requests belonging to *user-name*.

-x The queue header is shown in an extended format.

The *queue header* always includes the queue-name, queue type, queue status (see below), the number of requests in the queue, and the CM resource (which sections of which CM the queue is associated with). An extended queue header goes on to display the priority and run limit of a queue, as well as the access restrictions, cumulative use statistics, wait time (when queue's window ends, number of seconds between the SIGTERM and SIGKILL signals), and resource limits (if a batch queue).

By default, *qstat* displays the following information about a request: the *request-name*, the *request-id*, the owner, the relative request priority, and the current request state (see below). For running requests, the process group is also shown, as soon as this information becomes available to the local NQS daemon.

qstat -m shows the following additional information: If the request was submitted with the constraint that it not run before a certain time and date, then the constraining time and date will also be displayed.

qstat -l shows the time at which the request was created, an indication of whether or not mail will be sent, where mail will be sent, and the username on the originating machine. If a batch queue is being examined, resource limits, planned disposition of stderr and stdout, any advice concerning the command interpreter, and the umask value are shown.

It must be understood that the relative ordering of requests within a queue does not always determine the order in which the requests will be run. The NQS request scheduler is allowed to make exceptions to the request ordering for the sake of efficient machine resource usage. However, requests appearing near the beginning of the queue have higher priority than requests appearing later, and will usually be run before requests appearing later on in the queue.

QUEUE STATE

The general state of a queue is defined by two principal properties of the queue.

The first property determines whether or not requests can be submitted to the queue. If they can, then the queue is said to be *enabled*. Otherwise the queue is said to be *disabled*. One of the words **CLOSED, ENABLED,** or **DISABLED** will appear in the queue status field to indicate the respective queue states of: enabled (with no local NQS daemon), enabled (and local NQS daemon is present), and disabled. Requests can only be submitted to the queue if the queue is enabled, and the local NQS daemon is present.

The second principal property of a queue determines if requests that are ready to run, but are not now presently running, will be allowed to run upon the completion of any currently running requests, and whether any requests are presently running in the queue.

If queued requests not already running are blocked from running, and no requests are presently executing in the queue, then the queue is said to be *stopped*. If the same situation exists with the difference that at least one request is running, then the queue is said to be *stopping*, where the requests presently executing will be allowed to complete execution, but no new requests will be spawned.

If queued requests ready to run are only prevented from doing so by the NQS request scheduler, and one or more requests are presently running in the queue, then the queue is said to be *running*. If the same circumstances prevail with the exception that no requests are presently running in the queue, then the queue is said to be *inactive*. Finally, if the NQS daemon for the local host upon which the queue resides is not running, but the queue would otherwise be in the state of *running* or *inactive*, then the queue is said to be *shutdown*. The queue states describing the second principal property of a queue are therefore respectively displayed as **STOPPED, STOPPING, RUNNING, INACTIVE,** and **SHUTDOWN.**

REQUEST STATE

The state of a request may be *arriving*, *holding*, *waiting*, *queued*, *staging*, *routing*, *running*, *departing*, or *exiting*. A request is said to be *arriving* if it is being enqueued from a remote host. *Holding* indicates that the request is presently prevented from entering any other state (including the *running* state), because a *hold* has been placed on the request. A request is said to be *waiting* if it was submitted with the constraint that it not run before a certain date and time, and that date and time have not yet arrived. *Queued* requests are eligible to proceed (by *routing* or *running*). When a request reaches the head of a pipe queue and receives service there, it is *routing*. A request is *departing* from the time the pipe queue turns to other work until the request has arrived intact at its destination. *Staging* denotes a *batch* request that has not yet begun execution, but for which input files are being brought on to the execution machine. A *running* request has reached its final destination queue, and is actually executing. Finally, *exiting* describes a batch request that has completed execution, and will exit from the system after the required output files have been returned (to possibly remote machines).

Imagine a batch request originating on a workstation, destined for the batch queue of a computation engine, to be run immediately. That request would first go through the states *queued*, *routing*, and *departing* in a local pipe queue. Then it would disappear from the pipe queue. From the point of view of a queue on the computation engine, the request would first be *arriving*, then *queued*, *staging* (if required by the batch request), *running*, and finally *exiting*. Upon completion of the *exiting* phase of execution, the batch request would disappear from the batch queue.

IDENTIFICATION

Connection Machine System Software Release 6.0. Copyright © 1990 by Thinking Machines Corporation, Cambridge MA.

RESTRICTIONS

NQS is not finished, and continues to undergo development. Some of the request states shown above may not be supported in your version of NQS.

SEE ALSO

qdel(1), qdev(1), qlimit(1), qpr(1), qsub(1), qmgr(1M)
Thinking Machines Corporation, *The Connection Machine System User's Guide*.

NAME

qsub – submit an NQS batch request.

SYNOPSIS

qsub [*flags*] [*script-file*]

DESCRIPTION

qsub submits a batch request to the Network Queueing System (NQS).

If no *script-file* is specified, then the set of commands to be executed as a batch request is taken directly from the standard input file (*stdin*). In all cases however, the *script file* is spooled, so that later changes will **not** affect previously queued batch requests.

All of the flags that can be specified on the command line can also be specified within the first comment block inside the batch request *script file* as ***embedded default flags***. Such flags appearing in the batch request *script file* set default characteristics for the batch request. If the same flag is specified on the command line, then the command line flag (and any associated value) takes precedence over the *embedded* flag. See the section entitled **LONG DESCRIPTION** for more information on *embedded default flags*.

What follows is a terse definition of the flags implemented by the *qsub* command (see the section **LONG DESCRIPTION** for the complete definition and syntax used for each of these flags).

–a – run request after stated time
–e – direct stderr output to stated destination
–eo – direct stderr output to the stdout destination
–I – specify process's account ID
–ke – keep stderr output on the execution machine
–ko – keep stdout output on the execution machine
–lc – establish per-process corefile size limit
–ld – establish per-process data-segment size limits
–lf – establish per-process permanent-file size limits
–lF – establish per-request permanent-file space limits
–lm – establish per-process memory size limits
–lM – establish per-request memory space limits
–ln – establish per-process nice execution value limit
–ls – establish per-process stack-segment size limits
–lt – establish per-process CPU time limits
–lT – establish per-request CPU time limits
–lv – establish per-process temporary-file size limits
–lV – establish per-request temporary-file space limits
–lw – establish per-process working set limit
–mb – send mail when the request begins execution
–me – send mail when the request ends execution
–mu – send mail for the request to the stated user
–nr – declare that batch request is not restartable
–o – direct stdout output to the stated destination
–p – specify intra-queue request priority
–q – queue request in the stated queue
–r – assign stated request name to the request
–re – remotely access the stderr output file
–ro – remotely access the stdout output file
–s – specify shell to interpret the batch request script
–x – export all environment variables with request
–z – submit the request silently

LONG DESCRIPTION

As described above, it is possible to specify *default* flags within the batch request *script file* that configure the default behavior of the batch request. The algorithm used to scan for such *embedded default flags* within an NQS batch request script file is as follows:

1. Read the first line of the *script file*.
2. If the current line contains only whitespace characters, or the first non-whitespace character of the line is ":", then goto step 7.
3. If the first non-whitespace character of the current line is not a "#" character, then goto step 8.
4. If the second non-whitespace character in the current line is *not* the "@" character, or the character immediately following the second non-whitespace character in the current line is *not* a "$", then goto step 7.
5. If no "-" is present as the character *immediately* following the "@$" sequence, then goto step 8.
6. Process the *embedded* flag, stopping the parsing process upon reaching the end of the line, or upon reaching the first unquoted "#" character.
7. Read the next *script file* line. Goto step 2.
8. End. No more *embedded* flags will be recognized.

Here is an example of the use of *embedded* flags within the *script file*.

```
#
# Batch request script example:
#
# @$-a "11:30pm EDT" -lt "21:10, 20:00"
#           # Run request after 11:30 EDT by default,
#           # and set a maximum per-process CPU time
#           # limit of 21 minutes and ten seconds.
#           # Send a warning signal when any process
#           # of the running batch request consumes
#           # more than 20 minutes of CPU time.
# @$-lT 1:45:00
#           # Set a maximum per-request CPU time limit
#           # of one hour, and 45 minutes. (The
#           # implementation of CPU time limits is
#           # completely dependent upon the UNIX
#           # implementation at the execution
#           # machine.)
# @$-mb -me # Send mail at beginning and end of
#           # request execution.
# @$-q batch1 # Queue request to queue: batch1 by
#           # default.
# @$          # No more embedded flags.
#
make all
```

The following paragraphs give the detailed descriptions of the *flags* supported by the *Qsub* command.

–a *date-time* Do not run the batch request before the specified date and/or time. If a *date-time* specification is comprised of two or more tokens separated by whitespace characters, then the *date-time* specification must be placed within double quotes as in: **–a** "*July, 4, 2026 12:31-EDT*", or otherwise escaped such that *qsub* and the shell will interpret the entire *date-time* specification as a single character string. This restriction also applies when an embedded default **–a** flag with accompanying *date-time* specification appears within the batch request *script file*.

The syntax accepted for the *date-time* parameter is relatively flexible. Unspecified date and time values default to an appropriate value (e.g., if no date is specified, then the current month, day, and year are assumed).

A date may be specified as a month and day (current year assumed), or the year can also be explicitly specified. It is also possible to specify the date as a weekday name (e.g., "Tues"), or as one of the strings: "today", or "tomorrow". Weekday names and month names can be abbreviated by any three character (or longer) prefix of the actual name. An optional period can follow an abbreviated month or day name.

Time of day specifications can be given using a twenty-four hour clock, or "am" and "pm" specifications may be used alternatively. In the absence of a meridian specification, a twenty-four hour clock is assumed.

It should be noted that the time of day specification is interpreted using the precise meridian definitions, whereby "12am" refers to the twenty-four hour clock time of 0:00:00, "12m" refers to noon, and "12-pm" refers to 24:00:00. Alternatively, the phrases "midnight" and "noon" are accepted as time of day specifications, where "midnight" refers to the time of 24:00:00.

qsub does not allow a timezone specification. Specifying a timezone, e.g. EST, results in a syntax error. The local timezone is always used.

All alphabetic comparisons are performed in a case insensitive fashion such that both "WeD" and "weD" refer to the day of Wednesday.

Some valid *date-time* examples are:

```
01-Jan-1986 12am, PDT
Tuesday, 23:00:00
11pm tues.
tomorrow 23-MST
```

–e *[machine:][[/]path/] stderr-filename*

Direct output generated by the batch request that is sent to the *stderr* file to the named *[machine:][[/]path/] stderr-filename*.

The brackets "[" and "]" enclose optional portions of the *stderr* destination *machine*, *path*, and *stderr-filename*.

If no explicit *machine* destination is specified, then the destination machine defaults to the machine that originated the batch request, or to the machine that will eventually run the request, depending on the respective absence, or presence of the **–ke** flag.

If no *machine* destination is specified, and the path/filename does not begin with a "/", then the current working directory is prepended to create a fully qualified path name, provided that the **–ke** (keep stderr) flag is also absent. In all other cases, any partial path/filename is interpreted relative to the user's home directory on the *stderr* destination machine.

This flag cannot be specified when the **–eo** flag option is also present.

If the **–eo** and **–e** *[machine:][[/]path/] stderr-filename* flag options are not present, then all *stderr* output for the batch request is sent to the file whose name consists of the first seven characters of the *request-name* followed by the characters: ".e", followed by the request sequence number portion of the *request-id* discussed below. In the absence of the **–ke** flag, this default *stderr* output file will be placed on the machine that originated the batch request in the current working directory, as defined when the batch request was first submitted. Otherwise, the file will be placed in the user's home directory on the execution machine.

–eo Direct all output that would normally be sent to the *stderr* file to the *stdout* file for the batch request. This flag cannot be specified when the **–e** *[machine:][[/]path/] stderr-filename* flag option is also present.

–l *account-ID*

Account the process under " *account-ID* " , a string of up to 8 characters. Note that **qsub** does not recognize the **CM_ACCOUNT_ID** environment variable.

–ke In the absence of an explicit *machine* destination for the *stderr* file produced by a batch request, the *machine* destination chosen for the *stderr* output file is the machine that originated the batch request. In some cases however, this behavior may be undesirable, and so the **–ke** flag can be specified which instructs NQS to leave any *stderr* output file produced by the request on the machine that actually *executed* the batch request.

This flag is meaningless if the **–eo** flag is specified, and cannot be specified if an explicit *machine* destination is given for the *stderr* parameter of the **–e** flag.

–ko In the absence of an explicit *machine* destination for the *stdout* file produced by a batch request, the *machine* destination chosen for the *stdout* output file is the machine that originated the batch request. In some cases however, this behavior may be undesirable, and so the **–ko** flag can be specified; this instructs NQS to leave any *stdout* output file produced by the request on the machine that actually *executed* the batch request.

This flag cannot be specified if an explicit *machine* destination is given for the *stdout* parameter of the **–o** flag.

–lc *per-process corefile size limit*

Set a *per-process* maximum *core file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to exit creating a core file whose size would exceed the maximum *per-process core file size limit* for the request, then the core file image of the aborting process will be reduced to the necessary size by an algorithm dependent upon the underlying UNIX implementation.

Not all UNIX implementations support *per-process corefile size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process corefile size limit*.

–ld *per-process data-segment size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *data-segment size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process data-segment size-limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process data-segment warning size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–ld** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process data-segment size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process data-segment size limit*.

–lf *per-process permanent-file size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *permanent-file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a permanent file such that the file size would increase beyond the maximum *per-process permanent-file size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning permanent-file size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lf** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process permanent-file size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

At the time of this writing, the author was unaware of any UNIX implementation that made a distinction at the **kernel** level between *permanent* and *temporary* files. While it is certainly possible to construct a *pseudo-temporary* file by first creating it, and then unlinking its pathname, the disk space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from. Furthermore, such a file will be subject to the same quota enforcement mechanisms, if any are provided by the underlying UNIX implementation, that all other UNIX files are created under.

For all UNIX implementations that do not support a distinction between *permanent* and *temporary* files at the **kernel** level, this limit is interpreted as a *per-process file size limit*, with the word *permanent* removed from the definition.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process permanent-file size limit*.

–lF *per-request permanent-file space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *permanent-file space limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a permanent file such that the file space consumed by all permanent files opened for writing by all of the processes in the batch request, would increase beyond the maximum *per-request permanent-file space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning permanent-file space limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lF** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-request permanent-file space limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

At the time of this writing, the author was unaware of any UNIX implementation that made a distinction at the **kernel** level, between *permanent* and *temporary* files. While it is certainly possible to construct a *pseudo-temporary* file by first creating it, and then unlinking its pathname, the disk space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from. Furthermore, such a file will be subject to the same quota enforcement mechanisms, if any are provided by the underlying UNIX implementation, that all other UNIX files are created under.

For all UNIX implementations that do not support a distinction between *permanent* and *temporary* files at the **kernel** level, this limit is interpreted as a *per-request file space limit*, with the word *permanent* removed from the definition.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request permanent-file space limit*.

–lm *per-process memory size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *memory size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process memory size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning memory size limits*. When a warning limit is exceeded, a signal (as determined by the underlying UNIX implementation) is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lm** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process memory size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process memory size limit*.

–lM *per-request memory space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *memory space limit* for all processes that constitute the running batch request. If the sum of all memory consumed by all of the processes comprising the running request exceeds the maximum *per-request memory space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning memory size limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lM** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-request memory space limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request memory space limit*.

–ln *per-process nice value limit*

Set a *per-process nice value* for all processes comprising the running batch request.

At present, all UNIX implementations support the use of an integer called the *nice* value, which determines the *execution-time* priority of a process relative to all other processes in the system. By letting the user set a limit on the *nice* value for all processes comprising the running request, a user can cause a request to consume less (or more) of the CPU resource presented by the execution machine.

This is particularly useful when a user wishes to execute a CPU intensive batch request on a machine running interactive processes. By setting a low *execution-time priority*, a user can make a long running batch request give way to more interactive processes during the daytime, while the coming of the nighttime hours with typically smaller process loads will allow such a request to gain more and more of the CPU resource. In this way, long running batch requests can be polite to their more transient, interactive neighbor processes.

The only quirk associated with this flag results from the peculiar choice of *nice* values, implemented by the standard UNIX implementations. In general, increasingly *negative* nice values cause the relative execution priority of a process to *increase*, while increasingly *positive* nice values causes the relative priority to *decrease*! Thus, a *nice value* limit specification of: "-ln -10" is greedier than a *nice value* limit specification of: "-ln 0".

Since varying UNIX implementations often support a different finite range of *nice values*, NQS allows the specification of *nice values* that can eventually turn out to be outside the limits for the UNIX implementation running at the *execution* machine. In such cases, NQS will simply bind the specified *nice value* limit to within the necessary range as appropriate.

Lastly, any *nice value* specified by the use of this flag must be acceptable to the batch queue in which the request is ultimately placed (see the section entitled **LIMITS** for more information).

–ls *per-process stack-segment size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *stack-segment size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process stack-segment size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning stack-segment size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–ls** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process stack-segment size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process stack-segment size limit*.

–lt *per-process CPU time limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *CPU time limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process CPU time limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process CPU warning time limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lt** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process CPU time limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process CPU time limit*.

–lT *per-request CPU time limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *CPU time limit* for all of the processes that constitute the running batch request. If the sum of the CPU times consumed by all of the processes in the batch request exceeds the maximum *per-request CPU time limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request CPU warning time limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lT** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-request CPU time limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request CPU time limit*.

–lv *per-process temporary file size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *temporary (volatile) file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a *temporary* file such that the file size would increase beyond the maximum *per-process temporary-file size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning temporary-file size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lv** flag with its associated limit value(s) appears within the batch request *script file*.

At the time of this writing, no UNIX operating system known to the author supported a distinction at the **kernel** level between *permanent* and *temporary files*. Certainly, a *pseudo-temporary* file can be constructed by creating it, and then unlinking its pathname. However, the file space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from.

Until a mechanism is implemented in the **kernel** that knows about *permanent* and *temporary* files, this limit cannot be supported in the sense most useful for batch requests, namely the strict enforcement of disk quotas for *permanent* versus *temporary* files.

Until such a time, this limit will simply be ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process temporary-file size limit*.

–lV *per-request temporary file space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *temporary (volatile) file space limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a *temporary* file such that the file space consumed by all *temporary* files opened for writing by all of the processes in the batch request would increase beyond the maximum *per-request temporary-file space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning temporary-file space limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is composed of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **–lV** flag with its associated limit value(s) appears within the batch request *script file*.

At the time of this writing, no UNIX operating system known to the author supported a distinction at the **kernel** level between *permanent* and *temporary files*. Certainly, a *pseudo-temporary* file can be constructed by creating it, and then unlinking its pathname. However, the file space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from.

Until a mechanism is implemented in the **kernel** that knows about *permanent* and *temporary* files, this limit cannot be supported in the sense most useful for batch requests, namely the strict enforcement of disk quotas for *permanent* versus *temporary* files.

Until such a time, this limit will simply be ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *temporary-file space limit*.

–lw *per-process working set size limit*

Set a *per-process* maximum *working set size limit* for all processes that constitute the running batch request.

Not all UNIX implementations support *per-process working set size limits*, and such a limit only makes sense in the context of a paged virtual memory machine. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process working set size limit*.

–mb Send mail to the user on the originating machine when the request begins execution. If the **–mu** flag is also present, then mail is sent to the user specified for the **–mu** flag instead of to the invoking user.

–me Send mail to the user on the originating machine when the request has ended execution. If the **–mu** flag is also present, then mail is sent to the user specified for the **–mu** flag instead of to the invoking user.

–mu *user-name*

Specify that any mail concerning the request should be delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no '@' characters), or as *user@machine*. In the absence of this flag, any mail concerning the request will be sent to the invoker on the originating machine.

–nr Declare that the request is non-restartable. If this flag is specified, then the request will not be restarted by NQS upon system boot if the request was running at the time of an NQS shutdown or system crash.

By default, NQS assumes that all requests are restartable, with the caveat that it is the responsibility of the user to ensure that the request will execute correctly if restarted, by the use of appropriate programming techniques.

Requests that are not running are always preserved across host crashes and NQS shutdowns for later requeueing, with or without this flag.

When NQS is shutdown via an operator command to the *qmgr* (1M) NQS control program, a **SIGTERM** signal is sent to all processes comprising all running NQS requests on the local host, and all queued NQS requests are barred from beginning execution. After a finite number of seconds have elapsed (typically sixty, but this value can be overridden by the operator), all remaining processes comprising all remaining running NQS requests are killed by the signal: **SIGKILL**.

For an NQS request to be properly restarted after an NQS shutdown, the **–nr** flag must not be specified, and the spawned batch request shell must ignore **SIGTERM** signals (which is done by default). The spawned batch request shell must also not exit before the final **SIGKILL** arrives. Since the batch request shell is simply spawning commands and programs, waiting for their completion, this implies that the commands and programs being executed by the batch request shell must also be immune to **SIGTERM** signals, saving state as appropriate before being killed by the final **SIGKILL** signal.

See the **CAVEATS** section below for more discussion concerning the restartability of NQS batch requests.

–o *[machine:][[/]path/] stdout-filename*

Direct output generated by the batch request that is sent to the *stdout* file to the named *[machine:][[/]path/] stdout-filename*.

The brackets "[" and "]" enclose optional portions of the *stdout* destination *machine*, *path*, and *stdout-filename*.

If no explicit *machine* destination is specified, then the destination machine defaults to the machine that originated the batch request, or to the machine that will eventually run the request, depending on the respective absence, or presence of the **–ko** flag.

If no *machine* destination is specified, and the path/filename does not begin with a "/", then the current working directory is prepended to create a fully qualified path name, provided that the **–ko** (keep stdout) flag is also absent. In all other cases, any partial path/filename is interpreted relative to the user's home directory on the *stdout* destination machine.

If no **–o** *[machine:][[/]path/] stdout-filename* flag is specified, then all *stdout* output for the batch request is sent to the file whose name consists of the first seven characters of the *request-name* followed by the characters: ".o", followed by the request sequence number portion of the *request-id* discussed below. In the absence of the **–ko** flag, this default *stdout* output file will be placed on the machine that originated the batch request in the current working directory, as defined when the batch request was first submitted. Otherwise, the file will be placed in the user's home directory on the execution machine.

–p *priority*

Explicitly assign an *intraqueue* priority to the request. The specified *priority* must be an integer, and must be in the range [0..63], inclusive. A value of 63 defines the highest *intraqueue* request priority, while a value of 0 defines the lowest. This priority does **not** determine the execution priority of the request. This priority is only used to determine the relative ordering of requests within a queue.

When a request is added to a queue, it is placed at a specific position within the queue such that it appears ahead of all existing requests whose priority is less than the priority of the new request. Similarly, all requests with a higher priority will remain ahead of the new request when the queueing process is complete. When the priority of the new request is equal to the priority of an existing request, the existing request takes precedence over the new request.

If no *intraqueue* priority is chosen by the user, then NQS assigns a default value.

–q *queue-name*
: Specify the queue to which the batch request is to be submitted. If no **–q** *queue-name* specification is given, then the user's environment variable set is searched for the variable: **QSUB_QUEUE**. If this environment variable is found, then the character string value for **QSUB_QUEUE** is presumed to name the queue to which the request should be submitted. If the **QSUB_QUEUE** environment variable is not found, then the request will be submitted to the default batch request queue, *if* defined by the local system administrator. Otherwise, the request cannot be queued, and an appropriate error message is displayed to this effect.

–r *request-name*
: Assign the specified *request-name* to the request. In the absence of an explict **–r** *request-name* specification, the *request-name* defaults to the name of the *script file* (leading path name removed) given on the command line. If no *script file* was given, then the default *request-name* assigned to the request is **STDIN**.

In all cases, if the *request-name* is found to begin with a digit, then the character 'R' is prepended to prevent a *request-name* from beginning with a digit. All *request-names* are truncated to a maximum length of 15 characters.

–re
: By default, all output generated by a batch request sent to the *stderr* file is temporarily into a file residing in a protected directory on the machine that executes the request. When the batch request completes execution, this file is then spooled to its final destination, possibly on a remote machine.

This default spooling of the *stderr* output file is done to reduce the network traffic costs incurred by the submitter (owner) of a batch request that is to return its *stderr* output to a remote machine upon completion. In some cases, this behavior is not desired. If it is necessary to override this behavior, then the **–re** flag can be specified which says that *stderr* output produced by the request is to be *immediately* written to the final destination file, as output is generated, no matter what the networking cost.

Circumstances may not allow a given NQS implementation to support this flag, in which case it will be ignored, and the *stderr* output file will simply be spooled as it ordinarily would without this flag.

–ro
: By default, all output generated by a batch request sent to the *stdout* file is temporarily spooled into a file residing in a protected directory on the machine that executes the request. When the batch request completes execution, this file is then spooled to its final destination, possibly on a remote machine.

This default spooling of the *stdout* output file is done to reduce the network traffic costs incurred by the submitter (owner) of a batch request which is to return its *stdout* output to a remote machine upon completion. In some cases, this behavior is not desired. If it is necessary to override this behavior, then the **–ro** flag can be specified; this flag says that *stdout* output produced by the request is to be *immediately* written to the final destination file, as output is generated, no matter what the networking cost.

Circumstances may not allow a given NQS implementation to support this flag, in which case it will be ignored, and the *stdout* output file will simply be spooled as it ordinarily would without this flag.

–s *shell-name*
: Specify the absolute pathname of the shell that will be used to interpret the batch request script. This flag unconditionally overrides any *shell strategy* configured on the execution machine for selecting which shell to spawn in order to interpret the batch request script.

In the absence of this flag, the NQS system at the execution machine will use one of three (3) distinct *shell choice strategies* for the execution of the batch request. Any one of the three strategies can be configured by a system administrator for each NQS machine.

The three shell strategies are called:

fixed,
free, and
login.

These shell strategies respectively cause the configured *fixed* shell to be exec'd to interpret all batch requests, cause the user's login shell as defined in the password file to be exec'd which in turn chooses and spawns the appropriate shell for interpreting the batch request script, or cause only the user's login shell to be exec'd to interpret the script.

A shell strategy of *fixed* means that the same shell (as configured by the system administrator), will be used to execute **all** batch requests.

A shell strategy of *free* will run the batch request script *exactly* as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of *fixed* and *login* exist for host systems that are short on available free processes. In these two strategies, a single shell is exec'd, and that same shell is the shell that executes all of the commands in the batch request script.

The *shell strategy* configured for a particular NQS system can be determined by the *qlimit* (1) command.

–x Export all environment variables. When a batch request is submitted, the current values of the environment variables: **HOME, SHELL, PATH, LOGNAME** (not all systems), **USER** (not all systems), **MAIL**, and **TZ** are saved for later re-creation when the batch request is spawned, as the respective environment variables: **QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_USER, QSUB_MAIL**, and **QSUB_TZ**. Unless the **–x** flag is specified, no other environment variables will be exported from the originating host for the batch request. If the **–x** flag option is specified, then all remaining environment variables whose names do not conflict with the automatically exported variables, are also exported with the request. These additional environment variables will be recreated under the same name when the batch request is spawned.

–z Submit the batch request silently. If the request is submitted successfully, then no messages are displayed indicating this fact. Error messages will, however, always be displayed.

If the batch request is successfully submitted and the **–z** flag has not been specified, the *request-id* of the request is displayed to the user. A *request-id* is always of the form: *seqno.hostname* , where *seqno* refers to the sequence number assigned to the request by NQS, and *hostname* refers to the name of originating local machine. This identifier is used throughout NQS to uniquely identify the request, no matter where it is in the network.

The following events take place in the following order when an NQS *batch* request is spawned:

The process that will become the head of the *process group* for all processes comprising the batch request is created by NQS.

Resource limits are enforced.

The real and effective group-id of the process is set to the group-id as defined in the local password file for the request owner.

The real and effective user-id of the process is set to the real user-id of the batch request owner.

The user file creation mask is set to the value that the user had on the originating machine when the batch request was first submitted.

If the user explicitly specified a shell by use of the **–s** flag (discussed above), then that user-specified shell is chosen as the shell that will be used to execute the batch request script. Otherwise, a shell is chosen based upon the *shell strategy* as configured for the local NQS system (see the earlier discussion of the **–s** flag for a description of the possible *shell strategies* that can be configured for an NQS system).

The environment variables of **HOME, SHELL, PATH, LOGNAME** (not all systems), **USER** (not all systems), and **MAIL** are set from the user's password file entry, as though the user had logged directly into the execution machine.

The environment string: **ENVIRONMENT=BATCH** is added to the environment so that shell scripts (and the user's **.profile** (*Bourne shell*) or **.cshrc** and **.login** (*C-shell*) scripts), can test for batch request execution when appropriate, and not (for example) perform any setting of terminal characteristics, since a batch request is not connected to an input terminal.

The environment variables of **QSUB_WORKDIR, QSUB_HOST, QSUB_REQNAME,** and **QSUB_REQID** are added to the environment. These environment variables equate to the obvious respective strings of the working directory at the time that the request was submitted, the name of the originating host, the name of the request, and the request *request-id*.

All of the remaining environment variables saved for re-creation when the batch request is spawned are added at this point to the environment. When a batch request is initially submitted, the current values of the environment variables: **HOME, SHELL, PATH, LOGNAME** (not all systems), **USER** (not all systems), **MAIL,** and **TZ** are saved for later recreation when the batch request is spawned. When recreated however, these variables are added to the environment under the respective names: **QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_USER, QSUB_MAIL,** and **QSUB_TZ,** to avoid the obvious conflict with the local version of these environment variables. Additionally, all environment variables exported from the originating host by the **–x** option are added to the environment at this time.

The current working directory is then set to the user's home directory on the execution machine, and the chosen shell is exec'd to execute the batch request script with the environment as constructed in the steps outlined above.

In all cases, the chosen shell is exec'd as though it were the *login* shell. If the *Bourne* shell is chosen to execute the script, then the **.profile** file is read. If the *C-shell* is chosen, then the **.cshrc** and **.login** scripts are read.

If the user did not specify a specific shell for the batch request, then NQS chooses which shell should be used to execute the shell script, based on the *shell strategy* as configured by the system administrator (see the earlier discussion of the **–s** flag).

In such a case, a *free* shell strategy instructs NQS to execute the login shell for the user (as configured in the password file). The login shell is in turn instructed to examine the shell script file, and fork another shell *of the appropriate type* to interpret the shell script, behaving *exactly* as an interactive invocation of the script.

Otherwise no additional shell is spawned, and the chosen *fixed* or *login* shell sequentially executes the commands contained in the shell script file until completion of the batch request.

QUEUE ACCESS

NQS supports queue access restrictions. If access is *unrestricted*, any request may enter the queue. If access is *restricted*, a request can only enter the queue if the requester or the requester's login group has been given access to that queue (see *qmgr*(1M)). Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Use *qstat*(1) to determine who has access to a particular queue.

LIMITS

NQS supports many batch request resource limit types that can be applied to an NQS batch request. The existence of configurable resource limits allows an NQS user to set resource limits within which his or her request must execute. In many instances, smaller limit values can result in a more favorable scheduling policy for a batch request.

The syntax used to specify a *limit-value* for one of the *limit-flags* (**–l***limit-letter-type*), is quite flexible, and describes values for two general limit categories. These two general categories respectively deal with time related limits, and those limits are not time related.

For *finite* CPU time limits, the *limit-value* is expressed in the reasonably obvious format:

[[hours :] minutes :] seconds [.milliseconds]

Whitespace can appear anywhere between the principal tokens, with the exception that no whitespace can appear around the decimal point.

Example time *limit-values are:*

1234 : 58 : 21.29	– 1234 hrs 58 mins 21.290 secs
12345	– 12345 seconds
121.1	– 121.100 seconds
59:01	– 59 minutes and 1 second

For all other *finite* limits (with the exclusion of the *nice limit-value* **–ln**), the acceptable syntax is:

.fraction [units]

or

integer [.fraction] [units]

where the *integer* and *fraction* tokens represent strings of up to eight decimal digits, denoting the obvious values. In both cases, the *units* of allocation may also be specified as one of the case insensitive strings:

b	– bytes
w	– words
kb	– kilobytes (2^10 bytes)
kw	– kilowords (2^10 words)
mb	– megabytes (2^20 bytes)
mw	– megawords (2^20 words)
gb	– gigabytes (2^30 bytes)
gw	– gigawords (2^30 words)

In the absence of any *units* specification, the units of *bytes* are assumed.

For all limit types with the exception of the *nice limit-value* (**–l***n*), it is possible to state that no limit should be applied. This is done by specifying a *limit-value* of "unlimited", or any initial substring thereof. Whenever an *infinite limit-value* is specified for a particular resource type, then the batch request operates as though no explicit limits have been placed upon the corresponding resource, other than by the limitations of the physical hardware involved.

The complications caused by *batch request* resource limits first show up when queueing a *batch request* in a *batch queue* . This operation is described in the following paragraphs.

If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type. (See the *qlimit*(1) command to find out what limits are supported by a given machine.)

For each remaining *finite* limit that can be supported by the underlying UNIX implementation that is **not** a CPU *time-limit* or UNIX *execution-time nice-value-limit*, the *limit-value* is internally converted to the units of *bytes* or *words*, whichever is more appropriate for the underlying machine architecture.

As an example, a *per-process memory size limit value* of 321 megabytes would be interpreted as 321 x 2^20 bytes, provided that the underlying machine architecture was capable of directly addressing single bytes. Thus the original limit *coefficient* of 321 would become 321 x 2^20. On a machine that was only capable of addressing words, the appropriate conversion of 321 x 2^20 *bytes / #of-bytes-per-word* would be performed.

If the result of such a conversion would cause overflow when the coefficient was represented as a *signed-long integer* on the supporting hardware, then the coefficient is replaced with the coefficient of: of 2^*N*-1 where *N* is equal to the number of bits of precision in a signed long integer. For typical 32-bit machines, this *default extreme limit* would therefore be 2^*31*-1 bytes. For word addressable machines in the supercomputer class supporting 64-bit long integers, the *default extreme limit* would be 2^*63*-1 words.

Lastly, some implementations of UNIX reserve coefficients of the form: 2^*N*-1 as synonymous with infinity, meaning no limit is to be applied. For such UNIX implementations, NQS further decrements the *default extreme limit* so as not to imply infinity.

The identical internal conversion process as described in the preceding paragraphs is also performed for each *finite limit-value* configured for a particular batch queue using the *qmgr*(1M) program.

After all of the applicable *limit-values* have been converted as described above, each such resulting *limit-value* is then compared against the corresponding *limit-value* as configured for the destination batch queue. If, for every type of limit, the batch queue *limit-value* is *greater than* or *equal to* the corresponding batch request *limit-value*, then the request can be successfully queued, provided that no other anomalous conditions occur. For request *infinity limit-values*, the corresponding queue *limit-value* must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the *qsub(1)* command, or by the indirect placement of a batch request into a batch queue via a *pipe* queue. It is impossible for a batch request to be queued in an NQS batch queue if *any* of these resource limit checks fail.

Finally, if a request fails to specify a *limit-value* for a resource limit type that is supported on the execution machine, then the corresponding *limit-value* configured for the destination queue becomes the *limit-value* for the unspecified request limit.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent *qmgr*(1M) commands that alter the limits of the containing batch queue.

CAVEATS

When an NQS batch request is spawned, a new *process-group* is established such that all processes of the request exist in the same *process-group*. If the *qdel*(1) command is used to send a signal to an NQS batch request, the signal is sent to all processes of the request in the created *process-group*. However, should one or more processes of the request choose to successfully execute a *setpgrp*(2) system call, then such processes will **not** receive any signals sent by the *qdel*(1) command. This can lead to "rogue" requests whose constituent processes must be killed by other means such as the *kill*(1) command. However, NQS takes advantage of any UNIX implementations that provide a mechanism of "locking" a process, and all of its subsequent children in a particular *process-group*. For such UNIX implementations, this problem does not occur.

It is extremely wise for all processes of an NQS request to catch any **SIGTERM** signals. By default, the receipt of a **SIGTERM** signal causes the receiving process to die. NQS sends a **SIGTERM** signal to all processes in the established *process-group* for a batch request as a notification that the request should be prepared to be killed, either because of an *abort queue* command issued by an operator using the *qmgr*(1M) program, or because it is necessary to shutdown NQS and all running requests as part of a general shutdown procedure of the local host.

It must be understood that the spawned *shell* ignores **SIGTERM** signals. If the current immediate child of the *shell* does not ignore or catch **SIGTERM** signals, then it will be killed by the receipt of such, and the shell will go on to execute the next command from the script (if there is one). In any case, the shell will not be killed by the **SIGTERM** signal, though the executing command will have been killed.

After receiving a **SIGTERM** signal delivered from NQS, a process of a batch request typically has sixty seconds to get its "house in order" before receiving a **SIGKILL** signal (though the sixty second duration can be changed by the operator).

All batch requests terminated because of an operator *NQS shutdown request* that did not specify the **–nr** flag are considered restartable by NQS, and are requeued (provided that the batch request shell process is still present at the time of the **SIGKILL** signal broadcast as discussed above), so that when NQS is rebooted, such batch requests will be respawned to continue execution. It is however, up to the user to make the request restartable by the appropriate programming techniques. NQS simply spawns the request again as though it were being spawned for the first time.

Upon completion of a batch request, a mail message can be sent to the submitter (see the discussion of the **–me** flag above). In many instances, the completion code of the spawned *Bourne* or *C-Shell* will be displayed. This is merely the value returned by the shell through the *exit*(2) system call.

Lastly, there is no good way to echo commands executed by unmodified versions of the *Bourne* and *C* shells. While the *C-shell* can be spawned in such a fashion as to echo the commands it executes, it is often very difficult to tell an echoed command from genuine output produced by the batch request, because no "magic" character such as a '$' is displayed in front of the echoed command. The *Bourne* shell does not support any echo option whatsoever.

Thus, one of the better ways to write the shell script for a batch request is to place appropriate lines in the shell script of the form:

```
echo "explanatory-message"
```

where the echoed message should be a meaningful message chosen by the user.

LIMITATIONS AND IMPLEMENTATION NOTES

In the present implementation, it is **not** possible to see the *stderr* or *stdout* files produced by the batch request while the request is **running**, unless the **-re** and **-ro** flags have been respectively specified.

Lastly, the strange "@$" syntax used to introduce *embedded argument* flags was chosen because it rarely conflicts with anything else present in a shell script file. NQS users with better minds will (rightly) suggest improved alternatives to this convention.

SEE ALSO

mail(1), qdel(1), qdev(1), qlimit(1), qpr(1), qstat(1), kill(2), setpgrp(2), signal(2), qmgr(1M).

CM User's Guide.
CM System Administrator's Guide.

NAME

restart – run a checkpointed version of a Connection Machine program

SYNOPSIS

restart *feprefix*

DESCRIPTION

restart restarts a program that has been checkpointed using the CM checkpointing utility.

When a program is checkpointed, the checkpointing utility saves the state of the program in files that have the prefix *feprefix* and, optionally, *cmprefix*; these prefixes are specified by the routine that executes the checkpoint. *feprefix* is used for the front-end core file, a list of the files that the program had open when it was checkpointed, and the stored copy of the checkpointed program. *cmprefix* is used as the prefix for the CM core file, and specifies a pathname in the CM file system. The CM core file is not created if the program does not use the CM.

restart takes as an argument the front-end prefix used for naming the checkpointed files. It obtains the CM prefix, if any, from the program it is restarting. The program begins execution from the point at which the *feprefix* and *cmprefix* files were saved.

Be careful when using output redirection with *restart*. You must *append* the output to the output file, not just redirect it; otherwise, *restart* overwrites what is already in the file.

It can take up to several minutes to restart a checkpointed program, depending on the size of the files. If a flag is set in the program, progress reports are displayed while the program is being restarted.

To restart a checkpoint more than once, rename the checkpoint files and issue *restart* with different prefixes as arguments. If you move the checkpoint files to another directory, make sure the files used by the program are accessible from this directory with the same names they had when opened by the original invocation of the program.

FILES

feprefix-**core**
: The standard core file, containing the state of the program on the front end.

feprefix-**file-list**
: List of files that the program had open when it was checkpointed.

feprefix-**program**
: The stored copy of the checkpointed program.

cmprefix-**cm-core**
: The state of the program on the CM. This file is not created if the program is not using the CM.

IDENTIFICATION

Connection Machine System Software Release 6.0.

SEE ALSO

Thinking Machines Corporation, *CM User's Guide*.

Index