QUADRICS

Compaq AlphaServer SC RMS Reference Manual

Quadrics Supercomputers World Ltd. **Document Version 7 - June 22nd 2001 - AA-RLAZB-TE**

The Quadrics Supercomputers World Ltd. (Quadrics) web site can be found at:

```
http://www.quadrics.com/
```

Quadrics' address is:

One Bridewell Street
Bristol
BS1 2AA
UK

Tel: +44-(0)117-9075375
Fax: +44-(0)117-9075395

Circulation Control: None

Document Revision History

Revision	Date	Author	Remarks
1	January 1999	HRA	Initial Draft
2	Feb 2000	DR	Updated Draft
3	Apr 2000	DR	Draft changes for Product Release
4	Jun 2000	RMC	Corrections for Product Release
5	Jan 2001	HRA	Updates for Version 2
6	June 2001	DR	Further Version 2 changes
7	June 2001	DR	AlphaServer SC V2 Product Release

Contents

List of Figures

List of Tables

1

Introduction

1.1 Scope of Manual

This manual describes the Resource Management System (RMS). The manual's purpose is to provide a technical overview of the RMS system, its functionality and programmable interfaces. It covers the RMS daemons, client applications, the RMS database, the system call interface to the RMS kernel module and the application program interface to the RMS database.

1.2 Audience

This manual is intended for system administrators and developers. It provides a detailed technical description of the operation and features of RMS and describes the programming interface between RMS and third-party systems.

The manual assumes that the reader is familiar with the following:

- UNIX® operating system including shell scripts
- C programming language

1.3 Using this Manual

This manual contains ten chapters and five appendices. The contents of these are as follows:

Related Information

Chapter 1 (*Introduction*)

explains the layout of the manual and the conventions used to present information

Chapter 2 (*Overview of RMS*)

overviews the functions of the RMS and introduces its components

Chapter 3 (*Parallel Programs Under RMS*)

shows how parallel programs are executed under RMS

Chapter 4 (*RMS Daemons*)

describes the functionality of the RMS daemons

Chapter 5 (*RMS Commands*)

describes the RMS commands

Chapter 6 (*Access Control, Usage Limits and Accounting*)

explains RMS access controls, usage limits and accounting

Chapter 7 (*RMS Scheduling*)

describes how RMS schedules parallel jobs

Chapter 8 (*Event Handling*)

describes RMS event handling

Chapter 9 (*Setting up RMS*)

explains how to set up RMS

Chapter 10 (*The RMS Database*)

presents the structure of tables in the RMS database

Appendix A (*Compaq AlphaServer SC Interconnect Terms*)

defines terms relating to support for QsNet in RMS

Appendix B (*RMS Status Values*)

lists the status values of RMS objects

Appendix C (*RMS Kernel Module*)

describes the RMS kernel module and its system call interface

Appendix D (*RMS Application Interface*)

describes the RMS application interface

Appendix E (*Accounting Summary Script*)

contains an example of producing accounting information

1.4 Related Information

The following manuals provide additional information about the RMS from the point of view of either the system administrator or the user:

- *Compaq AlphaServer SC User Guide*
- *Compaq AlphaServer SC System Administration Guide*

1.5 Location of Online Documentation

Online documentation in HTML format is installed in the directory `/usr/opt/rms/docs/html` and can be accessed from a browser at `http://rmshost:8081/html/index.html`. PostScript and PDF versions of the documents are in `/usr/opt/rms/docs`. Please consult your system administrator if you have difficulty accessing the documentation. On-line documentation can also be found on the AlphaServer SC System Software CD-ROM.

New versions of this and other Quadrics documentation can be found on the Quadrics web site `http://www.quadrics.com`.

Further information on AlphaServer SC can be found on the Compaq website `http://www.compaq.com/hpc`.

1.6 Reader's Comments

If you would like to make any comments on this or any other AlphaServer SC manual please contact your local Compaq support centre.

1.7 Conventions

The following typographical conventions have been used in this document:

`monospace type`

Monospace type denotes literal text. This is used for command descriptions, file names and examples of output.

`bold monospace type`

Bold monospace type indicates text that the user enters when contrasted with on-screen computer output.

italic monospace type	Italic (slanted) monospace type denotes some meta text. This is used most often in command or parameter descriptions to show where a textual value is to be substituted.
italic type	Italic (slanted) proportional type is used in the text to introduce new terms. It is also used when referring to labels on graphical elements such as buttons.
`Ctrl/x`	This symbol indicates that you hold down the `Ctrl` key while you press another key or mouse button (shown here by `x`).
TLA	Small capital letters indicate an abbreviation (see Glossary).
`ls(1)`	A cross-reference to a reference page includes the appropriate section number in parentheses.
#	A number sign represents the superuser prompt.
%, $	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.

2

Overview of RMS

2.1 Introduction

This chapter describes the role of the Resource Management System (RMS). The RMS provides tools for the management and use of a Compaq AlphaServer SC system. To put into context the functions that RMS performs, a brief overview of the system architecture is given first in Section 2.2. Section 2.3 outlines the main functions of the RMS and introduces the major components of the RMS: a set of UNIX daemons, a suite of command line utilities and a SQL database. Finally, Section 2.4 describes the resource management facilities from the system administrator's point of view.

2.2 The System Architecture

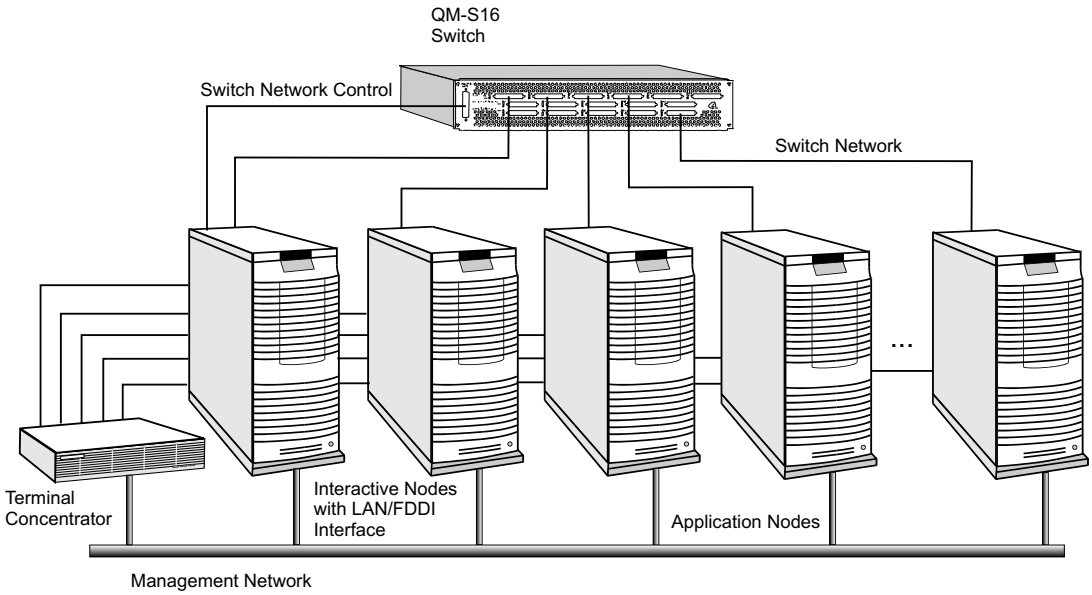
An RMS system looks like a standard UNIX system: it has the familiar command shells, editors, compilers, linkers and libraries; it runs the same applications. The RMS system differs from the conventional UNIX one in that it can run parallel applications as well as sequential ones. The processes that execute on the system, particularly the parallel programs, are controlled by the RMS.

2.2.1 Nodes

An RMS system comprises a network of computers (referred to as *nodes*) as shown in Figure 2.1. Each node may have single or multiple processors (such as a SMP server); each node runs a single copy of UNIX. Nodes used interactively to login to the RMS

system are also connected to an external LAN. The application nodes, used for running parallel programs, are accessed through the RMS.

Figure 2.1: A Network of Nodes

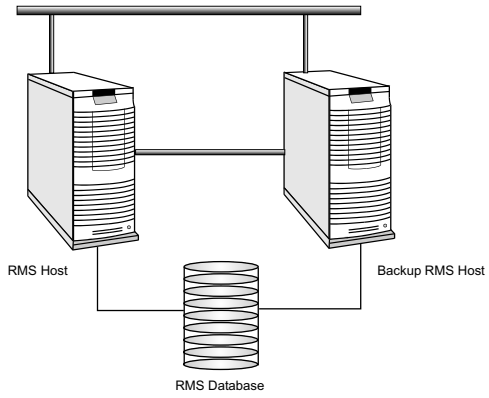


All of the nodes are connected to a *management network* (normally, a 100 BaseT Ethernet). They may also be connected to a Compaq AlphaServer SC Interconnect, to provide high-performance user-space communications between application processes.

The RMS processes that manage the system reside either on an interactive node or on a separate management server. This node, known as `rmshost`, holds the RMS database, which stores all state for the RMS system.

For high-availability installations, the `rmshost` node should be an interactive node rather than a management server. This will allow you to configure the system for failover, as shown in Figure 2.2 (see Chapter 15 of the *System Administration Guide* for details).

Figure 2.2: High Availability RMS Configuration



The RMS processes run on the node with the name `rmshost`, which migrates to the backup on fail-over. The database is held on a shared disk, accessible to both the primary and backup node.

2.3 The Role of the RMS

The RMS provides a single point interface to the system for resource management. This interface enables a system administrator to manage the system resources (CPUs, memory, disks, and so on) effectively and easily. The RMS includes facilities for the following administrative functions:

Monitoring controlling and monitoring the nodes in the network to ensure the correct operation of the hardware

Fault diagnosis diagnosing faults and isolating errors; instigating fault recovery and escalation procedures

Data collection recording statistics on system performance

Allocating CPUs allocating system resources to applications

Access control controlling user access to resources

Accounting single point for collecting accounting data

Parallel jobs providing the system support required to run parallel programs

Scheduling	deciding when and where to run parallel jobs
Audit	maintaining an audit trail of system state changes

From the user's point of view, RMS provides tools for:

Information	querying the resources of the system
Execution	loading and running parallel programs on a given set of resources
Monitoring	monitoring the execution of parallel programs

2.3.1 The Structure of the RMS

RMS is implemented as a set of UNIX commands and daemons, programmed in C and C++, using sockets for communications. All of the details of the system (its configuration, its current state, usage statistics) are maintained in a SQL database, as shown in Figure 2.3. See Section 2.3.4 for an overview and Chapter 10 (*The RMS Database*) for details of the database.

2.3.2 The RMS Daemons

A set of daemons provide the services required for managing the resources of the system. To do this, the daemons both query and update the database (see Section 2.3.4).

- The *Database Manager*, `msqld`, provides SQL database services.
- The *Machine Manager*, `mmanager`, monitors the status of nodes in an RMS system.
- The *Partition Manager*, `pmanager`, controls the allocation of resources to users and the scheduling of parallel programs.
- The *Switch Network Manager*, `swmgr`, supervises the operation of the Compaq AlphaServer SC Interconnect, monitoring it for errors and collecting performance data.
- The *Event Manager*, `eventmgr`, runs handlers in response to system incidents and notifies clients who have registered an interest in them.
- The *Transaction Log Manager*, `tlogmgr`, instigates database transactions that have been requested in the *Transaction Log*. All client transactions are made through this mechanism. This ensures that changes to the database are serialized and an audit trail is kept.
- The *Process Manager*, `rmsmhd`, runs on each node in the system. It starts the other RMS daemons.

- The *RMS Daemon*, `rmsd`, runs on each node in the system. It loads and runs user processes and monitors resource usage and system performance.

The RMS daemons are described in more detail in Chapter 4 (*RMS Daemons*).

2.3.3 The RMS Commands

RMS commands call on the RMS daemons to get information about the system, to distribute work across the system, to monitor the state of programs and, in the case of administrators, to configure the system and back it up. A suite of these RMS client applications is supplied. There are commands for users and commands for system administrators.

The user commands for gaining access to the system and running parallel programs are as follows:

- `allocate` reserves resources for a user.
- `prun` loads and runs parallel programs.
- `rinfo` gets information about the resources in the system.
- `rmsexec` performs load balancing for the efficient execution of sequential programs.
- `rmsquery` queries the database. Administrators can also use `rmsquery` to update the database.

The system administration commands for managing the system are as follows:

- `nodestatus` gets and sets node status information.
- `rcontrol` starts, stops and reconfigures services.
- `rmsbuild` populates the RMS database with information on a given system.
- `rmsctl` starts and stops RMS and shows the system status.
- `rmshost` reports the name of the node hosting the RMS database.
- `rmstbladm` builds and maintains the database.
- `msqladmin` performs database server administration.

The services available to the different types of user (application programmer, operator, system administrator) are subject to access control. Access control restrictions are embedded in the SQL database, based on standard UNIX group IDs (see

Section 10.2.20). Users have read access to all tables but no write access. Operator and administrative applications are granted limited write access. Password-protected administrative applications and RMS itself have full read/write access.

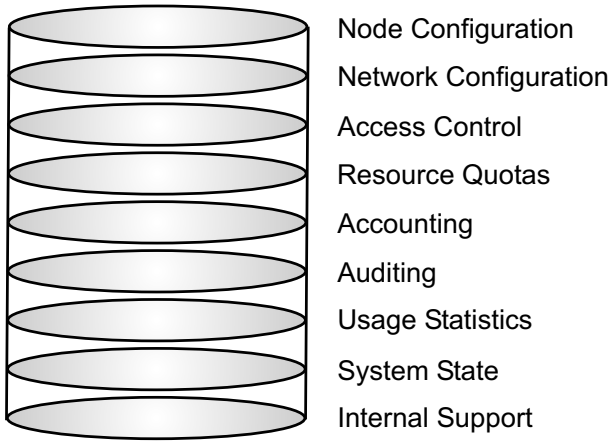
The RMS commands are described in more detail in Chapter 5 (*RMS Commands*).

2.3.4 The RMS Database

The database provides a platform-independent interface to the RMS system. Users and administrators can interact with the database using standard SQL queries. For example, the following query displays details about the nodes in the machine. It selects fields from the table called `nodes` (see Section 10.2.14). The query is submitted through the RMS client `rmsquery`.

```
$ rmsquery "select name,status from nodes"
atlasms  running
atlas0   running
atlas1   running
atlas2   running
atlas3   running
```

Figure 2.3: The Database



RMS uses the mSQL database engine from Hughes Technologies (for details see `http://www.Hughes.com.au`). Client applications may use C, C++, Java, HTML or UNIX script interfaces to generate SQL queries. See the Quadrics support page `http://www.quadrics.com/web/support` for details of the SQL language.

2.4 RMS Management Functions

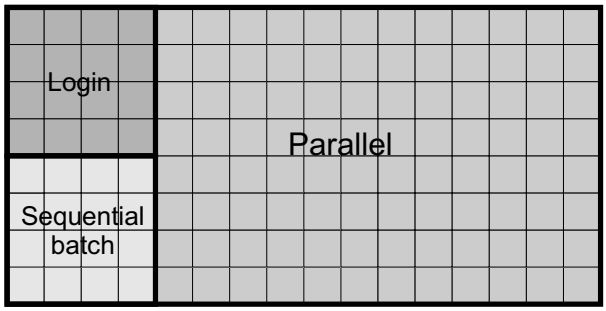
The RMS gives the system administrator control over how the resources of a system are assigned to the tasks it must perform. This includes the allocation of resources (Section 2.4.1), scheduling policies (Section 2.4.2), access controls and accounting (Section 2.4.3) and system configuration (Section 2.4.4).

2.4.1 Allocating Resources

The nodes in an RMS system can be configured into mutually exclusive sets known as *partitions* as shown in Figure 2.4. The administrator can create partitions with different mixes of resources to support a range of uses. For example, a system may have to cater for a variety of processing loads, including the following:

- Interactive login sessions for conventional UNIX processes
- Parallel program development
- Production execution of parallel programs
- Distributed system services, such as database or file system servers, used by conventional UNIX processes
- Sequential batch streams

Figure 2.4: Partitioning a System



The system administrator can allocate a partition with appropriate resources for each of these tasks. Furthermore, the administrator can control who accesses the partitions (by user or by project) and how much of the resource they can consume. This ensures that resources intended for a particular purpose, for example, running production parallel codes, are not diverted to other uses, for example, running user shells.

A further partition, the `root` partition, is always present. It includes all nodes. It does not have a scheduler. The `root` partition can only be used by administrative users (`root` and `rms` by default).

2.4.2 Scheduling

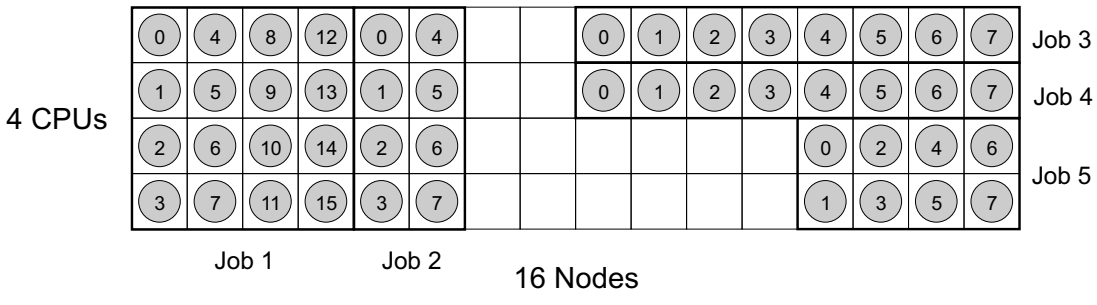
Partitions enable different scheduling policies to be put into action. On each partition, one or more of three scheduling policies can be deployed to suit the intended usage:

1. Gang scheduling of parallel programs, where all processes in a program are scheduled and de-scheduled together. This is the default scheduling policy for parallel partitions.
2. Regular UNIX scheduling with the addition of load balancing, whereby the user can run a sequential program on a lightly loaded node. The load may be judged in terms of free CPU time, free memory or number of users.
3. Batch scheduling, where the use of resources is controlled by a batch system.

Scheduling parameters such as time limits, time slice interval and minimum request size are applied on an individual partition basis. Default priorities, memory limits and CPU usage limits can be applied to users or projects to tune the partition's workload. For details see Chapter 6 (*Access Control, Usage Limits and Accounting*) and Chapter 7 (*RMS Scheduling*).

The partition shown in Figure 2.5 has its CPUs allocated to five parallel jobs. The jobs have been allocated CPUs in two different ways: jobs 1 and 2 use all of the CPUs on each node; jobs 3, 4 and 5 are running with only one or two CPUs per node. RMS allows the user to specify how their job will be laid out, trading off the competing benefits of increased locality on the one hand against increased total memory size on the other. With this allocation of resources, all five parallel programs can run concurrently on the partition.

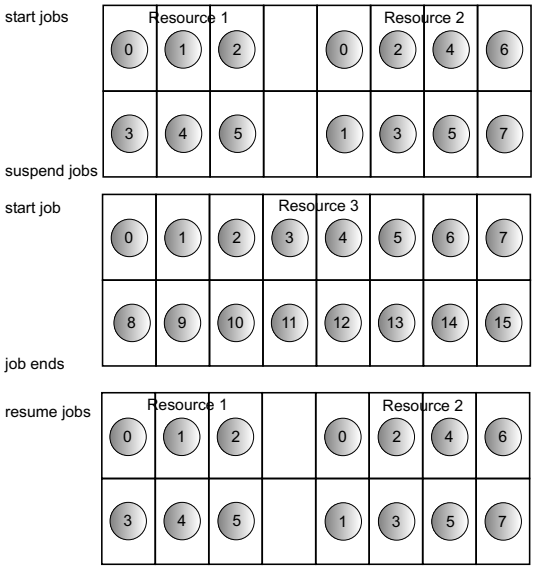
Figure 2.5: Distribution of Processes



The RMS scheduler allocates contiguous ranges of nodes with a given number of CPUs per node [1]. Where possible each *resource request* is met by allocating a single range of nodes. If this is not possible, an unconstrained request (those that only specify the number of CPUs required) may be satisfied by allocating CPUs on disjoint nodes. This ensures that an unconstrained resource request can utilize all of the available CPUs.

The scheduler attempts to find free CPUs for each request. If this is not possible, the request blocks until CPUs are available. RMS preempts programs when a higher priority job is submitted, as shown in Figure 2.6. Initially, CPUs have been allocated for resource requests 1 and 2. When the higher priority resource request 3 is submitted, 1 and 2 are suspended; 3 runs to completion after which 1 and 2 are restarted.

Figure 2.6: Preemption of Low Priority Jobs



2.4.3 Access Control and Accounting

Users are allocated resources on a per-partition basis. Resources in this context include both CPUs and memory. The system administrator can control access to resources both at the individual user level and at the project level (where a project is a list of users). This means that default access controls can be set up at the project level and overridden on an individual user basis as required. The access controls mechanism is described in

[1]The scheduler allocates contiguous ranges of nodes so that processes may take advantage of the Compaq AlphaServer SC Interconnect hardware support for broadcast and barrier operations which operate over a contiguous range of network addresses.

detail in Chapter 6 (*Access Control, Usage Limits and Accounting*).

Each partition, except the `root` partition, is managed by a Partition Manager (see Section 4.4), which mediates user requests, checking access permissions and usage limits before scheduling CPUs and starting user jobs.

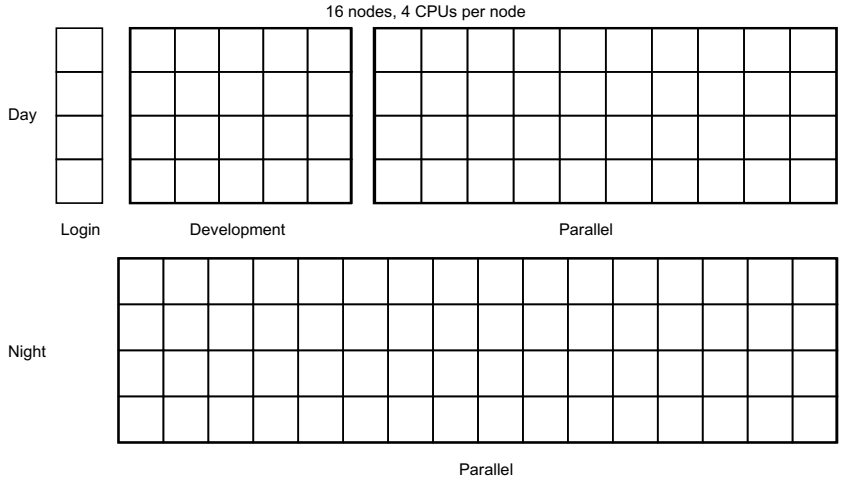
An accounting record is created as CPUs are allocated to each request. It is updated periodically until the resources are freed. The accounting record itemizes CPU and memory usage, indexed by job, by user and by project.

2.4.4 RMS Configuration

The set of partitions active at any time is known as a *configuration*. A system will normally have a number of configurations, each appropriate to a particular operating pattern. For example, there may be one configuration for normal working hours and another for night time and weekend operation.

The CPUs allocated to a partition may vary between configurations. For example, a login partition (nodes allocated for interactive use) may have more nodes allocated during working hours than at night – it may even be absent from the night time configuration. A pair of configurations are shown in Figure 2.7.

Figure 2.7: Two Configurations



RMS supports automated reconfiguration at shift changes as well as dynamic reconfiguration in response to a request from an operator or administrator. The RMS client `rcontrol` (Page 5-20) manages the switch-over from one configuration to another. For automatic reconfiguration, `rcontrol` can be invoked from a `cron` job.

3

Parallel Programs Under RMS

3.1 Introduction

RMS provides users with tools for running parallel programs and monitoring their execution, as described in Chapter 5 (*RMS Commands*). Users can determine what resources are available to them and request allocation of the CPUs and memory required to run their programs. This chapter describes the structure of parallel programs under RMS and how they are run.

A parallel program consists of a controlling process, `prun`, and a number of application processes distributed over one or more nodes. Each process may have multiple threads running on one or more CPUs. `prun` can run on any node in the system but it normally runs in a login partition or on an interactive node.

In a system with SMP nodes, RMS can allocate CPUs so as to use all of the CPUs on the minimum number of nodes (a block distribution); alternatively, it can allocate a specified number of CPUs on each node (a cyclic distribution). This flexibility allows users to choose between the competing benefits of increased CPU count and memory size on each node (generally good for multithreaded applications) and increased numbers of nodes (generally best for applications requiring increased total memory size, memory bandwidth and I/O bandwidth).

Parallel programs can be written so that they will run with varying numbers of CPUs and varying numbers of CPUs per node. They can, for example, query the number of processors allocated and determine their data distributions and communications patterns accordingly (see Appendix C (*RMS Kernel Module*) for details).

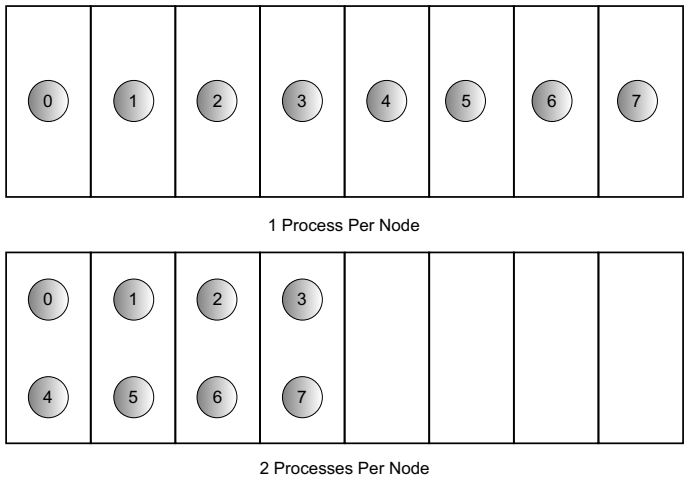
3.2 Resource Requests

Having logged into the system, a user makes a request for the resources needed to run a parallel program by using the RMS commands `prun` (see Page 5-11) or `allocate` (see Page 5-3). When using the `prun` command, the request can specify details such as the following:

- The partition on which to run the program (the `-p` option)
- The number of processes to run (the `-n` option)
- The number of nodes required (the `-N` option)
- The number of CPUs required per process (the `-c` option)
- The memory required per process (the `RMS_MEMLIMIT` environment variable)
- The distribution of processes over the nodes (the `-m`, `-B` and `-R` options)
- How standard input, output and error streams should be handled (the `-i`, `-o` and `-e` options)
- The project to which the program belongs for accounting and scheduling purposes (the `-P` option)

Two variants of a program with eight processes are shown in Figure 3.1: first, with one process per node; and then, with two processes per node.

Figure 3.1: Distribution of Parallel Processes



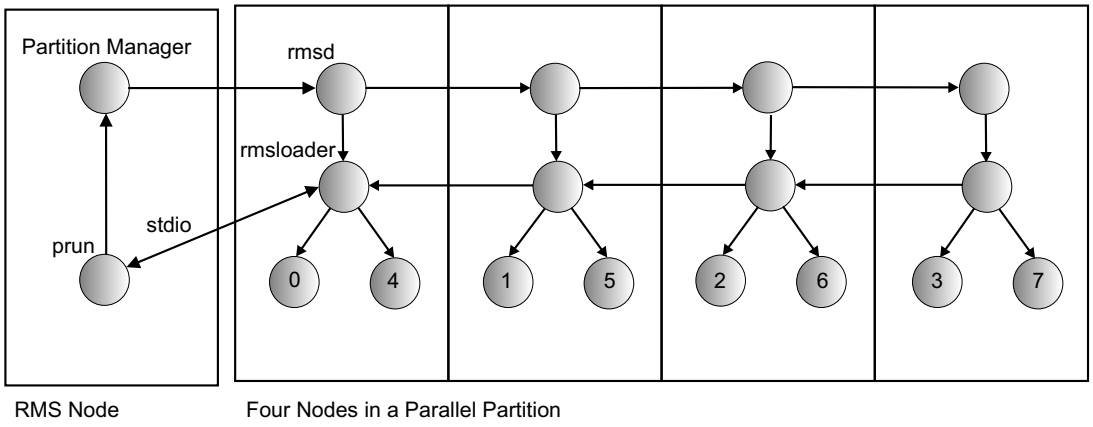
The resource request is sent to the Partition Manager, `pmanager` (described in Section 4.4). The Partition Manager performs access checks (described in Chapter 6 (*Access Control, Usage Limits and Accounting*)) and then allocates CPUs according to the policies established for the partition (see Chapter 7 (*RMS Scheduling*)).

RMS makes a distinction between allocating resources and starting jobs on them. Before the Partition Manager schedules a parallel program, it will ensure that the required CPUs and memory are allocated. Note that this may cause requests to block for longer than you might expect – especially when the job has not specified how much memory it requires. Once CPUs have been allocated, jobs can be started on them immediately.

3.3 Loading and Running Programs

A simple parallel program is shown in Figure 3.2. It has eight application processes, distributed over four nodes, two processes per node.

Figure 3.2: Loading and Running a Parallel Program



Once the CPUs have been allocated, `prun` asks the `pmanager` to start the application processes on the allocated CPUs. The `pmanager` does this by instructing the daemons running on each of the allocated nodes to start the loader process `rmsloader` on the user's behalf.

The `rmsloader` process starts the application processes executing, forwarding their `stdout` and `stderr` streams to `prun` (unless otherwise directed). Meanwhile, `prun` supplies information on the application processes as requested by `rmsloader` and forwards `stdout` and `stderr` to the controlling terminal or output files.

`prun` forwards `stdin` and certain signals (`QUIT`, `USR1`, `USR2`, `WINCH`) to the application processes. If `prun` is killed, RMS cleans up the parallel program, killing the application

processes, removing any core files if requested (see Page 5-11) and then deallocating the CPUs.

The application processes are run from the user's current working directory with the current limits and group rights. The data and stack size limits may be reduced if RMS has applied a memory limit to the program.

During execution, the processes may be suspended at any time by the scheduler to allow a program with higher priority to run. All of the processes in a parallel program are suspended together under the gang-scheduling policy used by RMS for parallel programs (see Chapter 7 (*RMS Scheduling*) for details). They are restarted together when the higher priority program has completed.

A parallel program exits when all of its processes have exited. When this happens, the `rmsloader` processes reduce the exit status back to the controlling process by performing a global `OR` of the exit status of each of the processes. If `prun` is run with verbose reporting enabled, a non-zero exit status is accompanied by a message, as shown in the following example:

```
$ prun -v myprog
...
myprog: process 0 exited with status 1
```

If the level of reporting is increased with the `-vv` option, `prun` provides a commentary on the resource request. With the `-vvv` option, `rmsloader` also outputs information identifying the activity on each node running the program, as shown in the following example.

```
$ prun -vvv myprog
prun: running /home/duncan/myprog
prun: requesting 2 CPUs
prun: starting 2 processes on 2 cpus default memlimit no timelimit
prun: stdio server running
prun: loader 1 starting on atlas1 (10.128.0.7)
prun: loader 0 starting on atlas0 (10.128.0.8)
loader[atlas1]: program description complete
loader[atlas1]: nodes 2 contexts 1 capability type 0xffff8002 entries 2
loader[atlas1]: run process 1 node=5 cntx=244
prun: process 1 is pid 1265674 on atlas1
loader[atlas0]: program description complete
loader[atlas0]: nodes 2 contexts 1 capability type 0xffff8002 entries 2
loader[atlas0]: run process 0 node=4 cntx=244
prun: process 0 is pid 525636 on atlas0
...
```

When the program has exited, the CPUs are deallocated and the scheduler is called to service the queue of waiting jobs.

Sometimes, it is desirable for a user to be granted more control over the use of a resource. For instance, the user may want to run several jobs concurrently or use the same nodes for a sequence of jobs. This functionality is supported by the command `allocate` (see Page 5-3) which allows a user to allocate CPUs in a parallel partition to a UNIX shell. These CPUs are used for subsequent parallel jobs started from this shell. The CPUs remain allocated until the shell exits or a time limit expires (see Section 7.3 and Section 7.4.5).

4

RMS Daemons

4.1 Introduction

This chapter describes the role of the RMS daemons. There are daemons that run on the `rmshost` node providing services for the system as a whole:

`msqld`	Manages the database (see Section 4.2).
`mmanager`	Monitors the health of the machine as a whole (see Section 4.3).
`pmanager`	Controls the use of resources (see Section 4.4).
`swmgr`	Monitors the health of the Compaq AlphaServer SC Interconnect (see Section 4.5).
`tlogmgr`	Carries out transactions on behalf of RMS servers (see Section 4.6).
`eventmgr`	Provides a system-wide event-handling service (see Section 4.7).

There are daemons that run on each node, providing support for RMS functionality on that node:

`rmsmhd`	Acts as the Process Manager, starting all of the other RMS daemons (see Section 4.8).
`rmsd`	Carries out instructions from `pmanager` to run users' programs (see Section 4.9).

4.1.1 Startup

RMS is started as each node executes the initialization script `/sbin/init.d/rms` with the `start` argument on startup. This starts the `rmsmhd` daemon which, in turn, starts the other daemons on that node.

The daemons can also be started, stopped and reloaded individually by `rcontrol` once RMS is running. See Page 5-20 for details.

4.1.2 Log Files

Output from the management daemons is logged to the directory `/var/rms/adm/log`. The log files are called *`daemon`*`.log`, where *`daemon`* gives the name of the RMS daemon, such as `swmgr`. The Partition Managers are distinguished by suffixing `pmanager` with a hyphen and then the name of the partition. For example, the Partition Manager for the partition `par1` is known is `pmanager-par1`.

Errors are logged to `/var/rms/adm/log/error.log`.

Output from `rmsmhd` and `rmsd` is logged to `/tmp/rms.log` on each node.

4.1.3 Daemon Status

The `servers` table contains information on the status of each daemon: the time it was started, its process ID and the name of its host node (see Section 10.2.19 for details of the table structure).

Note that the `status` field in the `servers` table is set to `error` if an error occurs when starting an RMS daemon. The corresponding entry in the `events` table describes what went wrong (see Chapter 8 (*Event Handling*) for details).

The command `rinfo` can be used to get reports on the status of each daemon. See Page 5-32 for details.

4.2 The Database Manager

The Database Manager, `msqld`, manages the RMS database, providing an SQL interface for its clients. Client applications may use C, C++, Java or UNIX scripts to generate SQL queries for `msqld`.

The database holds all state information for RMS. This information is initially created by the RMS client application `rmsbuild` (see Page 5-35). The information is updated by the other RMS daemons as RMS operates. The information can be backed up, restored and generally maintained using the database administration program, `rmstbladm` (see Page 5-44).

4.3 The Machine Manager

The Machine Manager, `mmanager`, is responsible for detecting and reporting changes in the state of each node in the system. It records the current state of each node and any changes in state in the database.

When a node is functioning correctly, `rmsd`, a daemon which runs on each node, periodically updates the database. However, if the node crashes, or IP traffic to and from the node stops, then these updates stop. RMS uses the external monitor, `mmanager`, to check periodically the service level of each node. It monitors whether IP is functioning and whether the RMS daemons on each node are operating.

4.3.1 Interaction with the Database

The Machine Manager records the current status of nodes in the `nodes` table (see Section 10.2.14) while changes to node status are entered in the `events` table (see Section 10.2.6).

The interval at which the Machine Manager performs status checks is set in the `attributes` table (see Section 10.2.3) with the `node-status-poll-interval` attribute. If this attribute is not present, the general attribute `rms-poll-interval` is used instead.

4.4 The Partition Manager

The nodes in the RMS machine are configured into mutually exclusive sets known as *partitions* (see Section 2.4). By restricting access to partitions, the system administrator can reserve particular partitions for specific types of tasks or users. In this way, the system administrator can ensure that resources are used most effectively; for example, that resources intended for running parallel programs are not consumed running user shells. The access restrictions are set up in the `access_controls` table (see Section 10.2.1) of the RMS database.

Each partition is controlled by a Partition Manager, `pmanager`. The Partition Manager mediates each user's requests for resources (CPUs and memory) to run jobs in the partition. It checks the user's access permissions and resource limits before adding the request to its scheduling queue. The request blocks until the resources are allocated for the job.

When the resources requested by the user become available, the Partition Manager instructs `rmsd`, a daemon that runs on each node in the partition (see Section 4.9), to create a communications context for the user's job. Finally, the Partition Manager replies to the user's request and the user's job starts.

The Partition Manager makes new scheduling decisions periodically and in response to incoming resource requests (see Chapter 7 (*RMS Scheduling*) for details). These decisions may result in jobs being suspended or resumed. Such scheduling operations, together with those performed as jobs are killed, are performed by the Partition Manager sending scheduling or signal delivery requests to the `rmsd`s.

The Partition Manager is connected to its `rmsd`s by a tree of sockets. Commands are routed down this tree; they complete when an acknowledgement is returned. For example, jobs are only marked as finished when the Partition Manager has confirmed that all of their processes have exited.

If the tree of sockets is broken by a node crash, the Partition Manager marks the node's partition as blocked and generates an event. The node can then be rebooted or configured out of the machine. If the node is rebooted, the `rmsd`s reconnect and the Partition Manager continues as before. If the node cannot be rebooted then the partition must be halted, the node configured out and the partition restarted. Jobs that spanned the failing node are cleaned up at this point. The other jobs run on unless explicitly killed. Scheduling and signal delivery operations are suspended while the partition is blocked.

4.4.1 Partition Startup

The Partition Manager is started by the `rmsmhd` daemon, running on the `rmshost` node, on instruction from `rcontrol` (see Page 5-20). Once the partition is running, a startup script `/opt/rms/etc/pstartup` is executed. This script performs site-specific and OS-specific actions depending upon the partition type.

4.4.2 Interaction with the Database

The Partition Manager makes updates to the `partitions` table (see Section 10.2.16) when it starts and as CPUs are allocated and freed.

The Partition Manager creates an entry in the `resources` table (see Section 10.2.18) each time a user makes a request for resources to run a job. This entry is updated each time CPUs are allocated or deallocated. The Partition Manager adds an entry to the `jobs` table (see Section 10.2.10) as each job starts, updating it if the job is suspended or resumed and when the job completes.

The Partition Manager creates an entry in the accounting statistics (`acctstats`) table (see Section 10.2.2) when CPUs are allocated. The entry is updated periodically until the request completes.

The Partition Manager consults the `users` table (see Section 10.2.24), the `projects` table (see Section 10.2.17) and the `access_controls` table (see Section 10.2.1) to verify users' access permissions and usage limits.

Configuration information about each partition is held in the `partitions` table (see Section 10.2.16). The information is indexed by the name of the partition together with the name of the active configuration.

4.5 The Switch Network Manager

The Switch Network Manager, `swmgr`, controls and monitors the Compaq AlphaServer SC Interconnect (see Appendix A (*Compaq AlphaServer SC Interconnect Terms*)). It does this using the switch network control interface connected to the parallel port of the primary management node. If `swmgr` detects an error in the switch network, it updates the status of the switch concerned and generates an event.

`swmgr` collects fan, power supply and temperature data from the Compaq AlphaServer SC Interconnect modules, updating status information and generating events if components fail or temperatures exceed their operating limits. See Section 9.5.4 for site-specific details of configuring the `swmgr`.

4.5.1 Interaction with the Database

The Switch Network Manager creates and maintains the entries in the `elites` table (see Section 10.2.5) and the `switch_boards` table (see Section 10.2.22). It maintains entries in the `elans` table (see Section 10.2.4). In the event of errors, it creates entries in the `link_errors` table (see Section 10.2.11).

4.6 The Transaction Log Manager

The Transaction Log Manager, `tlogmgr`, executes change of state requests that have been entered in the `transactions` table (see Section 10.2.23) by RMS administrative clients. This mechanism is employed to serialize changes to the database and to provide an audit trail of such changes.

The entry in the `transactions` table records who requested the change, and names the service required together with any arguments to pass to the process on startup. A transaction handle (a unique ID) is generated for the entry and passed to both the client and the RMS daemon that provides the service.

The RMS daemon uses the transaction handle to label any results it produces, such as an entry in the `transaction_outputs` table (see Section 10.1.3). The client uses the handle to select the result from the relevant table. Output from the service is appended to an output log. The name of this log is entered in the `transactions` table together with the status of the transaction.

The services that are available are listed in the `services` table (see Section 10.2.20).

Each entry in the `services` table specifies which command to run, who can run it and on which host.

4.6.1 Interaction with the Database

The Transaction Log Manager maintains the `transactions` table (see Section 10.2.23). It consults the `services` table (see Section 10.2.20) in order to execute transactions on behalf of its clients.

4.7 The Event Manager

When an RMS daemon detects an anomaly (such as a node crash or a high temperature reading), it writes an event description to the `events` table (see Section 10.2.6). It is the job of the Event Manager, `eventmgr`, to execute recovery scripts that either correct the fault or report it to the operators if manual intervention is required.

On receiving an event notification, the Event Manager looks for a matching entry in the `event_handlers` table (see Section 10.2.7), executing the handler script if it finds a match (see Section 8.2 for details). If no match is found, it runs the default event handler script; this script is site-specific, but it would typically run a command to escalate the event through SNMP or email.

The Event Manager also implements the event-waiting mechanism that enables client applications both to generate and to wait efficiently on a specified event. Typical events include the following:

- Nodes changing state
- Partitions starting
- Transaction log entries being executed

The details that describe the event are held in the `events` table (see Section 10.2.6). The Event Manager's job is to notify interested clients that the event has occurred. This frees the clients from having to poll for the information. For more information on RMS event handling, see Chapter 8 (*Event Handling*).

4.7.1 Interaction with the Database

The Event Manager consults the `events` table (see Section 10.2.6) and the `event_handlers` table (see Section 10.2.7).

4.8 The Process Manager

The Process Manager, `rmsmhd`, is responsible for starting and stopping the other RMS daemons. It runs on each node and is responsible for managing the other daemons that run on its node. It starts them as the node boots, stops them as the node halts and starts or stops them in response to requests from the RMS client application `rcontrol` (see Page 5-20).

4.8.1 Interaction with the Database

RMS stores information regarding which daemons run on which nodes; this information is stored centrally in the RMS database, rather than in node-specific configuration files. On startup, the Process Manager checks the `servers` table (see Section 10.2.19) for entries matching its node. This information is used to start the other daemons. If its child processes (the other daemons) are killed, it checks the table to see whether they should be restarted. The Process Manager creates its own entry in the `servers` table.

4.9 The RMS Daemon

The RMS daemon `rmsd` runs on each node in the machine. Its purpose is as follows:

- To start application processes
- To implement scheduling decisions made by the Partition Manager
- To clean up after parallel programs when they have finished
- To execute RMS remote procedure calls on behalf of clients elsewhere in the network
- To collect accounting data and performance statistics

`rmsd` carries out the following tasks on behalf of the Partition Manager to run a user's parallel program:

- Creating and destroying communication contexts (see Section C.2)
- Starting the application loader, `rmsloader`.
- Delivering signals
- Suspending and resuming processes
- Collecting accounting data from the kernel

The rmsds communicate with each other and with the Partition Manager that controls their node over a balanced tree of sockets. Requests (for example, to deliver a signal to all processes in a parallel program) are passed down this tree to the appropriate range of nodes. The results of each request are combined as they pass back up the tree.

rmsd is started by the RMS daemon rmsmhd and restarted when it exits – this happens when a partition is shut down.

4.9.1 Interaction with the Database

rmsd records configuration information about each node (number of CPUs, amount of memory and so on) in the nodes table (see Section 10.2.14) as it starts. It periodically records usage statistics in the node statistics (node_stats) table (see Section 10.2.15). The interval at which these statistics are sampled is set in the attributes table with the cpu-stats-poll-interval attribute.

rmsd records details of the node's Compaq AlphaServer SC Interconnect configuration in the elans table as it starts (see Section 10.2.4 and Appendix A (*Compaq AlphaServer SC Interconnect Terms*)).

5

RMS Commands

5.1 Introduction

This chapter describes the RMS commands. RMS includes utilities that enable system administrators to configure and manage the system, in addition to those that enable users to run their programs.

RMS includes the following commands intended for use by system administrators:

`rcontrol`	The `rcontrol` command is used to control the system resources.
`rmsbuild`	The `rmsbuild` command creates and populates an RMS database for a given machine.
`rmsctl`	The `rmsctl` script is used to stop and start the RMS system and to report its status.
`rmsquery`	The `rmsquery` command is used to select data from the database and, in the case of system administrators, to update it.
`rmstbladm`	The table administration `rmstbladm` program is used to create a database, to back it up and to restore it.

The following utilities are used internally by RMS and may also be used by system administrators:

`nodestatus`	The `nodestatus` command is used to get or set the status or run level of a node.

`rmshost`	The `rmshost` command reports the name of the node running the RMS management daemons.
`msqladmin`	The `msqladmin` command is used for creating and deleting databases and stopping the mSQL server.

RMS includes the following commands for all users of the system:

`allocate`	The `allocate` command is used to reserve access to a set of CPUs either for running multiple tasks in parallel or for running a sequence of commands on the same CPUs.
`prun`	The `prun` command is used to run a parallel program or to run multiple copies of a sequential program.
`rinfo`	The `rinfo` command is used to determine what resources are available and which jobs are running.
`rmsexec`	The `rmsexec` command is used to run a sequential program on a lightly loaded node.

The following sections describe the commands in more detail, listing them in alphabetical order.

NAME

allocate – Reserves access to CPUs

SYNOPSIS

```
allocate [-hIv] [-B base] [-C CPUs] [-N nodes | all] [-n CPUs]
         [-p partition] [-P project] [-R request]
         [script [args ...]]
```

OPTIONS

`-B` *base*
: Specifies the number of the base node (the first node to use) in the partition. Numbering within the partition starts at 0. By default, the base node is unassigned, leaving the scheduler free to select nodes that are not in use.

`-C` *CPUs*
: Specifies the number of CPUs required per node (default 1).

`-h`
: Display the list of options.

`-I`
: Allocate CPUs immediately or fail. By default, `allocate` blocks until resources become available.

`-N` *nodes* `| all`
: Specifies the number of nodes to allocate (default 1). To allocate one CPU on each node in the partition, use the argument `all` as follows: `allocate -N all`. Either the `-C` option or the `-n` option can be combined with `-N` but not both.

`-n` *CPUs*
: Specifies the total number of CPUs required.

`-P` *project*
: Specifies the name of the project with which the job should be associated for scheduling and accounting purposes.

`-p` *partition*
: Specifies the target partition from which the resources are to be allocated.

`-R` *request*
: Requests a particular configuration of resources. The types of *request* currently supported are as follows:

`immediate=0 | 1`	With a value of 1, this specifies that the request should fail if it cannot be met immediately (this is the same as the `-I` option).
`hwbcast=0 | 1`	With a value of 1, this specifies a contiguous range of nodes and constrains the scheduler to queue the request until a contiguous range becomes available.
`rails=`*n*	In a multirail system, this specifies the number of rails required, where $1 \leq n \leq 32$.

Multiple requests can be entered as a comma-separated list, for example, `-R hwbcast=1,immediate=1`.

`-v` Specifies verbose operation.

DESCRIPTION

The `allocate` program allocates resources for subsequent use by the `prun(1)` command. `allocate` is intended for use where a user wants to run a sequence of commands or several programs concurrently on the same set of CPUs.

The `-p`, `-N`, `-C`, `-B` and `-n` options control which CPUs are allocated. The `-N` option specifies how many nodes are to be allocated. When this option is specified the user is allocated a constant number of CPUs per node (default 1). The `-C` option specifies the number of CPUs required per node. The alternative `-n` option specifies the total number of CPUs to allocate. This option does not force the allocation of a constant number of CPUs per node.

The `-B` option specifies the base of a contiguous range of nodes relative to the start of the partition. The `-N` option specifies its extent. So for example `-B0-N4` specifies the first four nodes in the partition. Note that nodes that have been configured out are excluded. The `-B` option should be used to gain access to a specific file system or device that is not available on all nodes. If the `-B` option is used, the scheduler allocates a contiguous range of nodes and the same number of CPUs on each node. Using this option causes a request to block until the base node and any additional nodes required to run the program are free.

The `-p` option specifies the partition from which CPUs can be allocated. CPUs cannot be allocated from the `root` partition.

The Partition Manager, `pmanager`, allocates processing resources to users as and when the resources are requested and become available. (See Section 4.4). By default, a contiguous range of nodes is allocated to the request where possible. This enables programs to take advantage of the system's hardware broadcast facilities. The `-R` option

can be used with `hwbcast` set to 1 to ensure that the range of nodes allocated is contiguous.

Before allocating resources, the Partition Manager checks the resource limits imposed on the current project. The project can be specified explicitly with the `-P` option. This overrides the value of the environment variable `RMS_PROJECT` or any default setting in the `users` table. (See Section 10.2.24).

The *`script`* argument (with optional arguments) can be used in two different ways, as follows:

1. *`script`* is not specified, in which case an interactive command shell is spawned with the resources allocated to it. The user can confirm that resources have been allocated to an interactive shell by using the `rinfo` command. (See Page 5-32).

   The resources are reserved until the shell exits or until a time limit defined by the system administrator expires, whichever happens first. (See Section 10.2.16).

   Parallel programs, executed from this interactive shell, all run on the shell's resources (concurrently, if sufficient resources are available).

2. *`script`* specifies a shell script, in which case the resources are allocated to the named subshell and freed when execution of the script completes.

ENVIRONMENT VARIABLES

The following environment variables may be used to identify resource requirements and modes of operation to `allocate`. They are used where no equivalent command line options are given.

`RMS_IMMEDIATE` Controls whether to exit (value `1`) rather than block (value `0`) if resources are not immediately available. The `-I` option overrides the value of this environment variable. By default, `allocate` blocks until resources become available. Root resource requests are always met.

`RMS_MEMLIMIT` Specifies the maximum amount of memory required. This must be less than or equal to the limit set by the system administrator.

`RMS_PARTITION` Specifies the name of a partition. The `-p` option overrides the value of this environment variable.

`RMS_PROJECT` Specifies the name of the project with which the request should be associated for accounting purposes. The `-P` option overrides the value of this environment variable.

`RMS_TIMELIMIT` Specifies the execution time limit in seconds. The program will be signaled either after this time has elapsed or after any time limit imposed by the system has elapsed. The shorter of the two time limits is used.

`RMS_DEBUG` Specifies whether to execute in verbose mode and display diagnostic messages. Setting a value of `1` or more will generate additional information that may be useful in diagnosing problems. (See Section 9.6). If this environment variable is not set the `-v` option enables reporting of resource request debug information.

`allocate` passes all existing environment variables through to the shell that it executes. In addition, it sets the following environment variable:

`RMS_RESOURCEID` The identifier of the allocated resource.

EXAMPLES

To run a sequence of jobs on the same CPUs:

```
$ allocate -N 16 jobscript
```

where `jobscript` is a shell script such as the following:

```
#!/bin/sh
# simple job script
prun -n 16 program1
prun -n 16 program2
```

If the script was run directly then each resource request would block until resources became available and there would be no guarantee of both requests using the same CPUs. By running the script under `allocate`, there is only one resource request and both jobs are run on the same CPUs.

To run two programs on the same CPUs at the same time:

```
$ allocate -N 16 -C 2 << EOF
prun program1 &
prun program2 &
rinfo
wait
EOF
```

WARNINGS

In earlier versions, the `-i` option specified immediate mode. This functionality has been moved to the `-I` option. Use of `-i` is now deprecated. If `-i` is specified without an

argument, it is interpreted as `-I` and the user is warned that this feature should not be used anymore.

SEE ALSO

`prun`, `rinfo`

NAME

nodestatus – Gets or sets the status or run level of each node

SYNOPSIS

nodestatus [-bhr] [*status*]

OPTIONS

-b	Operate in the background.
-h	Display the list of options.
-r	Get/set run level.

DESCRIPTION

The nodestatus command is used to update status information in the RMS database as nodes are booted or halted. When run without arguments, nodestatus gets the status of the node on which it is running from the Machine Manager. When run with the -r flag, nodestatus gets the current run level.

When nodestatus is run with the *status* argument, it updates the node's status or, if the -r flag is set, it updates the node's run level. The change is reflected in the nodes table for the node on which the command is running. (See Section 10.2.14). This mechanism is used to track the progress of booting a node. Administrative privileges are required to update the status or run level of a node.

The *status* can be one of these values: not responding, active or running.

Status updates may be delayed if the node running the database server is down. If background operation is specified with the -b option, nodestatus runs in the background and keeps trying until the database server is up and running.

NAME

msqladmin – Perform administrative operations on the mSQL database server

SYNOPSIS

msqladmin [-q] [-f *confFile*] [-h *host*] *command*

OPTIONS

`-f` *confFile*	Specify a non-default configuration file to be loaded. The default action is to load the standard configuration file located in `/var/rms/msql.conf`.
`-h` *host*	Specify a remote hostname or IP address on which the mSQL server (`msql2d`) is running. The default is to connect to a server on the localhost using a UNIX domain socket rather than TCP/IP (which gives better performance).
`-q`	Put `msqladmin` into quiet mode. If this flag is specified, `msqladmin` will not prompt the user to verify dangerous actions (such as dropping a database).

DESCRIPTION

`msqladmin` is used to perform administrative operations on an mSQL database server. Such tasks include the creation of databases, performing server shutdowns and so on. The available commands for `msqladmin` are:

`create` *db_name*	Creates a new database called *db_name*.
`drop` *db_name*	Removes the database called *db_name* from the server. This will also delete all data contained in the database specified.
`shutdown`	Terminates the mSQL server.
`reload`	Forces the server to reload ACL information.
`version`	Displays version and configuration information about the currently running server.

`stats` Displays server statistics.

Most administrative functions can only be executed by the user specified in the run-time configuration as the admin user (`rms`). They can also only be executed from the host on which the server process is running (for example you cannot shut down a remote server process).

EXAMPLES

```
# msqladmin version

Version Details :-

        msqladmin version       2.0.11
        mSQL server version     2.0.11
        mSQL protocol version   23
        mSQL connection         Localhost via UNIX socket
        Target platform         OSF1-V5.0-alpha

Configuration Details :-

        Default config file     /var/rms/msql.conf
        TCP socket              1114
        UNIX socket             /var/rms/adm/msql/msql2.sock
        mSQL user               rms
        Admin user              rms
        Install directory       /var/rms
        PID file location       /var/rms/adm/msql/msql2.pid
        Memory Sync Timer       120
        Hostname Lookup         True
```

NAME

prun – Runs a parallel program

SYNOPSIS

```
prun [-hIOrstv] [-B base] [-c cpus] [-e mode] [-i mode] [-o mode]
     [-N nodes | all] [-n procs] [-m block | cyclic] [-P project]
     [-p partition] [-R request] program [args ...]
```

OPTIONS

`-B` *base*
: Specifies the number of the base node (the first node to use) in the partition. Numbering within the partition starts at 0. By default, the base node is unassigned, leaving the scheduler free to select nodes that are not in use.

`-c` *cpus*
: Specifies the number of CPUs required per process (default 1).

`-h`
: Display the list of options.

`-I`
: Allocate CPUs immediately or fail. By default, `prun` blocks until resources become available.

`-e` *mode*
: Specifies how standard error output is redirected. Valid values for *mode* and their meanings are described below.

`-i` *mode*
: Specifies how standard input is redirected. Valid values for *mode* and their meanings are described below.

`-o` *mode*
: Specifies how standard output is redirected. Valid values for *mode* and their meanings are described below.

`-m block | cyclic`
: Specifies whether to use block (the default) or cyclic distribution of processes over nodes.

`-N` *nodes* `| all`
: Specifies the number of nodes required. To use all nodes in a partition select the `all` argument as follows: `prun -N all`. If the number of nodes is not specified then the RMS scheduler will allocate one CPU per process.

`-n` *`procs`* Specifies the number of processes required. The `-n` and `-N` options can be combined to control how processes are distributed over nodes. If neither is specified, `prun` starts one process.

`-O` Allows resources to be over-committed. Set this flag to run more than one process per CPU.

`-P` *`project`* Specifies the name of the project with which the job should be associated for scheduling and accounting purposes.

`-p` *`partition`* Specifies the partition on which to run the program. By default, the partition specified in the `attributes` table is used. The default is `parallel`. (See Section 10.2.3).

`-R` *`request`* Requests a particular configuration of resources. The types of *`request`* currently supported are as follows:

`immediate=0 | 1`
With a value of 1, this specifies that the request should fail if it cannot be met immediately (the same as the `-I` option).

`hwbcast=0 | 1` With a value of 1, this specifies a contiguous range of nodes and constrains the scheduler to queue the request until a contiguous range of nodes becomes available.

`rails=`*`n`* In a multirail system, this specifies the number of rails required, where $1 \leq n \leq 32$.

Multiple requests can be entered as a comma-separated list, for example, `-R hwbcast=1,immediate=1`.

`-r` Run processes using `rsh`. Used for administrative operations such as starting and stopping RMS.

`-s` Print statistics as the job exits.

`-t` Prefix output with the process number.

`-v` Specifies verbose operation. Multiple `v`s increase the level of output: `-vv` shows each stage in running a program and `-vvv` enables debug output from the `rmsloader` processes on each node.

DESCRIPTION

The `prun` program executes multiple copies of the specified *`program`* on a partition. `prun` automatically requests resources for the program unless it is executed from a shell that already has resources allocated to it. (See Page 5-3).

The way in which processes are allocated to CPUs is controlled by the `-c`, `-n`, `-p`, `-B` and `-N` options. The `-n` option specifies the total number of processes to run. The `-c` option specifies the number of CPUs required per process, this defaults to 1. The `-N` option specifies how many nodes are to be used.

If the `-N` option is not used then the scheduler selects CPUs for the program from any of the available nodes. Where possible RMS will allocate a contiguous range of nodes, but will only be constrained to do so if the `-B` or `-R hwbcast=1` options are set. If the `-N` is used, the scheduler allocates the specified number of nodes (allocating a contiguous range of nodes if possible) and the same number of CPUs on each node. By default, a contiguous range of nodes is allocated to the request where possible. This enables programs to take advantage of the system's hardware broadcast facilities. The `-R` option can be used with `hwbcast` set to 1 to ensure that the range of nodes allocated is contiguous.

The `-B` option specifies the base of a contiguous range of nodes relative to the start of the partition. The `-N` option specifies its extent. So for example `-B0 -N4` specifies the first four nodes in the partition. Note that nodes that have been configured out are excluded. The `-B` option should be used to gain access to a specific file system or device that is not available on all nodes. If the `-B` option is used, the scheduler allocates a contiguous range of nodes and the same number of CPUs on each node. Using this option causes a request to block until the base node and any additional nodes required to run the program are free.

The `-I` option specifies that resource requests should fail if they cannot be met immediately. The default is to block until CPUs are available.

The `-m` option specifies how processes are to be distributed over nodes. The choice is between `block` (the default) and `cyclic`. If a program has `n` processes with identifiers `0,1,...n-1` distributed over `N` nodes then, in a block distribution, the first `n/N` processes are allocated to the first node and so on. If the distribution is cyclic, process 0 runs on the first node, process 1 on the second and so on until process `N-1` is placed on the last node, at which stage the distribution wraps around, with process `N` running on the first node and so on.

The `-p` option specifies the partition to use. If no partition is specified then the default partition is used. The default partition is stored in the `attributes` table. (See Section 10.2.3). Note that use of the `root` partition (all nodes in the machine) is restricted to administrative users.

Before allocating resources, prun checks the resource limits imposed on the current project. The project can be specified explicitly with the -P option. This overrides the value of the environment variable RMS_PROJECT or any default setting in the users table. (See Section 10.2.24).

By default, when running a parallel program, prun forwards standard input to the process with an identifier of 0. The -i option requests a different mode of operation. Valid values for *mode* and their meanings are as follows:

rank	Forward standard input to the process that is identified by *rank* where 0 ≤ *rank* ≤ n-1 and n is the number of processes in the program.
all	Broadcast standard input to all of the processes.
none	Do not forward standard input.
file	prun opens the named *file* and associates it with the standard input stream so that each process reads standard input from the file. If the file does not exist, a read returns EOF.
file.%	prun expands the % character to generate and open a separate file name for each process: process 0 reads standard input from *file*.0, process 1 reads standard input from *file*.1 and so on. If the file does not exist, a read returns EOF.

If the *mode* is rank or all, prun polls its standard input and forwards the data to the rmsloader of the application process (or processes if the *mode* is all). rmsloader writes the data to the standard input pipe for the process. This write may fail if the pipe is full, the application has not read the data. If this happens, rmsloader will periodically attempt to resend the data to the pipe. prun will not poll for further standard input until it has received an acknowledgement from the process (or all processes in the case of broadcast input) to say that this operation has completed.

The -o and -e options control the redirection and filtering of standard output and standard error respectively. Valid values for *mode* and their meanings for these options are as follows:

rank	Redirect to prun standard output (or standard error) from the process identified by *rank* where 0 ≤ *rank* ≤ n-1 and n is the number of processes in the program.
all	Redirect standard output (or standard error) from all processes to prun. This is the default.

none	Do not redirect standard output (or standard error) from any process.
file	prun opens the named *file* for output and associates it with the standard output (standard error) stream so that each process writes standard output (standard error) to the file.
file.%	prun expands the % character to generate and open for output a separate file name for each process: process 0 writes standard output (standard error) to *file*.0, process 1 writes to *file*.1 and so on.

Standard output from a parallel program is line-buffered and redirected to prun when a newline character is received. Output that does not end in a newline is buffered by rmsloader.

Standard error is unbuffered and forwarded to prun as soon as it is received by rmsloader.

There is no global synchronization of output from a parallel program. If multiple processes output data, the order in which the data is output will not necessarily be the same each time the program is run.

prun exits when all of the processes in the parallel program have exited or when one process has been killed. If all processes exit cleanly then the exit status of prun is the global OR of their individual exit status values. If one of the processes is killed, prun will exit with a status value of 128 plus the signal number. prun can also exit with the following codes:

125 One or more processes were still running when the exit timeout expired.

126 prun was run with the -I option and resources were not available.

127 prun was run with invalid arguments.

If an application process started by prun is killed, RMS will run a post mortem core analysis script that generates a backtrace if it can find a core file for the process.

The attribute rms-keep-core in the attributes table determines whether core files are saved. (See Section 10.2.3). The environment variable RMS_KEEP_CORE can be set to override the value in the attributes table.

Core files are saved in the directory *local-corepath/resource-id*. The value of local-corepath is defined in the attributes table. The *resource-id* can be listed by rinfo. (See Page 5-32). prun also sets the environment variable RMS_RESOURCE_ID to the value of the resource identifier.

ENVIRONMENT VARIABLES

The following environment variables may be used to identify resource requirements and modes of operation to `prun`. These environment variables are used where no equivalent command line options are given:

`RMS_IMMEDIATE` Controls whether to exit rather than block if resources are not immediately available. The `-I` option overrides the value of this environment variable. By default, `prun` blocks until resources become available. Root resource requests are always met.

`RMS_KEEP_CORE` Controls whether core files are saved. Overrides the default behaviour set by the system administrator.

`RMS_MEMLIMIT` The maximum amount of memory required per process in megabytes. This must be less than or equal to the limit set by the system administrator.

`RMS_PARTITION` Specifies the name of a partition. The `-p` option overrides the value of this environment variable.

`RMS_PROJECT` The name of the project with which the job should be associated for scheduling and accounting purposes. The `-P` option overrides the value of this environment variable.

`RMS_TIMELIMIT` Specifies the execution time limit in seconds. The program will be signaled either after this time has elapsed or after any time limit imposed by the system has elapsed. The shorter of the two time limits is used.

`RMS_DEBUG` Whether to execute in verbose mode and display diagnostic messages. Setting a value of `1` or more generates additional information that may be useful in diagnosing problems. (See Section 9.6).

`RMS_EXITTIMEOUT`

Specifies the time allowed in seconds between the first process exit and the last. This option can be useful in parallel programs where one process can exit leaving the others blocked in interprocess communication. It should be used in conjunction with an exit barrier at the end of correct execution of the program.

`RMS_STDINMODE` Specifies the *mode* for forwarding standard input to a parallel program. The `-i` option overrides the value of this environment variable. Values for *mode* are the same as those used with the `-i` option.

RMS_STDOUTMODE

Specifies the *mode* for redirecting standard output from a parallel program. The -o option overrides the value of this environment variable. Values for *mode* are the same as those used with the -o option.

RMS_STDERRMODE

Specifies the *mode* for redirecting standard error from a parallel program. The -e option overrides the value of this environment variable. Values for *mode* are the same as those used with the -e option.

prun passes all existing environment variables through to the processes that it executes. In addition, it sets the following environment variables:

RMS_JOBID	The identifier for the job.
RMS_NNODES	The number of nodes used by the application.
RMS_NODEID	The logical identifier of the node within the set allocated to the application.
RMS_NPROCS	The total number of processes in the application.
RMS_RANK	The rank of the process in the application. The rank ranges from 0 to n-1, where n is the number of processes in the program.
RMS_RESOURCEID	The identifier of the allocated resource.

EXAMPLES

In the following example, prun is used to run a four-process program with no specification of where the processes should run.

```
$ prun -n 4 hostname
atlas0.quadrics.com
atlas0.quadrics.com
atlas0.quadrics.com
atlas0.quadrics.com
```

The machine atlas has four CPUs per node and so, by default, the scheduler allocates all four CPUs on one node to run the program. Add the -N option, as follows, to control how the processes are distributed over nodes.

```
$ prun -n 4 -N 2 hostname
atlas0.quadrics.com
atlas0.quadrics.com
atlas1.quadrics.com
atlas1.quadrics.com
$ prun -n 4 -N 4 hostname
atlas1.quadrics.com
atlas3.quadrics.com
atlas0.quadrics.com
atlas2.quadrics.com
```

The `-m` option controls how processes are distributed over nodes. It is used in the following example in conjunction with the `-t` option which tags each line of output with the identifier of the process that wrote it.

```
$ prun -t -n 4 -N 2 -m block hostname
0 atlas0.quadrics.com
1 atlas0.quadrics.com
2 atlas1.quadrics.com
3 atlas1.quadrics.com
$ prun -t -n 4 -N 2 -m cyclic hostname
0 atlas0.quadrics.com
2 atlas0.quadrics.com
1 atlas1.quadrics.com
3 atlas1.quadrics.com
```

The examples so far have used simple UNIX utilities to illustrate where processes are run. Parallel programs are run in just the same way. The following example measures DMA performance between a pair of processes on different nodes.

```
$ prun -N 2 dping 0 1k
  0:        0 bytes      2.33 uSec     0.00 MB/s
  0:        1 bytes      3.58 uSec     0.28 MB/s
  0:        2 bytes      3.61 uSec     0.55 MB/s
  0:        4 bytes      2.44 uSec     1.64 MB/s
  0:        8 bytes      2.47 uSec     3.24 MB/s
  0:       16 bytes      2.55 uSec     6.27 MB/s
  0:       32 bytes      2.57 uSec    12.45 MB/s
  0:       64 bytes      3.48 uSec    18.41 MB/s
  0:      128 bytes      4.23 uSec    30.25 MB/s
  0:      256 bytes      4.99 uSec    51.32 MB/s
  0:      512 bytes      6.39 uSec    80.08 MB/s
  0:     1024 bytes      9.26 uSec   110.55 MB/s
```

The `-s` option instructs `prun` to print a summary of the resources used by the job when it finishes.

```
$ prun -s -N 2 dping 0 32
  0:        0 bytes      2.35 uSec     0.00 MB/s
```

```
  0:        1 bytes      3.60 uSec     0.28 MB/s
  0:        2 bytes      3.53 uSec     0.57 MB/s
  0:        4 bytes      2.44 uSec     1.64 MB/s
  0:        8 bytes      2.47 uSec     3.23 MB/s
  0:       16 bytes      2.54 uSec     6.29 MB/s
  0:       32 bytes      2.57 uSec    12.46 MB/s
Elapsed time         1.00 secs    Allocated time       1.99 secs
User time            0.93 secs    System time          0.13 secs
Cpus used               2
```

Note that the allocated time (in CPU seconds) is twice the elapsed time (in seconds) because two CPUs were allocated.

WARNINGS

In earlier versions, the `-i` option specified immediate mode. This functionality has been moved to the `-I` option. Use of `-i` is now deprecated. If `-i` is specified without an argument, it is interpreted as `-I` and the user is warned that this feature should not be used anymore.

SEE ALSO

`allocate`, `rinfo`

NAME

rcontrol – Controls use of system resources

SYNOPSIS

rcontrol *command* [*args ...*] [-ehs] [-r *level*] [*command args ...*]

OPTIONS

`-e` Exit on the first error.

`-h` Display the list of options.

`-r level` Set reporting level.

`-s` Stop and print warning on error.

command is specified as follows:

`create object [=] name [configuration=val] [partition=val] [attr=val]`

object may be one of: `access_control`, `attribute`, `configuration`, `node`, `partition`, `project`, `user`. If an `access_control` is specified, a `partition` must also be named to identify the object uniquely. Similarly, if a `partition` is specified, a `configuration` must also be named together with a list of nodes.

`remove object [=] name [configuration=val] [partition=val]`

object may be one of: `access_control`, `attribute`, `configuration`, `node`, `partition`, `project`, `user`. If an `access_control` is specified, a `partition` must also be named to identify the object uniquely. If a `partition` is specified, a `configuration` must also be named to identify the object uniquely.

`configure in nodes[=] list`

list specifies a quoted list of nodes, such as `'atlas[1-3,6,8]'`.

`configure out nodes[=] list`

list specifies a quoted list of nodes, such as `'atlas[1-3,6,8]'`.

`start` *`object`* `[=]` *`name`*

`object` may be one of: `configuration`, `partition`, `server`.

`stop` *`object`* `[=]` *`name`* `[option [=] kill | wait]`

`object` may be one of: `configuration`, `partition`, `server`. If `server` is specified as the *`object`*, no `option` should be given.

`reload` *`object`* `[=]` *`name`* `[debug [=]` *`value`*`]`

`object` may be one of: `partition`, `server`.

`suspend` *`job`* `[=]` *`name`* `[`*`name ...`*`]`

`job` may be one of: `resource`, `batchid`.

`suspend` *`attribute`* `[=]` *`value`* `[`*`attribute`* `[=]` *`value ...`*`]`

Attributes of the same name are `OR`ed together. Attributes with different names are `AND`ed together. The result of the logical expression identifies a resource or set of resources as the target of the command.

`resume` *`job`* `[=]` *`name`* `[`*`name ...`*`]`

`job` may be one of: `resource`, `batchid`.

`resume` *`attribute`* `[=]` *`value`* `[`*`attribute`* `[=]` *`value ...`*`]`

Attributes of the same name are `OR`ed together. Attributes with different names are `AND`ed together. The result of the logical expression identifies a resource or set of resources as the target of the command.

`kill` *`job`* `[=]` *`name`* `[`*`name ...`*`]` `[signal [=]` *`sig`*`]`

`job` may be one of: `resource`, `batchid`.

`kill` *`attribute`* `[=]` *`value`* `[`*`attribute`* `[=]` *`value ...`*`]` `[signal [=]` *`sig`*`]`

Attributes of the same name are `OR`ed together. Attributes with different names are `AND`ed together. The result of the logical expression identifies a resource or set of resources as the target of the command.

`set` *`job`* `[=]` *`name`* `priority [=]` *`value`*

`job` may be one of: `resource`, `batchid`.

`set` *`object`* `[=]` *`name attribute`* `[=]` *`value`* `[`*`attribute`* `[=]` *`value ...`*`]`

`object` may be one of: `access_control`, `configuration`, `node`, `partition`, `project`, `user`.

`set` *`attribute`* `[=]` *`name`* `val [=]` *`value`*

`exit`

`help [all |` *`command`*`]`

`show` *`object`* `[=]` *`name`*

`object` may be one of: `nodes`, `configuration`, `partition`.

DESCRIPTION

`rcontrol` is used to manage the following: nodes, partitions and configurations; servers; users and their resource requests, projects and access controls; system attributes.

`rcontrol` can create, start, stop and remove a configuration or partition. It can create, remove and set the attributes of nodes and configure them in and out of the machine. Operations on nodes may specify a single host name, such as `atlas4`, or a *`list`* of host names, such as `'atlas[4-7]'`. Lists of host names must always be quoted.

`rcontrol` can start or stop an RMS server. It can also instruct a running server to reload access control information or change its reporting level.

`rcontrol` can be used to suspend or resume the allocation of CPUs to a resource request, alter its scheduling priority or send a signal to its jobs. Operations on resource requests may specify a request by name or by using the batch system identifier. Alternatively, requests can be identified by attributes such as user name, partition, project or status.

`rcontrol` can be used to create or remove or to set the attributes of users, projects and access controls. Details of which attributes can be modified in this way are specified in the `fields` table in the RMS database. System attributes can also be created, removed or have their value set.

The `help` command prints information on all of the commands and their arguments. When used with the name of a command as an argument, it prints more information on the specified command.

When used without arguments, `rcontrol` runs interactively. A sequence of commands can be entered. Use the `exit` command or `Ctrl/d` to exit.

Most `rcontrol` commands are restricted to administrative users (`root` and `rms` users, by default). The job control commands (`suspend`, `resume`, `kill` and `set priority`) may also be issued by the user running the job in question.

In all of the `rcontrol` commands, the use of the equals sign is optional. The following two examples – using `rcontrol` to configure into the system three nodes named `atlas1`, `atlas2` and `atlas3` – are equivalent.

```
# rcontrol configure in nodes = 'atlas[1-3]'
# rcontrol configure in nodes 'atlas[1-3]'
```

Creating and Removing Nodes

To create a new node description, use `rcontrol` with the `create` command and the argument `node` followed by the hostname of the node. Additional attribute-value pairs specify properties of the node, such as its type and position. The attributes `rack` and `unit` specify the position of the node in the system.

```
# rcontrol create node = atlas1 type = ES40 rack = 0 unit = 3
```

To remove a node description from the RMS database, use `rcontrol` with the `remove` command and the argument `node` followed by the name of the node.

```
# rcontrol remove node = atlas1
```

Creating and Removing Partitions

RMS scheduling policy and access controls are based on partitions. Partitions are non-overlapping sets of nodes. The set of partitions in operation at any time is called the *active* configuration. RMS provides for several operational configurations and includes mechanisms for switching between them with `rcontrol`.

To create a new partition description, use `rcontrol` with the `create` command and the argument `partition` followed by the name of the partition. In addition, you must specify the configuration to which the partition belongs. Additional attribute-value pairs specify properties of the partition: a list of its nodes, its scheduling type, time limit, time slice interval, memory limit or minimum number of CPUs that may be allocated. The `nodes` attribute must be specified. Default values will be selected for the other attributes if none are given.

```
# rcontrol create partition = p1 configuration = day nodes = 'atlas[1-4]' type = parallel
```

The scheduling `type` attribute of the partition may be one of the following:

`parallel`	The partition is for the exclusive use of gang-scheduled parallel programs.
`login`	The partition runs interactive user logins and load-balanced sequential jobs.
`general`	The partition runs all classes of job. This is the default partition type.
`batch`	The partition is for the exclusive use of a batch system.

The timelimit attribute specifies the maximum time in seconds for which CPUs can be allocated on the partition. On expiry of the time limit, jobs will be sent the signal SIGXCPU. If they have not exited within a grace period, they will be killed. The grace period for a site is defined in the attributes table (attribute name grace-period). Its default value is 60 seconds.

The timeslice attribute specifies the period in seconds for which jobs are allocated CPUs before the CPUs may be reallocated to another job of equal priority. The default value for timeslice is NULL, disabling time-slicing.

The memlimit attribute defines the default memory limit per CPU for applications running on this partition. It can be overridden on a per-user or per-project basis. The default value of memlimit is NULL, disabling memory limits unless they are set for specific users or projects.

The mincpus attribute controls the minimum number of CPUs that may be allocated to a job running on this partition. The default value of mincpus is 0. The maximum number of CPUs that can be allocated is controlled on a per-user or per-project basis.

To remove a partition description from the RMS database, use rcontrol with the remove command and the argument partition followed by the name of the partition. You must also specify the name of the configuration since the same partition name may appear in a number of configurations. To remove an entire configuration from the RMS database, use rcontrol with the remove command and the argument configuration followed by the name of the configuration.

```
# rcontrol remove partition = par1 configuration = night
# rcontrol remove configuration = night
```

Note that partitions cannot be removed while they are in use. Similarly, the nodes and type of a partition cannot be changed while the partition is running. If the other attributes of a partition are changed while the partition is running, the Partition Manager is reloaded automatically so that it uses the new information for subsequent jobs. Jobs that are already running are not affected.

Starting and Stopping Partitions

To start a partition in the active configuration, use rcontrol with the start command and the partition argument followed by the name of the partition. To start all of the partitions in a configuration, use rcontrol with the start command and the configuration argument followed by the name of the configuration. A configuration is made active by starting it in this way.

```
# rcontrol start partition = par1
# rcontrol start configuration = day
```

To stop a partition in the active configuration, use `rcontrol` with the `stop` command and the `partition` argument followed by the name of the partition. To stop all of the partitions in the active configuration, use `rcontrol` with the `stop` command and the `configuration` argument followed by the name of the configuration.

When stopping partitions you can optionally specify what should happen to the running jobs. The options are to leave them running, to wait for them to exit or to kill them. The default is to leave them running.

```
# rcontrol stop partition = par1 option = kill
# rcontrol stop configuration = day option = wait
```

Configuring Nodes In or Out

To configure a node in or out, use `rcontrol` with the `configure in` or `configure out` commands. Use the `nodes` argument to specify the list of nodes being configured in or out.

```
# rcontrol configure in nodes = 'atlas[2-4]'
# rcontrol configure out nodes = 'atlas[2,5-7]'
```

Note that partitions must be stopped before nodes can be configured in or out. Jobs may be left running but any jobs running on a node while it is being configured out will be killed. When stopping a partition, it is advisable to wait until jobs have exited (or kill them).

Reloading Database Information

To instruct a Partition Manager to reload its `access_controls`, `users`, and `projects` tables, use `rcontrol` with the `reload` command and the `partition` argument followed by the name of the partition.

```
# rcontrol reload partition = par1
```

To instruct a Partition Manager to change its reporting level, use `rcontrol` with the `reload` command and the `partition` argument followed by the name of the partition. In addition, you should specify the attribute `debug` and a value. The Partition Manager writes its reports to a log file in the directory `/var/rms/adm/log`. See Section 4.1.2 and Section 9.6.

```
# rcontrol reload partition = par1 debug = 1
```

Managing Servers

To stop an RMS server, use `rcontrol` with the `stop` command and the `server` argument followed by the name of the server. To start it again, use `rcontrol` with the

`start` command, the `server` argument and the name of the server. The command `rinfo` (with the `-s` flag) can be used to show the status of the RMS servers.

To instruct an RMS server to change its reporting level, use the `reload` command and the `server` argument with the name of the server. In addition, you should specify the attribute `debug` and a value. RMS servers write their log files to the directory `/var/rms/adm/log` on the rmshost. See Section 9.6.

```
# rcontrol stop server = mmanager
# rcontrol start server = mmanager
# rcontrol reload server = mmanager debug = 1
```

Managing Resources

To instruct the scheduler to suspend the allocation of CPUs to a resource request, use `rcontrol` with the `suspend` command followed by either the name of the resource or the batch system's identifier for the request. This suspends jobs running on the allocated CPUs and decrements the user's CPU usage count.

```
# rcontrol suspend resource = 2234
# rcontrol suspend batchid = 14
```

Note that a resource request that has been suspended by an administrative user cannot be resumed by its owner.

To instruct the scheduler to resume the allocation of CPUs to a resource request, use `rcontrol` with the `resume` command followed by either the name of the resource or the batch system's identifier for the request. This reschedules jobs that were running on the allocated CPUs, unless doing so would cause the user's CPU usage limit to be exceeded.

```
# rcontrol resume resource = 2267
# rcontrol resume batchid = 384
```

To instruct RMS to send a signal to the jobs running on an allocated resource request, use `rcontrol` with the `kill` command followed by either the name of the resource or the batch system's identifier for the request. This kills the jobs running on the allocated CPUs (by sending the signal `SIGKILL` to each process). The optional attribute `signal` can be used to send a specific signal. For example, to send the signal `SIGTERM`:

```
# rcontrol kill resource = 9835 signal = 15
# rcontrol kill batchid = 396 signal = 15
```

To instruct the scheduler to change the priority of a resource request, use `rcontrol` with the `set` command and the `resource` argument followed by either the name of the resource or the batch system's identifier for the request. In addition, you should specify the attribute `priority` and the new value. Priority values range from 0 to 100 (default 50).

```
# rcontrol set resource = 32 priority = 25
# rcontrol set batchid = 48 priority = 40
```

`rcontrol` can also be used to suspend, kill or resume jobs identified by their attributes. The attributes that can be specified are: `partition`, `project`, `status` and `user`. Attributes of the same name are `OR`ed together, attributes with different names are `AND`ed.

For example, to kill a job run by a user called `tom` on the partition `par1` whether its status is blocked or queued:

```
# rcontrol kill user = tom status = blocked status = queued partition = par1
```

To suspend all of the jobs belonging to the project called `science`:

```
# rcontrol suspend project = science
```

Managing Users, Projects and Access Controls

In addition to managing partitions and nodes, `rcontrol` can be used to create, remove and set the attributes of users, projects and access controls. The `fields` table contains details of which objects and attributes may be modified. See Section 10.2.8.

The table has seven fields: the `tablename` field specifies the table that will be modified; the `name` field specifies which entry in the named table will be modified; the `type` field determines the range of valid values; the `min` field gives the minimum for values of `type` integer while the `max` field gives the maximum; the `textattr` field either gives a comma-separated list of valid values or a *`table-name.table-field`* pair. In the case of the *`table-name.table-field`* pair, the value in the `name` field of the `fields` table must also be present in the table named `table-name` in the field called `table-field`. The `access` field specifies whether this field can be updated by the system administrator.

To create a user, use the `rcontrol create` command to specify the object type (in this case, `user`) and the object name (for example, `frank`).

```
# rcontrol create user = frank
```

To update an existing user record, use the `rcontrol set` command. For example, to change the projects to which a user belongs, use `rcontrol set` followed by the object type (in this case, `user`), the object name (in this example, `frank`), the attribute to be changed (`projects`), and its new value (in this example, `parallax`); the new value must already have been defined as a project.

```
# rcontrol set user = frank projects = parallax
```

Note that a user can be in more than one project in which case the value would be a comma-separated list:

```
# rcontrol set user = frank projects = parallax,science
```

To create an access control called, for example, `science`, in the `par1` partition, use `rcontrol` with the `create` command followed by the type of the object, its name and the name of the partition. Additional attribute-value pairs specify attributes of the access control, for example, its class.

```
# rcontrol create access_control = science partition = par1 class = project
```

Just as partitions require a configuration name to identify them uniquely, access controls require a partition name.

To set the attributes of an object, use `rcontrol` with the `set` command followed by the name of the object. Specify the name of the attribute and the required value. An attribute's value can be set to null by entering `NULL`, `Null` or `null` as the value.

```
# rcontrol set access_control = std partition = par1 priority=75 memlimit=NULL
```

To remove an object, use `rcontrol` with the `remove` command and the name of the object.

```
# rcontrol remove user = frank
# rcontrol remove access_control = science partition = par1
```

After changing user, project or access control information, the Partition Managers must be reloaded so that they use the new information.

```
# rcontrol reload partition = par1
```

Jobs that were already running will not be affected by any change to resource limits except that they may be suspended if the new CPU usage limits are lower than before.

Setting System Attributes

System attributes can be created, removed or set using `rcontrol` `create`, `remove` and `set`.

```
# rcontrol create attribute = name val=value
# rcontrol remove attribute = name
# rcontrol set attribute = name val=value
```

Any system attributes can be modified in this way but there are some, mentioned below, whose values are checked if they are created or set. (See Section 10.2.3).

The attribute `pmanager-queuedepth` limits the number of resource requests that a Partition Manager will handle at any time. If the attribute is undefined or set to `NULL` or `0`, no limit is imposed. By default, it is set to `0`.

If a limit is set and reached, subsequent resource requests by `prun` will block or, if the immediate option to `prun` is set, fail. The blocked requests will not appear in the RMS database.

To set the `pmanager-queuedepth` attribute, use `rcontrol` with the `set` command. Specify `attribute`, give the attribute name and set the `val` argument to the required value.

```
# rcontrol set attribute = pmanager-queuedepth val = 20
```

If you set a limit while the partition is running, you should also reload the partition to make the limit take effect.

```
# rcontrol reload partition = par1
```

The attribute `pmanager-idletimeout` limits the amount of time an allocated resource may remain idle. If a resource request exceeds this limit, it will time out with an exit status of 125 and `allocate` will exit with the following message:

```
allocate: Error: idle timeout expired for resource allocation
```

If the attribute is undefined or set to `NULL`, no limit is imposed. By default, it is not set. To set a limit, use `rcontrol` with the `set` argument. Specify `attribute`, give the attribute name and set the `val` argument to the required timeout value in seconds.

```
# rcontrol set attribute = pmanager-idletimeout val = 5
```

If you set a time limit while the partition is running, you should also reload the partition to make the limit take effect.

```
# rcontrol reload partition = par1
```

The attribute `default-priority` determines the default priority given to resource requests. Priorities may range from 0 to 100. The default is 50.

To set the `default-priority` attribute, use `rcontrol` with the `set` command. Specify `attribute`, give the attribute name and set the `val` argument to the required value.

```
# rcontrol set attribute = default-priority val = 75
```

The attribute `grace-period` specifies the amount of time in seconds that jobs are given to exit after they have exceeded their time limit and received a signal to quit. It may be set to any value between 0 and 3600, the default being 60.

The attribute `cpu-poll-stats-interval` specifies the interval between successive polls for gathering node statistics. The interval is specified in seconds and must be in the range 0 to 86400 (1 day).

The attribute `rms-keep-core` determines whether core files are deleted or saved. By default, it is set to `1` so that core files are saved. Change this to `0` to delete core files. The attribute `local-corepath` specifies the directory in which core files are saved. By default, it is set to `/local/core/rms`.

EXAMPLES

The following command line creates a partition called `par1` with eight nodes called `atlas1`, `atlas2` and so on in the configuration called `day`.

```
# rcontrol create partition=par1 configuration=day nodes='atlas[1-8]'
```

The partition is started and stopped as follows:

```
# rcontrol start partition = par1
# rcontrol stop partition = par1
```

Stopping the partition in this way will leave the jobs running. Alternatives are to wait for them to exit or to kill them.

```
# rcontrol stop partition = par1 option = wait
# rcontrol stop partition = par1 option = kill
```

If the system has several operating configurations, for example, one for the prime shift (called `day`) and another for evening and weekends (called `night`) then the set of partitions making up a configuration can be started and stopped together:

```
# rcontrol stop configuration = day
# rcontrol start configuration = night
```

To suspend or resume the jobs running on a specified resource:

```
# rcontrol suspend resource = 2212
# rcontrol resume resource = 2212
```

To set the priority of a resource:

```
# rcontrol set resource = 2212 priority = 4
```

To kill the jobs running on some specified resources:

```
# rcontrol kill resource = 2212 2213
# rcontrol kill batchid = 44 45
```

To instruct a Partition Manager to reread the `user`, `projects` and `access_controls` tables:

```
# rcontrol reload partition = par1
```

To enable debug reporting from the RMS scheduler for the partition called `par1`:

```
# rcontrol reload partition = par1 debug = 41
```

NAME

rinfo – Displays resource usage and availability information for parallel jobs

SYNOPSIS

```
rinfo [-chjlmnpqr] [-L [partition] [statistic]] [-s daemon
      [hostname] | all] [-t node | name]
```

OPTIONS

`-c` List the configuration names.

`-h` Display the list of options.

`-j` List current jobs.

`-l` Give more detailed information.

`-m` Show the machine name.

`-n` Show the status of each node. This can be combined with `-l`.

`-p` Identify each active partition by name and indicate the number of CPUs in each partition.

`-q` Print information on the user's quotas and projects.

`-r` Show the allocated resources.

`-L [partition] [statistic]`

Print the hostname of a lightly loaded node in the machine or the specified *partition*. RMS provides a load-balancing service, accessible through `rmsexec`, that enables users to run their processes on lightly loaded nodes, where loading is evaluated according to a given *statistic*. (See Page 5-39).

`-s daemon [hostname] | all`

Show the status of the *daemon*. Used with the argument `all`, `rinfo` shows the status of all daemons running on the `rmshost` node. For daemons that run on multiple nodes, such as `rmsd`, the optional *hostname* argument specifies the hostname of the node on which the daemon is running.

`-t` *node* | *name*

Where *node* is the network identifier of a node, `rinfo` translates it into the hostname; where *name* is a hostname, `rinfo` translates it into the network identifier. See Section A.1 for more information on network identifiers.

DESCRIPTION

The `rinfo` program displays information about resource usage and availability. Its default output is in four parts that identify: the machine, the active configuration, resource requests and the current jobs. Note that the latter sections are only displayed if jobs are active.

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   atlas[0-2]
parallel        2/4   running    01:02:29             atlas[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  atlas0
```

The machine section gives the name of the machine and the active configuration.

For each partition in the active configuration, `rinfo` shows the number of CPUs in use, the total number of CPUs, the partition status, the time since the partition was started, any CPU time limits imposed on jobs, and the node names. This information is extracted from the `partitions` table. See Section 10.2.16. The description of the `root` partition shows the resources of the whole machine.

The resource section identifies the resource allocated to the user, the number of CPUs that the resource includes, the user name, the node names and the status of the resource. The `time` field specifies how long the resource has been held in hours, minutes and seconds.

The job section gives the job identifier, the user name, the number of CPUs the job is using, on which nodes and the status of the job. The `time` field specifies how long the job has been running in hours, minutes and seconds.

EXAMPLES

When used with the `-q` flag, `rinfo` prints information on the user's projects, CPU usage limits, memory limits and priorities.

```
$ rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT   PRIORITY
parallel      project      default      0/8        100          0
parallel      project      divisionA  16/64       none          1
```

In this example, the access controls allow any user to run jobs on up to 8 CPUs with a memory limit of 100MB. Jobs submitted for the `divisionA` project run at priority 1, have no memory limit and can use up to 64 CPUs. 16 of these 64 CPUs are in use.

When used with the `-s` option, `rinfo` prints information on the status of the RMS servers.

```
$ rinfo -l -s all
SERVER               HOSTNAME     STATUS         PID
tlogmgr              rmshost      running        239241
eventmgr             rmshost      running        239246
mmanager             rmshost      running        239260
swmgr                rmshost      running        239252
pmanager-parallel    rmshost      running        239175

$ rinfo -l -s rmsd
SERVER               HOSTNAME     STATUS         PID
rmsd                 atlas0       running        740600
rmsd                 atlas1       running        1054968
rmsd                 atlas2       running        1580438
rmsd                 atlas3       running        2143669
rmsd                 atlasms      running        239212
```

In the above example, the system is functioning correctly. In the following example, one of the nodes has crashed.

```
$ rinfo -l -s rmsd
SERVER               HOSTNAME     STATUS         PID
rmsd                 atlas0       running        740600
rmsd                 atlas1       running        1054968
rmsd                 atlas2       not responding
rmsd                 atlas3       running        2143669
rmsd                 atlasms      running        239212
```

SEE ALSO

`allocate`, `prun`

NAME

rmsbuild – Creates and populates an RMS database

SYNOPSIS

```
rmsbuild [-dhv] [-I list] [-m machine] [-n nodes | -N list]
         [-p ports] [-t type]
```

OPTIONS

`-d`	Create a demonstration database.
`-h`	Display the list of options.
`-I` *list*	Specifies the names of any interactive nodes.
`-m` *machine*	Specifies a name for the machine.
`-n` *nodes*	Specifies the number of nodes in the machine.
`-N` *list*	Specifies the nodes in the machine by name.
`-p` *ports*	Specifies the number of ports on a terminal server (default 32).
`-t` *type*	Specifies the node type.
`-v`	Specifies verbose operation.

Nodes can be specified by number (`-n`) or by name (`-N`) but not both. Lists of node names should be quoted, for example `'atlas[0-15]'`

DESCRIPTION

`rmsbuild` creates a database for a machine of a given size, adding default entries to the `nodes` table and `modules` table. For detailed information on these tables see Section 10.2.14 and Section 10.2.12 respectively.

`rmsbuild` is used during the initial installation of a machine. It should be run on the `rmshost` node. `rmsbuild` runs `rmstbladm` to create a new database or update an existing one. (See Page 5-44).

Detailed information about each node (number of CPUs, amount of memory and so on) is added later by `rmsd` as it starts on each node.

The machine name is specified with the `-m` option. Machines should be given a short name that does not end a digit. Node names are generated by appending a number to the machine name.

Database entries for the nodes are generated by the `-n` or `-N` options. Use `-n` with a number to generate entries for nodes `0` through `n-1`. Use `-N` to generate entries for a named list of nodes such as `atlas[4-8]`.

Some systems include a management server. You should use the `-I` option to specify the management server name and create a description of the management server in the RMS database. To devise the management server name, append the letters `ms` to the machine name; for example, `atlasms`.

`rmsbuild` is run after the system is installed, creating database entries for all installed nodes. Additional entries can be added later if further nodes are installed.

If the demonstration mode is selected with the `-d` option, `rmsbuild` constructs the entries for a demonstration database; that is to say, a database that does not necessarily correspond to the physical resources of the system. Attributes of the nodes that would normally be set by `rmsd` are set to representative values and a default partition is created. The `-d` option is primarily for testing purposes but can be useful when demonstrating RMS. When creating such a database, you should take care to give it a different name from that of your system.

EXAMPLES

To create a description of a 64-node system called `atlas` with one management server, use `rmsbuild` as follows:

```
# rmsbuild -m atlas -I 'atlasms' -N 'atlas[0-63]'
```

To create a machine description for a 128-node system called `demo`, use `rmsbuild` as follows:

```
# rmsbuild -d -m demo -n 128
```

SEE ALSO

`rmstbladm`, `msqladmin`

NAME

rmsctl – Stops, starts or shows the status of the RMS system.

SYNOPSIS

rmsctl [-aehv] [*start* | *stop* | *restart* | *show*]

OPTIONS

-a	Show all servers, when used with the show command.
-e	Only show errors, when used with the show command.
-h	Display the list of options.
-v	Verbose operation

DESCRIPTION

The rmsctl script is used to start, stop or restart the RMS system on all nodes in a machine, and to show status information.

rmsctl starts and stops RMS by executing the /sbin/init.d/rms script on each node. Note that rsh must be enabled for root users in order for this to function correctly.

rmsctl start starts all of the partitions in the active configuration and sets their autostart fields in the servers table to 1. rmsctl stop stops all of the partitions and sets the autostart fields to 0. (See Section 10.2.19).

This contrasts with the behavior of the /sbin/init.d/rms script, run from the rmshost node, which preserves the current state of the active configuration over a stop/start cycle. (See Section 9.3.1).

When used with the command show, rmsctl shows the current status of the system.

EXAMPLES

To stop the RMS system, use rmsctl as follows:

```
# rmsctl stop
RMS service stopped on atlas1
```

```
RMS service stopped on atlas0
RMS service stopped on atlas3
RMS service stopped on atlas2
RMS service stopped on atlasms
```

To start the RMS system, use `rmsctl` as follows:

```
# rmsctl stop
RMS service started on atlas0
RMS service started on atlas1
RMS service started on atlasms
RMS service started on atlas2
RMS service started on atlas3
pmanager-parallel: cpus=16 (4 per node) maxfree=4096MB swap=5171MB no memory limits
pstartup.OSF1: general partition parallel starting
pstartup.OSF1: enabling login on partition parallel
Enabling login on node atlas1.quadrics.com
Enabling login on node atlas3.quadrics.com
Enabling login on node atlas0.quadrics.com
Enabling login on node atlas2.quadrics.com
```

To show the status of the RMS system, use `rmsctl` as follows:

```
# rmsctl show
SERVER               HOSTNAME     STATUS         PID
tlogmgr              rmshost      running        778
eventmgr             rmshost      running        780
mmanager             rmshost      running        789
swmgr                rmshost      running        799
pmanager-parallel    rmshost      running        33357

STATUS          NODES
running         atlas[0-3] atlasms

CPUS            NODES
4               atlas[0-3] atlasms

MEMORY          NODES
4096            atlas[0-3]
1024            atlasms

SWAP SPACE      NODES
5171            atlas0[0-3] atlasms

TMP SPACE       NODES
6032            atlas[0-3]
5703            atlasms
```

SEE ALSO

rcontrol

NAME

rmsexec – Runs a sequential program on a lightly loaded node

SYNOPSIS

rmsexec [-hv] [-p *partition*] [-s *stat*] [*hostname*] *program* [*args ...*]

OPTIONS

`-h`	Display the list of options.
`-v`	Specifies verbose operation.
`-p` *partition*	Specifies the target partition. The request will fail if load-balancing is not enabled on the partition. (See Section 10.2.16).
`-s` *stat*	Specifies the statistic on which to base the load-balancing calculation (see below).

DESCRIPTION

The `rmsexec` program provides a mechanism for running sequential programs on lightly loaded nodes – nodes, for example, with free memory or low CPU usage. It locates a suitable node and then runs the *program* on it.

The user can select a node from a specific partition (of type `login` or `general`) with the `-p` option. Without the `-p` option, `rmsexec` uses the default load-balancing partition (specified with the `lbal-partition` attribute in the `attributes` table). In addition, the *hostname* of the node can be specified explicitly. The request will fail if this node is not available to the user. System administrators may select any node.

The `-s` option can be used to specify a statistic on which to base the loading calculation. Available statistics are:

`usercpu`	Percentage of CPU time spent in the user state.
`syscpu`	Percentage of CPU time spent in the system state - a measure of the I/O load on a node.
`idlecpu`	Percentage of CPU time spent in the idle state.

`freemem`	Free memory in megabytes.
`users`	Lowest number of users.

By default, `usercpu` is used as the statistic. Statistics can be used on their own, in which case a node is chosen that is lightly loaded according to this statistic, or you can specify a threshold using *`statistic < value`* or *`statistic > value`*

EXAMPLES

Some examples follow:

```
$ rmsexec -s usercpu myprog
$ rmsexec -s "usercpu < 50" myprog
$ rmsexec -s "freemem > 256" myprog
```

SEE ALSO

`rinfo`

NAME

rmshost – Prints the name of the node running the RMS management daemons

SYNOPSIS

rmshost [-hl]

OPTIONS

-h	Display the list of options.
-l	Prints the fully qualified domain name.

DESCRIPTION

The rmshost command prints the name of the node that is running (or should run) the RMS management daemons. It is used by the RMS system.

NAME

rmsquery – Submits SQL queries to the RMS database

SYNOPSIS

rmsquery [-huv] [-d *name*] [-m *machine*] [*SQLquery*]

OPTIONS

`-d` *name*	Select database by name.
`-h`	Display the list of options.
`-m` *machine*	Select database by machine name.
`-u`	Print dates as seconds since January 1st 1970. The default is to print dates as a string created with `localtime(3)`.
`-v`	Verbosely prints field names above each column of output.

DESCRIPTION

`rmsquery` is used to submit SQL queries to the RMS database. Users are restricted to using the `select` statement to extract information from the database. System administrators may also submit SQL statements that update the database: `create`, `delete`, `drop`, `insert` and `update`. Note that queries modifying the database are logged.

When used without arguments, `rmsquery` operates interactively and a sequence of commands can be issued.

When used interactively, `rmsquery` supports GNU readline and history mechanisms. Type `history` to see recent commands, use `Ctrl/p` and `Ctrl/n` to step back and forward through them. The `tables` command lists the tables in the selected database. The command `fields` followed by the name of a table lists the fields in a table. The command `verbose` toggles printing of field names. To quit interactive mode, type `Ctrl/d` or `exit` or `quit`.

`rmsquery` is distributed under the terms of the GNU General Public License. See `http://www.gnu.org` for details and more information on GNU readline and history.

The source is provided in `/usr/opt/rms/src`. Details of the SQL language can be found on the Quadrics support page `http://www.quadrics.com/web/support`.

EXAMPLES

An example follows of a `select` statement that results in a list of the names of all of the nodes in the machine. Note that the *`query`* must be quoted. This is because `rmsquery` expects a single argument.

```
$ rmsquery "select name from nodes"
atlas0
atlas1
atlas2
atlas3
```

In the following example, `rmsquery` is used to print information on all jobs run by a user:

```
$ rmsquery "select name,status,hostnames,cpus,startTime,endTime from \
                      resources where username='user'"
7     finished  atlas[0-3]   4      12/21/99 11:16:44  12/21/99 11:16:46
8     finished  atlas0       2      12/21/99 11:54:23  12/21/99 11:54:29
9     finished  atlas[0-3]   4      12/21/99 11:54:35  12/21/99 11:54:39
```

The `-v` option prints field names. In the following example, `rmsquery` is used to print resource usage statistics:

```
$ rmsquery -v "select * from acctstats"
name  uid   project  started             etime   atime   utime   stime  ...
-------------------------------------------------------------------------
7     1507  1        12/21/99 11:16:44    2.00    8.00    0.10    0.22  ...
8     1507  1        12/21/99 11:54:23    6.65   13.30   10.62    0.10  ...
9     1507  1        12/21/99 11:54:35    4.27   16.63   12.28    0.44  ...
```

When used without arguments, `rmsquery` operates interactively and a sequence of commands can be issued.

```
$ rmsquery -v
sql> select name, status from partitions
name       status
------------------
login      running
parallel   running
sql>
```

NAME

rmstbladm – Database administration

SYNOPSIS

rmstbladm [-BcdDfhmuv] [-r *file*] [-t *table*] [*machine*]

OPTIONS

`-B`	Dump the first five rows of each table to `stdout` as a sequence of SQL statements. A specific table can be dumped if the `-t` option is used.
`-c`	Clean out old entries from the node statistics (`node_stats`) table, the `resources` table, the `events` table and the `jobs` table. (See Chapter 10 (*The RMS Database*). `rmstbladm` uses the `data-lifetime` and `stats-lifetime` attributes, specified in the `attributes` table, to determine how many entries are to be removed. The default is to keep statistics for 24 hours and job descriptions for 48 hours.
`-d`	Dump the contents of the database to `stdout` as a sequence of SQL statements. A specific table can be dumped if the `-t` option is used.
`-D`	Dump the contents of the database to `stdout` as plain text. A specific table can be dumped if the `-t` option is used.
`-f`	Recreate the database from scratch. A specific table can be recreated if the `-t` option is used.
`-h`	Displays the list of options.
`-m`	Displays the names of machines in the RMS databases managed by the `msqld` server.
`-u`	By default, `rmstbladm` checks the consistency of the database. If the `-u` flag is specified, the database is updated to the current revision level. A specific table can be updated if the `-t` option is used.
`-v`	Specifies verbose operation.
`-r` *file*	Restore database tables from the named file.
`-t` *table*	Specifies a single table to be worked on.

DESCRIPTION

The command `rmstbladm` is used to administer the RMS database. It creates the tables and their default entries. It can be used to back up individual tables (or the whole database) to a text file, to restore tables from file or to force the recreation of tables. Unless a specific *`machine`* is specified, `rmstbladm` operates on the database of the host machine.

When installing or upgrading a system, `rmstbladm` is used to check the consistency of the database, to change the structure of the tables and to add default entries. Once the system is installed and working correctly, the database should be backed up using `rmstbladm` with the `-d` option. The backup should be kept safely so that the database can be restored later should this prove necessary.

Certain tables in the RMS database (the `resources`, `jobs`, `events`, `acctstats` and `node_stats` tables in particular) grow over time and as each job is run. To remove old entries from the database, use `rmstbladm` with the `-c` option. Note that this does not remove entries from the accounting statistics table. These should be removed once the accounting data has been processed. (See Section 9.4.5).

Access to `rmstbladm` options that update the database is restricted to administrative users.

EXAMPLES

To backup the contents of the RMS database or a selected table to a text file as a sequence of SQL statements:

```
$ rmstbladm -d > backup_full.sql
$ rmstbladm -d -t nodes > backup_nodes.sql
```

To update the database on installing a new version of RMS:

```
$ rmstbladm -u
```

6

Access Control, Usage Limits and Accounting

6.1 Introduction

RMS access controls and usage limits operate on a per-user or per-project basis (a project is a list of named users). Each partition may have its own controls. This mechanism allows system administrators to control the way in which the resources of a machine are allocated amongst the user community.

RMS accounts for resource usage by user and by project. As each request is allocated CPUs, an accounting record is created, containing the `uid` of the user, the project name, the resource identifier and information on resource usage (see Section 6.5). This record is updated periodically while the CPUs remain allocated.

6.2 Users and Projects

When a system is first installed, there is only one project, called the *default* project. All users are members of this project and anyone who has logged into the system can request all of the CPUs. This simple setup is intended for a single class of cooperating users.

To account for resource usage by user or by project, the administrator must create additional user and project records in the RMS database. To control the resource usage of individuals or groups of users, the administrator must, in addition, create access

control records.

When submitting requests for CPUs, users can select any project of which they are a member (by setting the `RMS_PROJECT` environment variable or by using the `-P` flag when executing `prun` or `allocate`). RMS rejects requests to use projects that do not exist or requests to use projects of which the user is not a member. Users without an RMS user record are subject to the constraints on the default project.

In general, each user is a member of several projects, while projects may have many users. Membership of a project is specified in the `users` table with the `projects` field (see Section 10.2.24). The value of `projects` may be either a single name or list of project names, separated by commas or space. The wildcard character, `*`, may be entered as a project name, denoting that the user is a member of all projects. The ordering of the names in the list is significant: the first project specified becomes the user's default project.

User and project records are created by the system administrator and stored in the `users` and `projects` tables (see Section 10.2.24 and Section 10.2.17).

6.3 Access Controls

Access control records specify the maximum resource usage of a user or project on a given partition. They are created by the system administrator using `rcontrol` or `rmsquery` and stored in the `access_controls` table (see Section 10.2.1).

Each entry specifies the following attributes:

name The name of the user or project.

class Whether the entry refers to a user or a project.

partition The partition to which the access control applies.

priority The default priority of requests submitted by this user or project. Priorities range from 0, the lowest priority, to 100. The default is 50.

maxcpus The total number of CPUs that this user or project can have allocated at any time.

memlimit The maximum amount of memory in megabytes per CPU that can be allocated.

A suspended request does not count against a user's or project's maximum number of CPUs. However, when the request is resumed, a usage check is performed and the request is blocked if starting it would take the user or project over their usage limit.

The access controls for individual users must set lower limits than those of the projects of which they are a member. That is to say, they must have a lower priority, smaller number of CPUs, smaller memory limit and so on than the access control record for the project. Where a memory limit exists for a user or project, it takes precedence over any default limit set on the partition (see Section 10.2.16).

When the system is installed, there are no access control records. If there is no default access control record in the database when a Partition Manager starts, it creates one using information from the partition. The memory limit is set to that of the partition, the priority is 0 and the CPU usage limit is equal to the number of CPUs in the partition. If the partition has no memory limit then all jobs run with memory limits disabled until access control records are created.

6.3.1 Access Controls Example

To illustrate how the RMS access controls mechanism works, we consider an example in which a system is primarily intended for use by Jim, Mary and John, members of the project called design. When they are not using the system, anyone else can submit small jobs.

First, create a project record for design:

```
rcontrol create project = design description = "System Design Team"
```

name	id	description
design	1	System Design Team

Now create user records for Jim, Mary and John:

```
rcontrol create user = jim project = design
rcontrol create user = mary project = design
rcontrol create user = john project = design
```

name	projects
jim	design
mary	design
john	design

Now create access controls for the design project and for the default project (all other users):

```
rcontrol create access_control = design class = project partition = \
```

```
    parallel priority = 5
rcontrol create access_control = default class = project partition = \
    parallel priority = 0 memlimit = 256
```

name	class	partition	priority	maxcpus	memlimit
design	project	parallel	5	Null	Null
default	project	parallel	0	Null	256

Requests submitted by Jim, Mary and John run at priority 5, causing other users' jobs to be suspended if running. These requests are not subject to CPU or memory limits.

Requests submitted by other users run at priority 0 and are subject to a memory limit of 256MB per CPU. Note that on a system with 4 CPUs and 4GB of memory per node, it would be necessary for each node to have at least 5GB of swap space to ensure that jobs submitted by the design group were not blocked by other users (see Section 7.4.2 for details).

In this example, we have not set the `maxcpus` limit as we do not mind how many CPUs the users allocate. This limit could be set if there were two groups of users of equal priority and you wanted to bound the number of CPUs that each could allocate.

6.4 How Access Controls are Applied

The rules governing memory limits, priority values and CPU usage limits are described in more detail in the following sections.

6.4.1 Memory Limit Rules

Memory limits for a resource request are derived by applying the following rules in sequence until an access control record with a memory limit is found.

1. The root user has no memory limits.
2. If the user has an access control record for the partition, the memory limit in the access control record applies.
3. The access control record for the user's current project determines the memory limit.
4. The access control record for the default project determines the memory limit.

Having selected an access control record, the memory limit for the program is set by the value of its `memlimit` field. A null value disables memory limits. Other values are interpreted as the memory limit in megabytes for each CPU. A process with one CPU

allocated has its memory limits set to this value. A process with more than one CPU allocated has proportionately higher memory limits.

The `RMS_MEMLIMIT` environment variable can be used to reduce the memory limit set by the system, but not to raise it.

By default, the memory limit is capped by the minimum value for any node in the partition of the smaller of these two amounts:

1. The amount of memory on the node.
2. The amount of swap space.

If lazy swap allocation is enabled (see Section 7.4.2), the memory limit is capped by the minimum value for any node in the partition of the amount of memory per node.

6.4.2 Priority Rules

The priority of a resource request is derived by applying the following rules in sequence until an access control record with a priority is found.

1. The root user has priority over all other users.
2. If the user has an access control record for the partition then this record determines the priority.
3. The access control record for the user's current project determines the priority.
4. The access control record for the default project determines the priority.

Having selected an access control record, the priority of the resource request is set by the value of its priority field. A null value sets the priority to 50, the default. Higher priority jobs are scheduled first. The user can instruct `rcontrol` to lower the initial priority but not to raise it. An administrator can raise or lower priorities.

6.4.3 CPU Usage Limit Rules

RMS keeps track of the number of CPUs in use by each user and each project. A request to allocate additional CPUs is blocked if it would cause the usage limit for the user or the usage limit for the user's current project to be exceeded. The request remains blocked until the user or other users in the user's current project free enough CPUs to allow the request to be granted. The CPUs can be freed either because the resources are deallocated or because the user suspends the resource using `rcontrol`.

The CPU usage limit is derived by applying the following rules in sequence until an access control record with a CPU usage limit is found.

1. No CPU usage limits are set on jobs run by the root user.
2. If the user has an access control record for the partition, the CPU usage limit is determined by the `maxcpus` field in this record.
3. The access control record for the user's current project determines the CPU usage limit.
4. The access control record for the default project determines the CPU usage limit.

CPU usage limits can be set to a higher value than the actual number of CPUs available in the partition. This is useful if gang scheduling and time slicing are in operation on the partition. For example, if a partition has 16 CPUs and the usage limit for a given user is 32 then RMS will allow two 16 CPU jobs to run (see Section 7.4.3 for details).

6.5 Accounting

As each request is allocated CPUs, an entry is added to the accounting statistics (`acctstats`) table (see Section 10.2.2) specifying the following details about the job:

name Resource name (see Section 10.2.18).

uid Identifier of the user.

project Name of the user's current project.

started Time at which resources were allocated (UTC).

etime Elapsed time (in seconds) since CPUs were allocated.

atime Time (in CPU seconds) for which CPUs have been allocated. Note that `atime` stops ticking while a request is suspended.

utime Time (in seconds) for which processes were executing in user state.

stime Time (in seconds) for which processes were executing in system state.

cpus Number of CPUs allocated.

mem Maximum memory extent of the program in megabytes.

pageflts Number of page faults requiring I/O summed over processes.

memint Memory integral for the program in megabyte hours.

running Set to show that the CPUs are in use.

Accounting records are updated periodically until the CPUs are deallocated. The `running` flag is set to 0 at this point.

The `atime` statistic is summed over all CPUs allocated to the resource request. The `utime` and `stime` statistics are accumulated over all processes in all jobs running on the allocated CPUs.

Note

The `memint` statistics are not implemented in the current release. All values for this fields are 0.

7

RMS Scheduling

7.1 Introduction

The Partition Manager (see Section 4.4) is responsible for scheduling resource requests and enforcing usage limits. This chapter describes the RMS scheduling policies and explains how the Partition Manager responds to resource requests.

7.2 Scheduling Policies

The scheduling policy in use on a partition is controlled by the `type` attribute of the partition. The `type` attribute can take one of four values:

login Normal UNIX time-sharing applies. This scheduling policy is used for partitions that do not run parallel programs, such as interactive login partitions.

In addition, RMS supports load-balanced sequential processing, whereby users can request to have sequential programs executed on a lightly loaded node. Load balancing is enabled on a per-partition basis by an entry in the `partitions` table (see Section 10.2.16). `rmsexec` (see Page 5-39) can be used to run a program with load balancing.

parallel A *gang scheduling* policy is used. This is for partitions intended for production runs of parallel programs. With gang scheduling, the scheduling decisions apply to all processes in a parallel program

together. That is to say, all of the processes in a program are either running or suspended at the same time.

Gang scheduling is required for tightly coupled parallel programs which communicate frequently. It becomes increasingly important as the rate of interprocess communication increases. For example, if a program is executing a *barrier* synchronization, all processes must be scheduled before the barrier completes.

Effective scheduling of parallel programs requires that user access through commands such as `rsh`, `rlogin` and `telnet` is disabled. This is carried out by the partition startup script (see Section 4.4).

general The scheduling policy supports UNIX time-sharing with load balancing and gang scheduling. It is appropriate for a login partition that is used for developing and debugging parallel programs.

batch The scheduling policy is determined by a batch system. It is appropriate for partitions that are for the exclusive use of a batch system. The batch system may run sequential or parallel programs as it wishes but interactive use is prohibited.

7.3 Scheduling Constraints

The scheduling decisions made while gang scheduling are controlled by a number of parameters. These parameters can be specified for individual users and for projects (groups of users) in the `access_controls` table (see Section 10.2.1). Restrictions on the partition itself are specified in the `partitions` table (see Section 10.2.16). The parameters are as follows:

Priority

Each resource request is assigned a priority taken from the `priority` field of the `access_controls` table. The Partition Manager schedules resource requests in order of priority. Where a number of requests are queued with the same priority, they are scheduled by order of submission time. The submission into the queue of a high priority request may cause existing low priority jobs to be suspended. Changing the priority of a request requires administrator privileges.

Maximum Number of CPUs

An upper limit can be set on the number of CPUs that may be allocated to a user or project at any point in time. Requests that take the usage count for the user or project above this limit are blocked. Requests for more CPUs than the limit on a user or project are rejected.

Time Limit

Jobs are normally run to completion or until they are preempted by a higher priority request. Each partition may have a time limit associated with it which restricts the amount of time the Partition Manager may allow for a parallel job. On expiry of this time limit, the job is sent a `SIGXCPU` signal. A period of grace is allowed following this signal for the job to clean up and exit. After this period, the job is killed and the resource deallocated. The duration of the grace period is specified in the `attributes` table (see Section 10.2.3) and can be set using `rcontrol`.

Memory Size

The Partition Manager can enforce memory limits that restrict the size of a job. The default memory limits are designed to prevent memory starvation (a node having free CPUs but no memory) and to control whether parallel jobs page or not.

7.4 What Happens When a Request is Received

A user's request for resources, made through the RMS commands `prun` or `allocate`, specifies the following parameters:

cpus The total number of CPUs to be allocated.

nodes The number of nodes across which the CPUs are to be allocated. This parameter is optional.

base node The identifier of the first node to be allocated. This parameter is optional.

hwbcast A contiguous range of nodes. This parameter is optional. When a contiguous range of nodes is allocated to a job, messages can be broadcast in hardware. This offers advantages of speed over a software implementation if the job makes use of broadcast operations.

memory The amount of memory required per CPU. This parameter is optional (set through the environment variable `RMS_MEMLIMIT`) but jobs with low memory requirements may be scheduled sooner if they make these requirements explicit.

time limit The length of time for which the CPUs are required. This parameter is optional (set through the environment variable `RMS_TIMELIMIT`).

samecpus The same set of CPUs on each node. This parameter is optional.

immediate The request should fail rather than block if resources are not available immediately.

Note

The RMS scheduler attempts to allocate CPUs on a contiguous range of nodes. If a contiguous range of nodes is not available then requests that explicitly specify a contiguous range with the `hwbcast` parameter will block if the requested CPUs cannot be allocated.

When the Partition Manager receives a request, it first checks to see if the partition has sufficient resources. If the resources are available, the next check is on the resource limits applied to the user and the project. If these checks fail, the request is rejected.

If the checks succeed, the scheduler attempts to allocate CPUs from those that are currently free. If sufficient CPUs are free but allocating them would exceed the user's CPU usage limit, the request is marked as `blocked` (or, if the immediate parameter is set, the request fails). If CPUs can be allocated, the resource request is marked as `allocated` and job(s) may use the CPUs. If the request cannot be met, it is added to the list of active requests and marked as `queued`. The scheduler than re-evaluates the allocation of CPUs to all of the requests in the list.

The list of resource requests is sorted in priority order. Requests of the same priority are sorted by submission time. When evaluating the list, the scheduler works down the requests trying to allocate CPUs to them. The highest priority request is allocated CPUs first except when doing so would cause the system to run out of swap space (see Section 7.4.2).

In considering each request, the scheduler first looks at whether it has already been allocated CPUs (a *bound* request). CPUs remain allocated to a request unless they are preempted by a higher priority request, in which case the request of lower priority is suspended together with any jobs that were running on it. If the request is not yet bound then CPUs are allocated, if sufficient are free.

The list of requests is re-evaluated when free CPUs cannot be found for a new request, when an existing request completes or on the expiry of the time-slice period (see Section 7.4.3).

Consider what happens when a high priority request is submitted to a partition that is already running jobs. If there are sufficient CPUs free (matching the constraints of the request) then the job(s) start. If there are not enough free CPUs, the list of requests is re-evaluated. CPUs are allocated to the high priority request and its job(s) are allowed to start. The jobs of the lower priority requests, whose CPUs were taken for the high priority request, are suspended. Any of the low priority jobs for which CPUs are available continue.

7.4.1 Memory Limits

If memory limits are enabled (by setting the `memlimit` attribute of a partition or access control) then a request is only allocated CPUs on nodes that have sufficient memory available. RMS enforces memory limits by setting the data and stack size limits on a process. If the process exceeds the allowed size, it is killed (and the parallel program terminated).

Users, whose programs consume a known amount of memory each time they run, can set their own memory limit with the environment variable `RMS_MEMLIMIT`. Setting this variable (especially if the new value is much smaller than their default memory limit) may cause their jobs to be scheduled sooner than would otherwise be the case. Users cannot raise their memory limits above the level set by the system administrator. They can only lower them.

The default memory limit is calculated by dividing the memory available by the number of CPUs per node. For example, if a node has 4GB of memory and 4 CPUs then each CPU that is allocated comes with 1GB of memory. Larger memory limits can be set but this risks having CPUs idle through memory starvation, unless there is a plentiful supply of jobs requesting small amounts of memory.

If memory limits are enabled, the RMS scheduler keeps track of the maximum memory usage per node. The ratio of memory limit to memory size determines how many requests can be present (allocated or suspended) before jobs start to page.

7.4.2 Swap Space

By default, the operating system reserves swap space as a program allocates memory. Hence, a process requiring 1GB of memory must also have 1GB of swap space. If memory limits are enabled, RMS does not allocate CPUs to new requests if the addition of their maximum memory usage to that already allocated would cause the total for the node to exceed the swap space available.

Each node normally has significantly more swap space than memory. The ratio of memory limit to swap space determines how many requests (allocated or suspended) can be present on each node.

Tru64™ UNIX supports a `lazy` swap allocation policy in which swap space is only allocated when required. If this policy is enabled then RMS uses the total memory available on the node to limit the size and number of jobs run. This enables large memory jobs to run on nodes with relatively little swap space.

Warning

If `lazy` swap allocation is enabled, there must be sufficient swap space for the UNIX daemons and any other processes running on such nodes. Without this,

processes (including those belonging to the system) will be killed if the system runs out of swap space.

7.4.3 Time Slicing

Time slicing is enabled on a partition by setting its `timeslice` attribute; values of 15–120 seconds are recommended. If a `timeslice` is set, the Partition Manager evaluates the list of requests periodically. The list of requests is still sorted by priority but requests of the same priority are sorted on the number of time slices since they were last scheduled (rather than the submission time). When the system has requests for more CPUs than are available, the scheduler suspends requests at the end of each time slice so that others can use the CPUs.

When setting up a system for time slicing, it is important to set memory limits that ensure that all jobs remain resident in memory. System performance will be poor if time slicing between large jobs causes paging. The ratio of memory limit to memory size controls how many requests can progress concurrently. For example, on a node with 4 CPUs and 4GB of memory, setting a memory limit of 512MB will allow two jobs to be time sliced without paging.

The scheduler processes resource requests as it receives them. It tries to fit new requests to free CPUs. If no CPUs are available the request blocks, at least until the next time slice.

7.4.4 Suspend and Resume

The allocation of CPUs to a request can be suspended using `rcontrol`. Doing this reduces the CPU usage counts for the user and project, enabling other jobs to start. Either the user or administrator can resume the allocation at a later time. To resume the jobs, the CPUs are reallocated unless doing so would exceed a CPU usage limit. In this case, the request is marked as `blocked` and CPUs will only be allocated and the jobs restarted when sufficient CPUs become available. Note that requests that are suspended by the administrator cannot be resumed by their owner.

7.4.5 Idle Time

The amount of time that the resources allocated to a request can remain idle can be constrained by using `rcontrol` to set an idle timer (see also Section 10.2.3). By default, no timer is set. If an idle timer is set, it starts timing as soon as the resource has been allocated. It is stopped if the request is suspended and restarted when the request is resumed. If the idle timeout expires the CPUs are deallocated.

8

Event Handling

8.1 Introduction

RMS includes a general mechanism for posting, waiting on and handling events. This functionality is provided by the Event Manager, `eventmgr` (see Section 4.7).

Events are specified by RMS class, name, type and description.

class The class of object generating the event

name The instance of the object

type The event type

description A text description of the event

Generally, the description is either a single word that an event handler script can act on or a full description of some problem.

Events have a string representation as follows:

```
class:name:type:description
```

For example, the following event signifies that the new status of the node `atlas0` is `running`.

```
node:atlas0:status:running
```

The Event Manager writes events to the `events` table in the RMS database. The following query prints the contents of the `events` table in time order:

```
$ rmsquery -v "select * from events order by ctime"
id  name    class  type    ctime              handled description
-------------------------------------------------------------
20  atlas0  node   status  05/04/01 15:53:02  1       running
21  atlas0  node   status  05/05/01 11:27:29  1       not responding
```

8.1.1 Posting Events

Events are normally posted by RMS servers but they can also be generated by the command line utility `rmspost`. This is useful for testing the response of the system to rare events. It can be run with a single argument as follows:

```
rmspost "class:name:type:description"
```

Alternatively, it can be run with 4 arguments as follows:

```
rmspost class name type "long description"
```

Note that the multiple word description, given as the fourth argument, must be quoted.

8.1.2 Waiting on Events

The command line utility `rmswait` waits on events. It can be run with a single argument as follows:

```
rmswait "class:name:type"
rmswait ":name:"
```

Alternatively, flags can be used to specify the class, name and type. The following example specifies the class with the `-c` option and the name with the `-n` option. The `-t` flag is used to specify the type of the event.

```
$ rmswait -c node -n atlas0
```

`rmswait` completes when a matching event is posted, after printing the event details on `stdout`.

Two events match if their class, name and type are same. They also match if one or more of the class, name and type is null. For example:

`node:atlas0:status` matches `node:atlas0:status`

`node::status` matches `node:atlas0:status`

`node::` matches `node:atlas0:status`

`:::` matches `node:atlas0:status`

Note that the class, name, type and description must all be specified when posting events but one or more of the class, name and type can be null when waiting on events.

8.2 Event Handling

Event handler scripts are specified in the `event_handlers` table. The default handlers installed by RMS are as follows:

```
rmsquery -v "select * from event_handlers"
name  class      type        timeout  handler
-------------------------------------------------------------------------------
      node       status      600      /usr/opt/rms/etc/rmsevent_node
                 temphigh    300      /usr/opt/rms/etc/rmsevent_env
                 tempwarn    300      /usr/opt/rms/etc/rmsevent_env
                 fan         300      /usr/opt/rms/etc/rmsevent_env
                 psu         300      /usr/opt/rms/etc/rmsevent_env
      event      escalation  -1       /usr/opt/rms/etc/rmsevent_escalate
      partition  status      600      /usr/opt/srasysman/bin/rmsevent_partition
```

The script `rmsevent_node` is run for all node status events. `rmsevent_env` is run for all environment events (temperature warnings, fan failures and PSU failures).

A timeout can be associated with each event handler. If the timeout is exceeded, the handler is killed and an *event escalation* event posted. `rmsevent_escalate` is run when one of the other handlers does not complete in the time allowed.

The `eventmgr` daemon runs on the `rmshost` node.

The handler scripts can send mail to users warning them of events. To enable this, set the `users-to-mail` attribute in the machine `attributes` table.

By default, event handler scripts are filed in `/usr/opt/rms/etc`. Local scripts should be filed in `/usr/local/rms` as the contents of the `bin` directory may change when a new release is installed.

Each of the scripts tests for the existence of both site-specific and OS-specific handler scripts before performing the default action.

Event handler scripts are called with five arguments: the event identifier, the class, name and type of the event and the event description. For example:

```
#!/bin/sh
#
# default OSF1 handler for node status events
#
# args:   id class name type description
#
```

```
program=‘basename $0‘
id=$1
class=$2
name=$3
type=$4
description=$5

#
# format event description message
#
message()
{
    echo "‘date ’+%h %e %X’‘ OSF1 event $id $type $class $name $description"
}

#
# log the event
#
message >> /var/rms/adm/log/event.log

#
# execute OSF1 specific handler
#
/usr/opt/srasysman/bin/checkout.exp -I -R -i $id -c $class -n $name -t $type -d $description
```

8.3 List of Events Generated

The following events are generated by RMS:

class = node type = status

The name field contains the name of the node. The description contains one of the following:

`unknown`	node status cannot be determined
`not responding`	node does not respond to `ping`
`active`	node is booted but RMS is not running
`running`	RMS is running

class = module type = temperature

The name field contains the name of the module. The description contains one of the following:

`ambient=`*`value`*	DS20, ES40, QM-S16, QM-S128

class = module type = temphigh

If the temperature exceeds the threshold value, the event type is `temphigh` and the description contains the above report and, in addition, the words `threshold exceeded`.

In the event of multiple failures, the reports are concatenated.

class = module type = psu

The name field contains the name of the module. The description field contains one of the following where `value` is a bitmap that identifies the PSUs:

`psu` *`value`* `failure`	QM-S128

In the event of multiple failures, the reports are concatenated.

class = module type = fan

The name field contains the name of the module. The description field contains one of the following where `value` is a bitmap that identifies the component (fan or PSU).

`enc fan` *`value`* `failure`	DS20, ES40, QM-S128

In the event of multiple failures, the reports are concatenated.

class = partition type = status

The name field contains the name of the partition. The description contains one of the following:

`running`	partition is running
`blocked`	partition is blocked
`closing`	partition is closing down
`down`	partition has been shut down

class = transaction type = status

The name contains the unique identifier for the transaction (the transaction handle) and the description contains one of the following:

`submitted`	transaction submitted
`started`	transaction being executed
`complete`	transaction completed successfully
`failed`	transaction failed to execute
`error`	transaction completed but there were errors

In the case of a transaction completing with errors (a link error test or boundary scan, for example), details of the failures are added to the transaction outputs table.

class = event type = escalation

The name contains the name of the event being escalated and the description contains `did not complete`. If a handler is registered, it is called with a description of the event that was not handled. The handler should pass the event to an external management agent.

class = server type = status

The description contains information on errors that occurred when starting the server.

8.3.1 Extending the RMS Event Handling Mechanism

The RMS event handling mechanism is open and extensible. New event types and handlers for them can be added on a site-by-site basis. For example, you might run a periodic file system status check on each node, execute a local cleanup script and post a file system event to RMS if the free space in the file system dropped below a prescribed level. The handler script could perform partition-wide or machine-wide cleanup and post a notification of the problem via email or an SNMP message.

9

Setting up RMS

9.1 Introduction

This chapter describes how to set up RMS and carry out routine operations. The information is organized as follows:

- Planning the installation (see Section 9.2).
- Starting RMS and configuring the system (see Section 9.3).
- Carrying out day-to-day operations and establishing backup and archive procedures (see Section 9.4).
- Customizing RMS (see Section 9.5).
- Dealing with log files (see Section 9.6).

9.2 Installation Planning

Before you install RMS, think about how the resources of the system will be used and who is going to use them. Ask yourself the following questions:

- Will the system be open to anyone to use or is it for a specific group of users?
- Will the machine run a constant workload or do you expect cyclical patterns in usage, for example, a prime shift versus evenings and weekends?

- Is the machine primarily for running parallel jobs or do you expect a significant workload from sequential jobs?
- Will some of your users have jobs that consume all of the resources of the system for extended periods of time? If so, are you happy for other users to wait until the machine is available or do they need access to resources of their own?
- How do you wish to process the accounting data?

The answers to these questions should help you to determine how to configure the system. It may be that you cannot answer these questions, in which case you should start with one of the basic configurations described below.

9.2.1 Node Names

While planning the machine installation give some thought to its name and the names of its nodes. We recommend selection of a short name for the machine (for example `atlas`). Node names should use the name of the machine as a base and their network port number as a suffix (for example `atlas0`, `atlas1`, `...`, `atlas63`). RMS will compress such lists of hostnames (for example `atlas[0-63]`).

Some machines have a management server node that is connected to the management network but is not connected to the Compaq AlphaServer SC Interconnect. By convention, this node is given the suffix `ms` (for example `atlasms`).

9.3 Setting up RMS

RMS should be installed according to the instructions in the *Compaq AlphaServer SC Installation Guide*.

For the purposes of this section, we assume a machine with 64 nodes, where each node has 4 CPUs, 4GB of memory and an 18GB disk. You should make adjustments for the actual number of nodes in your system. If RMS is already running on the machine, skip to Section 9.3.2.

9.3.1 Starting RMS

The RMS initialization script, `/sbin/init.d/rms`, is run on each node with the argument `start` as the node boots. Conversely, when the node halts, the script is run with the `stop` argument.

To start or stop RMS manually on all of the nodes at once, run `rmsctl` on the `rmshost` node with the appropriate argument (`start` or `stop`). This command runs `/sbin/init.d/rms` on each of the nodes in turn. `rsh` must be enabled for `root` users

for this command to work correctly. This should have been enabled as part of the installation.

```
# rmsctl start
```

Configure all of the nodes into the machine using `rcontrol`.

```
# rcontrol configure in 'atlas[0-63]'
```

Use `rinfo` with the `-n` option to check the status of the nodes. The output should show that all of the nodes are `running`.

```
# rinfo -n
running atlas[0-63], atlasms
```

If any of the nodes show a status other than `running`, restart them by running `/sbin/init.d/rms` on the nodes in question. For example, to restart RMS on `atlas3`, enter the following:

```
# /sbin/init.d/rms stop
# /sbin/init.d/rms start
```

If necessary, configure out any nodes that fail:

```
# rcontrol configure out atlas3
```

Restarting RMS

RMS daemons such as the Machine Manager and the Partition Manager can be stopped and started by executing the `/sbin/init.d/rms` script on the `rmshost` node. When run on the `rmshost` node, the init script checks the status of each of the partitions in the active configuration. If a partition is in the `running` state or `blocked` state, the partition is stopped and its `autostart` field in the `servers` table is set to 1, otherwise the field is set to 0. When the node boots, only those partitions that have their `autostart` field set to 1, are restarted. This means that the state of the configuration is preserved.

By contrast, if `rmsctl` is used to start and stop the machine, all of the partitions in the active configuration are started: when `rmsctl` stops RMS, it sets all of the `autostart` fields to 0; when it starts RMS, it sets them to 1.

9.3.2 Initial Setup with One Partition

This example describes the simplest possible setup. All nodes are in a single partition and there are no memory limits, time limits or access controls. Any user can run a job using all of the CPUs.

Once RMS is running on all of the nodes, you set up a single partition as follows:

```
# rcontrol create partition=parallel configuration=day nodes='atlas[0-63]'
# rcontrol start partition=parallel
```

You should now be able to run a parallel program across all 64 nodes, for example:

```
# prun -N64 hostname
...
# prun -N64 dping 0 32
...
```

9.3.3 Simple Day/Night Setup

In this example, the system is set up with two operating configurations: one called `day` and the other called `night`. During the day, the resources are split into two partitions: a login partition (called `login`) for program development and a parallel partition (called `parallel`) for execution of short parallel programs. At night, all of the nodes are assigned to a single partition (again called `parallel`) with a longer time limit for running parallel jobs.

Use the following commands to create this pair of configurations:

```
# rcontrol create partition=login configuration=day type=login nodes='atlas[0-7]'
# rcontrol create partition=parallel configuration=day type=parallel \
                  timelimit=600 nodes='atlas[8-63]'
# rcontrol create partition=parallel configuration=night type=parallel \
                  timelimit=3600 nodes='atlas[0-63]'
```

To start the `day` configuration, enter the following:

```
# rcontrol start configuration=day
...
```

To switch to the `night` configuration, use this command:

```
# rcontrol start configuration=night
...
```

Note that, after the switch, any jobs running on the `parallel` partition will continue to run as the `parallel` partition in the configuration `night` has more nodes. However, when changing back from `night` to `day`, you must decide what to do with any jobs that are running on nodes `'atlas[0-7]'`. The options are to wait for them to finish or to kill them. To wait for them to finish, stop the partition with the `wait` option.

```
# rcontrol stop partition=parallel option=wait
# rcontrol start configuration=day
...
```

Note

In the current release, any requests that are suspended when a partition is stopped must be resumed manually if the partition is restarted.

9.4 Day-to-Day Operation

Once the system is up and running, give some thought to automating some routine or day-to-day operations:

- Periodic shift changes
- Backing up the database
- Summarizing accounting data
- Archiving data
- Database maintenance

You may also want to configure nodes out of the system in the event of failures.

9.4.1 Periodic Shift Changes

The commands for switching between configurations are described in Section 9.3.3. When you are satisfied with the shift changes, install a `cron` job to perform them automatically.

9.4.2 Backing Up the Database

As soon as the RMS installation is stable, back up the database to a text file so that it can be recovered in the event of failure. Do this as follows, using `rmstbladm`, the table administration program (see Page 5-44).

```
$ rmstbladm -d > database_backup.txt
```

The backup file contains the sequence of SQL statements required to recover the current state of the database.

The RMS database is stored in `/var/rms/msqldb` on the `rmshost` node. The database server will exit if this file system fills up. RMS will not operate until sufficient space has been created in this file system. Ensure that there is at least 100MB free to allow for updates. The database server can be restarted using the script `/sbin/init.d/msqld`.

9.4.3 Summarizing Accounting Data

Accounting records accumulate in the RMS database as each job is run. By default, they are not processed as each site has its own requirements in this respect. A simple example script to produce a summary of resource usage is included in the release in `/usr/opt/rms/examples/scripts/accounting_summary`. See Appendix E (*Accounting Summary Script*) for a listing. The script produces the following output.

```
Accounting Summary of Machine atlas at 16:01 Wed 21 Feb 2001
Usage by Project/User For Previous Day

Project    User               CPU          User           Sys  Number
Name       Name              Secs          Secs          Secs Sessions
-------------------------------------------------------------------------
default    addy               596           533             6      8
           duncan              58            37             2      6
           johnt              540           227            51     15
           root             29272             2             8     37
           stephen            286            87           134     56
-------------------------------------------------------------------------
Total default               30751           885           201    122
-------------------------------------------------------------------------
Grand Total                 30751           885           201    122
-------------------------------------------------------------------------
```

When the accounts have been processed, the script can optionally delete the accounting records for resource requests that have completed.

This script (or one based on it) can be run nightly with a `cron` job, as shown in the following example.

```
0 0 * * * /usr/opt/rms/examples/accounting_summary
```

9.4.4 Archiving Data

To keep the database to a reasonable size, old entries should be removed on a regular basis as described in Section 9.4.5. Before clearing old entries from the database, archive any data you want to preserve. Generally, this is data from the following tables:

`resources`	Descriptions of each request to allocate resources
`jobs`	Descriptions of each job
`node_stats`	Utilization statistics for each node
`acctstats`	Accounting statistics logged by RMS

The data can be archived as a sequence of SQL statements using `rmstbladm`. The following example archives data from the node statistics (`node_stats`) table (see Section 10.2.15):

```
$ rmstbladm -d -t node_stats > nodestats.sql
```

Alternatively, you can execute a SQL query to extract the data, as follows:

```
$ rmsquery -v -u "select * from node_stats"
name    ctime   usercpu syscpu freemem ubc wired freeswap pages interrupts ...
-------------------------------------------------------------------------------
atlas0  973162911   0      1      295  483  162    5103      4        1    ...
atlas3  973162917   0      0      117   35   61    5103      4        1    ...
atlas2  973162922   0      0      124   29   62    5108      5        0    ...
atlasms 973162907  22     54       61  301   90     137     10      531    ...
atlasms 973163027  23     59       62  301   89     138      4      566    ...
```

Use the `ctime` field to select old data. For example, select data that was collected 2 days ago or more as follows:

```
now=‘rmsgettime‘
old=‘expr $now - 172800‘
rmsquery -v "select * from node_stats where ctime <= $old \
  order by ctime" > node_stats.sql
```

The following queries return data from the jobs, resources and accounting statistics tables. Accounting statistics can also be managed using the script described in Section 9.4.3.

```
rmsquery -v "select * from jobs where endTime <> 0 and \
 endTime < $old order by startTime" > jobs.dat

rmsquery -v "select * from resources where endTime <> 0 and \
  endTime < $old order by startTime" > resources.dat

rmsquery -v "select * from acctstats where running = 0 and \
  started < $old order by started" > acctstats.dat
```

After executing these queries, run `rmstbladm` to clean up the database as described in Section 9.4.5.

9.4.5 Database Maintenance

Certain tables in the RMS database grow over time or as jobs are submitted, in particular, the node statistics (`node_stats`) table, the `resources` table, the `events` table and the `jobs` table. These tables can be kept to a reasonable size by periodically

instructing the table administration program, `rmstbladm`, to remove old entries. Before running `rmstbladm`, archive any data you want to keep as described in Section 9.4.4. Remove old entries as follows:

```
# rmstbladm -c
```

`rmstbladm` clears out all entries that are older than a specified lifetime. The lifetime for job data and the lifetime for statistical data are specified in the `attributes` table (see Section 10.2.3).

Failure to clear old entries can cause problems as described at the end of this section. See Section 9.4.3 for details about the accounting statistics table which also grows over time.

A `cron` job can be set up to clear out the tables. In the following example, this task is performed at 2 a.m. each weekday morning.

```
0 2 * * 1-5 /usr/bin/rmstbladm -c
```

Troubleshooting

If the tables are not cleared out on a regular basis, the database continues to grow until the performance of RMS is affected. Indications that this is happening include the following:

- The database server, `msqld`, uses more memory.
- The table join operations performed by `rinfo` take longer.
- Queries acting on large tables may exceed normal user memory limits.
- `rmstbladm` takes a long time to clear out old entries or may fail, although insert operations succeed and the tables continue to grow.

The point at which memory limits are exceeded varies with the number of nodes in the machine and the amount of memory on the `rmshost` node. To check that the size of the database is within operating limits, enter the following query:

```
$ rmsquery; "select * from node_stats" > /tmp/stats.sql
```

If this fails, follow these steps to recover from the problem:

1. Log in to the `rmshost` node as `root` and stop the database server, as follows:

   ```
   # /sbin/init.d/msqld stop
   MSQL: service stopped
   ```

2. Change to the directory that contains the database, for example:

```
# cd /var/rms/msqldb/rms_atlas
```

   Delete the following files: `node_stats.dat`, `node_stats.def`, `node_stats.idx` and `node_stats.ofl`.

```
# rm node_stats.*
```

3. Restart the database server, as follows:

```
# /sbin/init.d/msqld start
MSQL: daemon started
```

4. Create a new node statistics table, as follows:

```
# rmstbladm -u
```

After this, `rmstbladm` should succeed in cleaning out old entries.

9.4.6 Configuring Nodes Out

If a node fails and cannot be rebooted, it must be configured out while it is being repaired. The procedure for this is as follows:

1. Stop the partition containing the failed node. Any jobs that are running on the failed node when the partition is stopped will be killed. Other jobs will continue to run.

```
# rcontrol stop partition=parallel
```

2. Configure out the node (`atlas2` in this case). Note that RMS reports an error if you try to configure a node in or out while the partition is running.

```
# rcontrol configure out node=atlas2
```

3. Restart the partition:

```
# rcontrol start partition=parallel
```

After this procedure, the partition runs without the node. This reduces temporarily the maximum size of job that can run.

When the node has been repaired, stop the partition again and configure the node back in as follows:

1. Stop the partition containing the failed node:

```
# rcontrol stop partition=parallel
```

2. Configure in the repaired node (`atlas2` in this case):

```
# rcontrol configure in node=atlas2
```

3. Restart the partition:

```
# rcontrol start partition=parallel
```

This brings the partition back up to its full complement of nodes.

9.5 Local Customization of RMS

RMS can be customized to suit local operating patterns in a variety of ways. Customization is done through site-specific scripts in `/usr/local/rms/etc`. The following site-specific customizations are supported:

- Core file analysis
- Partition startup
- Event handling
- Switch manager configuration

If site-specific scripts exist then they override the defaults supplied with RMS.

9.5.1 Partition Startup

The default partition startup script enables or disables logging in to a node according to the partition type. Site-specific variants might check whether users are logged in to the node and warn them of changes. They might also check on the availability of space in local temporary file systems.

To create a site-specific partition startup script, copy the default script `/opt/rms/etc/pstartup` to `/usr/local/rms/etc` and modify it as required.

9.5.2 Core File Handling

By default, RMS instructs the operating system to dump core files to local temporary file space under `/local/core/rms`. Change the attribute `local-corepath` in the `attributes` table to select an alternative default directory for core files. Subdirectories are created in *`local_corepath/resource-id`* for each resource request. Change the attribute `rms-keep-core` to disable the dumping of core files.

If dumping is disabled, a core file analysis script is run on at least one node before the core files are deleted. The default script prints a backtrace showing why the program

crashed. A site-specific variant might copy core files from the local temporary directory to a global file system for subsequent analysis.

To create a site-specific core file analysis script, copy the default script `/opt/rms/etc/core_analysis` to `/usr/local/rms/etc` and modify it as required.

9.5.3 Event Handling

The default event handlers check for the existence of a site-specific handler of the same name in `/usr/local/rms/etc`. If such a script exists, it will be executed in preference to the default handler. To make site-specific changes, copy the default scripts to this directory and amend them to your needs. Use `rmspost` to test their correct operation.

9.5.4 Switch Manager Configuration

The switch network manager (`swmgr`) must be run on the node to which the switch network control cable is connected. By default, this is the `rmshost` node. Depending on the configuration of your system, you may need to change this default.

The `swmgr` process consumes CPU time while sampling the network for errors. Therefore, it should ideally be run on a lightly loaded node that is not used to run parallel jobs; for example, a management server. Use `rcontrol` to stop the running `swmgr`, and run `rmsquery` to set the node that should run the `swmgr`, as follows:

```
# rcontrol stop server=swmgr
# rmsquery "update servers set hostname='atlasms' where name = 'swmgr'"
# rmsquery "select name,hostname from servers where name = 'swmgr'"
swmgr  atlasms
# rcontrol start server=swmgr
```

If your system does not have a suitable management server, you should run the `swmgr` on the `rmshost` node. If `rmshost` is an alias for one node of a resilient pair, the `swmgr` should run on the primary node. Under these circumstances, you should set the rate at which the `swmgr` polls so as to reduce the impact on other processes, by changing the polling interval from the default value (30 seconds) to 15 minutes. Use `rcontrol` to do this, as follows:

```
# rcontrol create attribute name=swmgr-poll-intervalval=900
```

The change in polling frequency will take effect next time the `swmgr` is started. To force the change to occur immediately, use `rcontrol` to stop and start the server, as described above.

9.6 Log Files

The RMS daemons output reports to log files in the directory `/var/rms/adm/log`. The amount of detail is controlled for each daemon by setting a reporting level. By default, the reporting level is set to 0.

The reporting level is a bitmap that turns on different reports. Values for the reports are as follows:

Symbolic Name	Value	Description
`INIT_DEBUGGING`	1	Server initialization and shutdown messages
`REQ_DEBUGGING`	2	Requests made to servers
`JOB_DEBUGGING`	8	Job startup and change of state
`RESOURCE_DEBUGGING`	32	Resource allocation and change of state
`EDIT_DEBUGGING`	64	SQL queries
`MALLOC_DEBUGGING`	256	Monitor server memory allocation

The level of reporting can be controlled in three ways.

1. On an individual user basis, by setting the environment variable `RMS_DEBUG`.
2. Using `rcontrol` to reload the daemon with a specified debug value. For example, the following command reloads the Machine Manager with a reporting level of 1:

   ```
   # rcontrol reload server=mmanager debug=1
   ```

   The following example reloads the Partition Manager for the `par1` partition with a reporting level of 41 (initialisation, job and resource information)

   ```
   # rcontrol reload partition=par1 debug=41
   ```

3. Using `rmsquery` to set the `args` field of the daemon's entry in the `servers` table (see Section 10.2.19) to `-r` *`value`*, where *`value`* is the required reporting level.

   The following example gives the Partition Manager for the `par1` partition a reporting level of 33.

   ```
   # rmsquery "update servers set args='-r 33' where name='pmanager-par1'"
   ```

   Then restart the Partition Manager. This change remains in place each time the partition is restarted. The output files in `/var/rms/adm/log` can grow in size rapidly when debug options are enabled. Take care not to fill the file system.

10

The RMS Database

10.1 Introduction

This chapter describes the tables which make up the RMS database. Each machine has its own database, called `rms_`*`machine`*, where *`machine`* is the name of the machine. This allows a single database server to support multiple machines.

The database contains tables storing information on the following:

- The configuration of the machine: its nodes and the Compaq AlphaServer SC Interconnect
- The users of the machine: the access controls and resource quotas applied to them; their requests to run jobs; the accounting records for these jobs
- The operation of the machine, including its current state and performance statistics

10.1.1 General Information about the Tables

- All of the field names in the database are case sensitive.
- Fields are of these types:

`char(`*`length`*`)`	This denotes a character string of the specified *`length`*.
`int`	This denotes an integer value.
`%`	This denotes a percentage value stored in an integer field.
`UTC`	This denotes a UTC time value stored in an integer field.

`x-y`	This denotes a range of possible integer values.
`text`	This denotes a character string of arbitrary length.

- Fields of type `text` can be selected by the field name but the text entry cannot be matched.

  If the text is a list of items, for example, a list of node names, the items in the list may be separated by white space. A list of names, all of which share a common base, for example, `atlas0 atlas1 atlas2`, may also be represented by a `glob`-like expression, in this example, `atlas[0-2]`. The normal `glob(7)` expression syntax is relaxed to include multiple digit numbers. For example, `atlas[0-10,14]` represents the nodes numbered from `atlas0` to `atlas10` inclusive plus `atlas14`.

- Information on time is stored as UTC time in integer fields. Client programs should convert time to local time and output the result as a string.

10.1.2 Access to the Database

There are three levels of access to the database:

1. Users can extract information from all of the tables but cannot update them.
2. Operators and administrators can extract information from all of the tables and, in addition, update a limited selection of fields in some tables.
3. RMS itself can extract and update information in all fields of all tables. The description of the tables in Section 10.2 includes information about which RMS programs create and update each field.

10.1.3 Categories of Table

This chapter describes the tables in the database, listing them in alphabetical order. This section groups the tables by category.

Configuration of Nodes

The following tables contain information about the individual nodes and about the machine as a whole.

nodes	describes the attributes of each node
node statistics	contains performance statistics for each node
partitions	defines each partition and its scheduling parameters
modules	physical location and environmental data for each module
module types	describes the characteristics of each supported module

Operational State

The following tables hold details of the current state of the machine.

events	records changes to the state of the machine
event handlers	lists the handlers used to act on events
attributes	holds site-specific attribute-value pairs
fields	specifies how objects and attributes may be modified
servers	holds details on each daemon (hostname, port number, pid)
transactions	records requests to change the machine configuration
software products	describes the components of each software product
installed components	describes the components installed on each node

User Details

The following tables contain information about the users of the RMS: their privileges and priorities and their usage of the system.

users	lists the projects to which users belong
projects	lists the projects
access controls	describes limits on user access to resources
resources	describes the allocation of resources to users
jobs	describes the users' jobs
accounting statistics	contains an accounting record for each resource

Configuration of the Network

The following tables describe the network components. Definitions of terms used in describing the Compaq AlphaServer SC Interconnect can be found in Appendix A (*Compaq AlphaServer SC Interconnect Terms*).

elans	records the position and state of the Elan network adapters
elites	records the position and state of the Elite switches
switch boards	records the position and state of each switch board
link errors	logs network errors

Internal Tables

The RMS database includes a number of tables that are mainly used internally. These are noted in this chapter as being of internal use but are not described in any detail.

transaction outputs	contains output from requests posted to the transaction log
request types	describes output formats in the transaction outputs table
statistics	lists the performance statistics available in the current release
services	describes the services available and who can use them

10.2 Listing of Tables

This section lists the tables in alphabetical order.

10.2.1 The Access Controls Table

The `access_controls` table shown in Table 10.1, contains access control and usage limit descriptions for users and projects.

Table 10.1: Access Controls Table

Field	Type	Description
`name`	char(16)	name of the user or project
`class`	char(8)	class of control: `user` or `project`
`partition`	char(16)	partition to which access controls apply
`priority`	int	default scheduling priority
`maxcpus`	int	maximum number of CPUs
`memlimit`	int	maximum memory per CPU in megabytes

An entry for the reserved partition name `default` specifies the `priority`, `maxcpus` and `maxmem` that should apply for any partition names not explicitly specified.

The `priority` field stores the default scheduling priority of jobs submitted by a user or project. The higher the value, the more likely the job is to run. Priority values range from 0 to 100, the default being 50.

The `maxcpus` field stores the maximum number of CPUs that a user or project may have allocated at once. Requests for more than this number of CPUs fail. Once this number of CPUs is allocated, additional requests block until some CPUs are freed.

The `memlimit` field stores the default memory limit per CPU for jobs submitted by the named user or project.

10.2.2 The Accounting Statistics Table

Each time CPUs are allocated to a request, a record is created in the accounting statistics (`acctstats`) table shown in Table 10.2. Records are updated periodically and

at the end of each job by the Partition Manager, `pmanager` (see Section 4.4).

Table 10.2: Accounting Statistics Table

Field	Type	Description
`name`	char(16)	name of the allocated resource
`uid`	int	user ID
`project`	char(16)	project name
`started`	UTC	time when CPUs were allocated
`etime`	int	elapsed time in seconds
`atime`	int	CPU seconds allocated
`utime`	int	user time in seconds
`stime`	int	system time in seconds
`cpus`	int	number of CPUs allocated
`mem`	int	maximum memory extent in megabytes
`pageflts`	int	number of page faults requiring I/O
`memint`	int	memory integral in megabyte hours
`running`	`0 | 1`	CPUs allocated and running jobs

The `etime` field stores the elapsed time (in seconds) since CPUs were first allocated to the resource, including any time during which the resource was suspended.

If a partition is stopped, while a job is running, and the partition is restarted before the job completes, the `etime` field will correctly show the total elapsed time of the running job including the time when the partition was down. If a partition is stopped, while a job is running, and the job completes before the partition is restarted, the `acctstats` table entry will reflect only the time when the partition was running. Any additional time that the job was running while the partition was down is not included in the `acctstats` table entry. If a job is terminated because it exceeds its timelimit or a job is terminated with `rcontrol`, the `etime` field reflects the time for which CPUs were allocated.

The `atime` field stores the total elapsed time (in seconds) that CPUs have been allocated – this excludes time during which the resource was suspended, but includes any time when the partition was down while jobs were running. The value stored is the total for all CPUs used by the resource.

The `utime` and `stime` fields are summed over all processes in the program.

The `running` field is set to `1` while CPUs are allocated. It is set to `0` when the CPUs are deallocated, at which point no further updates take place.

The `pageflts` field shows the number of page faults requiring I/O summed over all of the processes in the parallel program. It is normally 0. A non-zero and growing value indicates that the program is paging on some or all nodes.

The memint field is set to 0 in AlphaServer SC Version 2.0.

The number of entries in the accounting statistics table can grow rapidly. The table should be cleared periodically of old entries as described in Section 9.4.3.

10.2.3 The Attributes Table

The attributes table shown in Table 10.3, stores information specific to the site or the release. This information is stored as attribute-value pairs. The table is created by the table administration program, rmstbladm (see Page 5-44), which adds a minimal set of default entries. Further attributes are added by RMS daemons. The values can be modified by the administrator.

The entries in the attributes table can be grouped into four sections:

1. Machine attributes
2. Performance statistics attributes
3. Server attributes
4. Parallel processing attributes

The machine attributes in the following table are supported:

Table 10.3: Machine Attributes

Attribute	Default	Description
network-type	QM-S16	data network type (QM-S16 or QM-S128)
network-levels	2	number of levels of switch network
network-layers	1	number of layers (rails) of switch network
racks	4	number of 19" racks in the machine
units-per-rack	40	height of a 19" rack in units
rmshost		node running the RMS daemons

The performance statistics attributes shown in Table 10.4 control the collection and lifetime of performance statistics. The statistics are collected by rmsd at the intervals given in this table. In the current release, only CPU statistics are gathered.

The number of entries in the jobs table, the resources table, the accounting statistics table, the events table and the node statistics table can grow rapidly, especially on a large busy machine or if the value of cpu-stats-poll-interval is very small. The lifetime entries in Table 10.4 assign a finite life to this data. Once this lifetime has been reached, the RMS table administration program, rmstbladm, will clean out old

entries, if called with the `-c` option (see Page 5-44). Note that the accounting statistics table is not cleared out (see Section 10.2.2).

Table 10.4: Performance Statistics Attributes

Attribute	Default	Description
`node-statistics`	`cpu`	statistics collected per node
`cpu-stats-poll-interval`	`120`	time in seconds between CPU samples
`data-lifetime`	`48`	time in hours to keep job data
`stats-lifetime`	`24`	time in hours to keep statistical data

The server attributes in Table 10.5 control the behavior of the RMS daemons. All of the modification times are in UTC. Client applications should convert this to local time and print it as a string.

If the attribute `node-status-poll-interval` is not set or set to zero, the value of `rms-poll-interval` is used instead.

Table 10.5: Server Attributes

Attribute	Default	Description
`rms-poll-interval`	`60`	polling interval for RMS daemons
`node-status-poll-interval`	0	time between monitoring node status
`status-modify-time`		last time the status changed
`resource-modify-time`		last time a resource was modified
`version`		RMS version number
`altzone`	0	shift in seconds to apply to UTC time to get local time on `rmshost`

The attributes in Table 10.6 control the scheduling of parallel programs. If the number of resource requests reaches `pmanager-queuedepth`, subsequent requests either block or fail immediately (if the immediate option to `prun` has been selected). The blocked requests do not appear in the database. If the `pmanager-idletimeout` is exceeded, the resource times out with an exit status of 125.

Table 10.6: Scheduling Attributes

Attribute	Default	Description
default-partition	parallel	the partition used by default for parallel programs
default-priority	50	the default scheduling priority
grace-period	60	the time allowed in seconds for a parallel program to exit after a CPU time signal
lbal-partition	login	the default partition for load balancing requests
exit-timeout		default exit timeout (absent by default)
pmanager-queuedepth	0	maximum number of queued requests
pmanager-idletimeout	0	number of seconds an allocated resource may remain idle
rms-keep-core	1	keep (1) or remove (0) core files
local-corepath	/local/core/rms	directory path for core files

10.2.4 The Elans Table

The elans table shown in Table 10.7, contains one entry for each Elan network adapter connected to the Compaq AlphaServer SC Interconnect. Entries are created and maintained by the rmsd running on the node containing the Elan.

Table 10.7: Elans Table

Field	Type	Description
name	char(8)	unique identifier for the adapter
hostname	char(16)	name of node containing the adapter
layer	0–31	layer (or rail) number
netid	int	network address within the layer
revision	int	chip revision level
ecount	int	error count for the last sample
ecount10	int	error count for the last 10 samples
status	char(8)	Elan status (ok, unknown, error)
linkstate	char(2)	state of the link
linkerrors	text	description of errors in the last 10 samples

Entries in the linkerrors field give the ID of the link and then, in brackets, a vector of error counts (see Appendix A (*Compaq AlphaServer SC Interconnect Terms*)).

10.2.5 The Elites Table

The `elites` table shown in Table 10.8, contains one entry for each switch in the network. Its entries are created and maintained by the Switch Network Manager, `swmgr` (see Section 4.5).

Table 10.8: Elites Table

Field	Type	Description
`name`	char(8)	Elite name, a unique ID for each switch
`layer`	`0–31`	layer (or rail) number
`level`	`0–3`	level number
`netid`	`0–255`	network address within the layer
`plane`	`0–63`	plane number
`board`	char(8)	name of the board containing the switch
`chip`	`0–7`	chip number on the board
`revision`	int	chip revision number
`ecount`	int	error count at the last sample
`ecount10`	int	error count for the last 10 samples
`status`	char(8)	Elite status (`ok`, `unknown`, `error`)
`linkstate`	char(8)	state of each link
`linkerrors`	text	description of errors in the last 10 samples

The `linkstate` field contains a character for each of the 8 links. Each link can be in one of the states shown in Table B.2.

Entries in the `linkerrors` field give the ID of the link and then, in brackets, a vector of counts for each of the supported error types (for more details on Compaq AlphaServer SC Interconnect terms see Appendix A (*Compaq AlphaServer SC Interconnect Terms*)).

10.2.6 The Events Table

Entries are added to the `events` table shown in Table 10.9, each time an object managed by the RMS changes state, for example, when a node status changes, a partition starts or a component fails.

Table 10.9: Events Table

Field	Type	Description
`id`	int	unique identifier for each event
`name`	char(16)	name of object that has changed state

(continued on next page)

Table 10.9: Events Table (cont.)

Field	Type	Description
class	char(16)	class of the object, such as node or partition
type	char(16)	type of event
ctime	UTC	time at which the event occurred
handled	0 | 1	whether the event has been handled or not
description	text	description of the event

Table 10.10 shows three typical events. The first shows a node status change as RMS starts on node cfs1, the second shows a temperature change on module mod2 and the third shows the partition parallel starting.

Table 10.10: Example of Status Changes

name	class	ctime	type	description
cfs1	node	893427468	status	running
mod2	module	894991521	temperature	ambient=15
parallel	partition	894991490	status	running

The events table can grow rapidly. Running the table administration program, rmstbladm, with the -c option removes old entries. This should be done periodically using a cron script. See Page 5-44 for details. The data-lifetime attribute in the attributes table (see Section 10.2.3) determines how old the entries must be before they are removed.

Events are discussed in detail in Chapter 8 (*Event Handling*).

10.2.7 The Event Handlers Table

The event_handlers shown in Table 10.11, defines the handler scripts that are run in response to events. Event handling is discussed in detail in Chapter 8 (*Event Handling*).

Table 10.11: Event Handlers Table

Field	Type	Description
name	char(16)	name of object that has changed state
class	char(16)	class of the object, such as node or partition
type	char(16)	type of event
timeout	int	time limit in seconds for the handler to complete

(continued on next page)

Table 10.11: Event Handlers Table (cont.)

Field	Type	Description
handler	char(32)	handler script to run

10.2.8 The Fields Table

The fields table shown in Table 10.12, defines which RMS objects and attributes can be created and modified using rcontrol (see Page 5-20), identifying them by a table name and field name within that table.

Table 10.12: Fields Table

Field	Type	Description
name	char(16)	name of the field
tablename	char(16)	name of the table
access	char(8)	currently unused; always set to admin
type	char(16)	defines the type of value
rangemin	int	minimum value
rangemax	int	maximum value
textattr	text	specifies how values are validated

The value of the type field determines how rcontrol checks the validity of values entered by an administrator. The type field may hold one of the values shown in Table 10.13.

Table 10.13: Type Values

Value	Description
null	no checking
selectedtext	textattr gives a comma-separated list of valid values
integer	entry must be in range bounded by min and max
relation	textattr gives a tablename.fieldname pair; entry must be a value of fieldname in tablename

Values in the attributes table are not checked using this method; the valid values for attributes are built into rcontrol.

10.2.9 The Installed Components Table

The `installed_components` shown in Table 10.14, contains information about software components installed on each node.

Table 10.14: Installed Components Table

Field	Type	Description
`hostname`	char(16)	hostname of the node on which the component is installed
`product`	char(16)	name of the software product to which the component belongs
`prodversion`	char(16)	version of the software product to which the component belongs
`component`	char(16)	name of the component
`compversion`	char(32)	version of the component
`ctime`	UTC	time the component was installed

10.2.10 The Jobs Table

The `jobs` table shown in Table 10.15, contains one entry for each parallel job. The entries are maintained by the Partition Manager, `pmanager` (see Section 4.4).

The `jobs` table can grow rapidly. Running the table administration program, `rmstbladm`, with the `-c` option removes old entries. This should be done periodically using a `cron` script. See Page 5-44 for details. The `data-lifetime` attribute in the `attributes` table (see Section 10.2.3) determines how old the entries must be before they are removed.

Table 10.15: Jobs Table

Field	Type	Description
`name`	char(16)	unique identifier for each job
`resource`	char(16)	the name of the resource on which the job is running
`status`	char(16)	status of the job
`cpus`	text	list of CPUs allocated to this job
`nodes`	text	list of nodes allocated to this job
`hostnames`	text	list of hostnames allocated to this job
`startTime`	UTC	time the job started
`endTime`	UTC	time the job completed
`contexts`	char(16)	range of Elan contexts allocated to the job

(continued on next page)

Table 10.15: Jobs Table (cont.)

Field	Type	Description
exitStatus	int	exit status of the job
session	int	UNIX session ID of the allocating process
cmd	text	command being executed

Job names are sequence numbers generated automatically. The status field holds one of the values shown in Table B.1.

While the job is running, endTime is set to the time by which the job must end, assuming there is a timelimit on the partition. If there is no time limit, the endTime is set to 0. Finally, endTime is updated to show the time the job completed.

The nodes and cpus fields contain lists of node and CPU numbers in use by a job. Each pair of values defines a cpus x nodes box allocated to the job. The total number of CPUs allocated is the sum of the area of these boxes. See also Section 2.4.2.

A command name, cmd, passed to prun, may be up to MAXPATHLEN in length. In the jobs table, the command name is truncated to 32 characters, including three dots (...) appended to the name to show that it has been truncated.

10.2.11 The Link Errors Table

The link_errors shown in Table 10.16, contains one entry for each link error detected by the Switch Network Manager, swmgr (see Section 4.5).

Table 10.16: Link Errors Table

Field	Type	Description
id	int	unique identifier for each error
name	char(16)	name of the chip detecting the fault
class	char(16)	type of chip detecting the fault (elan, elite)
ctime	UTC	time at which the error was detected
description	text	description of the error

Entries in this table are updated by the swmgr. High error counts in the description field indicate that an error is persistent. Increasing counts indicate that it is current.

Entries in the description field give the ID of the link and then, in brackets, a vector of counts for each of the supported error types. Compaq AlphaServer SC Interconnect link errors are described in Appendix A (*Compaq AlphaServer SC Interconnect Terms*).

10.2.12 The Modules Table

The `modules` table shown in Table 10.17, contains descriptions of each hardware module in a machine. The modules may be nodes, network components or storage devices. It is created by `rmsbuild`. Entries are added and removed by `rcontrol` and updated by `rmsd` and the Switch Network Manager, `swmgr`.

Table 10.17: Modules Table

Field	Type	Description
`name`	char(16)	name of the module
`type`	char(16)	type of the module, from the module types table
`class`	char(16)	class of module (`node`, `network`)
`rack`	int	ID of the rack that contains the module
`unit`	int	location of the module in the rack
`psus`	int	bitmap of the functioning power supply units
`fans`	int	bitmap of the functioning fans
`estatus`	char(16)	environmental status of the module
`environment`	text	description of the environmental status

Valid values for the `type` field are listed in the modules type table (see Section 10.2.13).

The `psus` and `fans` fields are bitmaps; their width is controlled by the corresponding values in the module types table (see Section 10.2.13).

`rmsd` collects environmental data from the kernel on each node. The operational status of the cooling fans and power supplies is logged along with the temperature status of vital system components. This is used to generate an environment status, `estatus`, and an environment string, `environment`. The environment status can take one of the values shown in Table B.3.

If the environment status is recorded as `ok`, the environment string contains temperature readings from the CPU, power supply unit and enclosure. If a node has more than one instance of each type of temperature sensor, the maximum of their values is recorded.

Temperature information is recorded as a list of attribute-value pairs, for example:

```
ambient=15 cpu=40 psu=20
```

If an error occurs, the environment string contains details of what has failed, for example, the following string indicates that the CPU fan number 1 has failed on the node.

```
cpu fan 1 failure
```

10.2.13 The Module Types Table

The `module_types` table shown in Table 10.18, contains descriptions of each of the module types supported in a given release of the RMS. It is updated by the table administration program, `rmstbladm` (see Page 5-44), when a new release is installed.

Table 10.18: Module Types Table

Field	Type	Description
`name`	char(16)	name of the module type
`class`	char(16)	class of module (`node`, `network`, `storage`)
`units`	int	height of the module in units
`cpus`	int	number of CPUs in the module
`psus`	int	number of PSUs in the module
`fans`	int	number of fans in the module
`thermistors`	int	number of thermistors in the module
`description`	text	description of the module

The module types supported in the current release are shown in Table 10.19.

Table 10.19: Valid Module Types

Name	Class	Units	CPUs	PSUs	Fans	Therm	Description
`DS20`	`node`	12	2	1	1	1	`AlphaServer DS20`
`ES40`	`node`	8	4	3	6	7	`AlphaServer ES40`
`QM-S16`	`network`	4	0	1	0	1	`QM-S16 switch network`
`QM-S128`	`network`	16	0	3	6	24	`QM-S128 switch net-work`

See Appendix A (*Compaq AlphaServer SC Interconnect Terms*) for more details of the network modules.

10.2.14 The Nodes Table

The `nodes` table shown in Table 10.20, contains configuration information on each node in a machine. The entries are created by the RMS clients `rmsbuild` and `rcontrol`. Fields are updated by `rmsd` when the node is booted or RMS is restarted and by the

Machine Manager, mmanager, when the node's status or run level changes.

Table 10.20: Nodes Table

Field	Type	Description
name	char(16)	the name of the node
type	char(8)	node type, such as ES40
maxmem	int	maximum memory available in megabytes
maxfree	int	maximum free memory available in megabytes
maxswap	int	maximum swap space available in megabytes
maxtmp	int	temporary file system space in megabytes
cpus	1–32	number of CPUs
cpus_reserved	0–32	number of CPUs reserved for OS services
elans	int	mask of Elan devices present
netid	0–255	physical network ID (if applicable)
configured	0 | 1	whether node is configured in or out
status	char(16)	current node status
runlevel	char(16)	UNIX run level
boot_time	UTC	time when node was last booted
swap_eager	0 | 1	swap allocation is lazy(0) or eager(1)
console	char(32)	command line to connect to console

The type field takes a value from the module types table (see Section 10.2.13).

The cpus_reserved field specifies the number of CPUs that are not available for running parallel programs. These CPUs are reserved for running system services.

The configured field indicates whether a node is active (1) or configured out for repair or upgrade (0).

The status field indicates the service level being provided by a node. Valid values are shown in Table B.4. State changes are logged in the events table (see Section 10.2.6); entries are keyed by class=node.

The runlevel may have one of the values shown in Table B.5.

The elans field is a mask of the Elan devices present in the node. It has one bit set for each device. In previous releases, only device 0 was supported.

10.2.15 The Node Statistics Table

The node statistics (node_stats) table shown in Table 10.21, contains performance statistics collected periodically by the rmsd daemon running on each node.

To enable the collection of these statistics, the node-statistics field in the

`attributes` table (see Section 10.2.3) must be set to `cpu`. This is the default setting. The interval at which the nodes are sampled for CPU statistics is controlled by the attribute `cpu-stats-poll-interval` in the `attributes` table; the default is to sample every 2 minutes.

The node statistics (`node_stats`) table can grow rapidly, especially on a large machine. Running the table administration program, `rmstbladm`, with the `-c` option removes old entries. This should be done periodically using a `cron` script. See Page 5-44 for details. The `stats-lifetime` attribute in the `attributes` table (see Section 10.2.3) determines how old the entries must be before they are removed.

Table 10.21: Node Statistics Table

Name	Type	Description
`name`	char(16)	name of the node
`ctime`	UTC	time at which sample was collected
`usercpu`	%	user CPU time since last sample
`syscpu`	%	system CPU time since last sample
`freemem`	int	free memory in megabytes
`ubc`	int	size of the unified buffer cache in megabytes
`wired`	int	wired memory in megabytes
`freeswap`	int	free swap space in megabytes
`pages`	int	page fault rate
`interrupts`	int	interrupts rate (except clock)
`syscalls`	int	system call rate
`users`	int	number of users logged in
`freetmp`	int	free temporary file space in megabytes

The `usercpu` and `syscpu` statistics are percentages calculated over the period since the last sample.

The `interrupts`, `pages` and `syscalls` statistics are rates averaged over the interval since the last sample.

10.2.16 The Partitions Table

The nodes in a machine are grouped into partitions according to their function. For example, there may be an administrative partition, a login partition and a parallel programming partition. A set of partitions spanning the machine is called a *configuration*. Different configurations may be appropriate to different times of the day or week. For example, one for daytime running and another for nights and weekends. Only one configuration, the *active configuration*, can be running at a time.

The `partitions` table shown in Table 10.22, describes how nodes are allocated to partitions in each of the configurations. It also contains scheduling parameters (see also Section 7.3) for each partition.

The entries in the `partitions` table are created by `rcontrol`. The information is updated by the Partition Manager, `pmanager`, as it starts.

Table 10.22: Partitions Table

Field	Type	Description
`name`	char(16)	name of the partition, such as `par`
`configuration`	char(16)	name of the configuration, such as `day`
`nodes`	text	list of nodes in the partition
`configured_nodes`	text	list of nodes configured into the partition
`cpus`	int	number of CPUs configured in
`freecpus`	int	number of free CPUs
`active`	`0 | 1`	whether partition is active (`1`) or not (`0`)
`startTime`	UTC	time partition was last started
`status`	char(16)	status of the partition
`timelimit`	int	time limit in seconds for a parallel job
`type`	char(16)	partition type (`parallel`, `login`, `general`, `batch`)
`timeslice`	int	time slice interval in seconds
`mincpus`	int	minimum number of CPUs that can be allocated
`memlimit`	int	default memory limit in megabytes

Partition names do not have to be unique but the combination of a partition and a configuration name must be unique. For example, there may be a partition named `login` in two different configurations, each with a different set of values in the `partitions` table.

Valid values for the `status` of the partition are shown in Table B.6.

The `type` field controls how jobs are scheduled on the partition (see also Section 7.2). If the partition type is `parallel` then it is exclusively for gang-scheduled parallel programs. Partitions of type `login` support interactive logins and load-balanced sequential program execution. Partitions of type `general` support login shells, load-balanced sequential program execution and parallel programs. Partitions of type `batch` are under the exclusive control of a batch system. The batch system can use them for sequential or parallel jobs but interactive use is prohibited.

The `freecpus` field stores the count of the number of CPUs available in the partition. It is updated by the `pmanager` each time CPUs are allocated or freed.

The `configured_nodes` field stores the subset of nodes that were configured in when the partition was started.

The `timeslice` field stores the interval in seconds between periodic rescheduling of parallel jobs. Time slicing is disabled when this field is null, the default.

The `timelimit` field stores the maximum interval in seconds for which CPUs in a partition may remain allocated. Time limits are disabled when this field is null, the default.

The `memlimit` field stores the default memory limit in megabytes per CPU for jobs running on this partition.

10.2.17 The Projects Table

The `projects` table shown in Table 10.23, lists all of the projects that have been defined. A project is a list of users. Membership of the project is specified in the `projects` field of the `users` table (see Section 10.2.24). All accounting records include the project to which a user's job is being billed (see Table 10.2).

Table 10.23: Projects Table

Field	Type	Description
`name`	char(16)	project name
`description`	text	label describing the project

10.2.18 The Resources Table

The `resources` table shown in Table 10.24, contains one entry for each current resource request. The entries in this table are maintained by the Partition Manager, `pmanager` (see Section 4.4).

The `resources` table can grow rapidly. Running the table administration program, `rmstbladm`, with the `-c` option removes old entries. This should be done periodically using a `cron` script. See Page 5-44 for details. The `data-lifetime` attribute in the `attributes` table (see Section 10.2.3) determines how old the entries must be before they are removed.

Table 10.24: Resources Tables

Field	Type	Description
`name`	char(16)	resource name
`partition`	char(16)	partition name

(continued on next page)

Table 10.24: Resources Tables (cont.)

Field	Type	Description
username	char(16)	name of the user
hostnames	text	list of hostnames allocated
status	char(16)	status of the resource
cpus	text	list of CPUs allocated
nodes	text	list of nodes allocated
startTime	UTC	time resources were allocated
endTime	UTC	time resources were deallocated
priority	int	current priority of the request
flags	int	scheduler flags for the resource
ncpus	int	number of cpus allocated
batchId	int	batch system identifier for the request
memlimit	int	memory allocated per CPU in megabytes
project	char(16)	name of the project associated with the resource
pid	int	pid of the allocating process (prun or allocate).

Resource names are sequence numbers generated automatically.

The hostnames field lists the names of the nodes allocated to this request.

Valid values for the status field are given in Table B.7.

The cpus and nodes fields contain lists of CPU and node numbers in use by a job. Each pair of values defines a cpus x nodes box allocated to the job. The total number of CPUs allocated is the sum of the area of these boxes.

The batchid field contains the batch system identifier for this request. If the request was made by LSF then the field contains LSB_JOBID. If the request was made by DPCS then this field contains PSUB_JOBID.

10.2.19 The Servers Table

The servers table shown in Table 10.25, contains one entry for each RMS daemon. The table administration program, rmstbladm (see Page 5-44), creates the entries in the table. The daemons update their entries when they start up.

Table 10.25: Servers Table

Field	Type	Description
name	char(16)	server (daemon) name

(continued on next page)

Table 10.25: Servers Table (cont.)

Field	Type	Description
hostname	char(16)	host on which the daemon is running
port	int	IP port number to bind to for this server
pid	int	process ID of the server
rms	0 | 1	where daemon is an RMS server (1) or not (0)
startTime	int	time at which the daemon was started
autostart	0 | 1	where daemon starts automatically (1) or not (0)
status	char(8)	server status
args	char(32)	site-specific arguments for the server

The hostname field contains the name of the node on which the daemon should run, or rmshost if it should run on the rmshost node.

The rms field specifies whether the server is an RMS daemon or a conventional UNIX daemon. This controls the method used to determine whether or not the process is running.

The autostart field determines whether a daemon should be restarted automatically if it exits or is killed by a signal.

10.2.20 The Services Table

The services table shown in Table 10.26, is an internal table used by RMS to define the command to execute for each service, the names of the hosts that support the command and which users have permission to use the service. It contains one entry for each of the RMS clients that provides a configuration management service (for example, rmsquery and rcontrol). The entries are created by the table admininstration program, rmstbladm. See Chapter 5 (*RMS Commands*) for details of these services.

Table 10.26: Services Table

Field	Type	Description
name	char(16)	name of the service
hostname	char(16)	host on which the service runs, such as rmshost
group	char(8)	group(s) with access to the service
sequential	0 | 1	commands must wait for this command to finish
cmd	int	command to execute

The hostname field contains the name of the host on which the service should run.

The group field holds the name of the UNIX group which is allowed to run this service.

Currently, only rms is valid.

Some services, such as rcontrol, must have exclusive access to the database, requiring that other transactions wait until they complete. The sequential field should be set to 1 for these services. Others such as swctrl may run for long periods of time and should not block the execution of other transactions. sequential should be set to 0 for these services.

Sample records from the services table are shown in Table 10.27.

Table 10.27: Entries in the Services Table

name	hostname	group	sequential	command
rcontrol	rmshost	rms	1	/usr/opt/rms/bin/rcontrol
sql	rmshost	rms	1	/usr/opt/rms/bin/rmsquery

10.2.21 The Software Products Table

The software_products shown in Table 10.28, contains information about the components that make up a software product.

Table 10.28: Software Products Table

Field	Type	Description
name	char(16)	name of the product
version	char(16)	version of the product
component	char(16)	name of the component
comptype	char(16)	type of component
compversion	char(32)	version of the component
compattr	text	component attributes

The only valid value for the comptype field is subset.

The compattr field currently contains one value which dictates where a software component will be installed. The possible values are shown in Table 10.29.

Table 10.29: Component Attribute Values

Value	Description
opt	Component should only be installed on rmshost
root	Component should be installed on rmshost and the cluster root node(s)

10.2.22 The Switch Boards Table

The `switch_boards` shown in Table 10.30, contains one entry for each switch board in the Compaq AlphaServer SC Interconnect. It is created and maintained by the Switch Network Manager, `swmgr` (see Section 4.5).

Table 10.30: Switch Boards Table

Field	Type	Description
`name`	char(8)	board name
`module`	char(16)	name of module containing the board
`layer`	`0`–`31`	layer (or rail) number
`slot`	`0`–`31`	slot number in the module
`type`	char(16)	board type, such as `QM401` or `QM402`
`status`	char(8)	board status (`ok`, `absent`, `unknown`, `error`)
`environment`	char(32)	temperature data from thermistors on the board

10.2.23 The Transactions Table

Changes to the state of the machine are made through a request entered in the `transactions` table shown in Table 10.31. This table records who made each change, when it was made and whether or not the operation was successful.

The Transaction Log Manager, `tlogmgr` (see Section 4.6), actions requests in the `transactions` table, running commands on the user's behalf (in practice, the user here is a system administrator). This mechanism provides an audit trail, and sequential ordering of changes in state.

Table 10.31: Transaction Log Table

Field	Type	Description
`name`	char(16)	name of the service
`status`	char(16)	transaction status
`ctime`	UTC	creation time
`mtime`	UTC	last modification time
`handle`	int	unique identifier for the transaction
`logfile`	char(32)	`stdout` or `stderr` log for the transaction
`username`	char(16)	user issuing the command
`args`	text	arguments for the command

Valid values for the transaction `status` field are given in Table B.8.

An example of the transaction to add a partition is shown below in Table 10.32.

The `handle` is a unique number, generated automatically, which is passed to both the service and the client. The service uses the handle to label any output resulting from the transaction; the client uses the handle to select the resulting entries.

If the service fails, the output log (conventionally in the directory `/var/rms/adm/log`) may contain useful diagnostics. Client applications wait for the transaction to complete and then `cat` the logfile.

Table 10.32: Entry in the Transactions Table

name	status	handle	logfile
`rcontrol`	`complete`	44	`/var/rms/adm/log/tr44.log`

username	args
`rms`	`create partition=login nodes='n[0-3]'`

10.2.24 The Users Table

The `users` table shown in Table 10.33, contains information on each user's projects.

Table 10.33: Users Table

Field	Type	Description
`name`	char(16)	login name
`projects`	text	list of the user's projects

The `projects` field may contain a single project name or a comma-separated list of project names. The wildcard, `*`, may be specified as a project name denoting that the user is a member of all projects.

The ordering of the names in the list is significant: the first project specified is the user's *default* project. For purposes of accounting, access control and scheduling, the default project is assumed unless the user explicitly specifies another project. A project can be specified explicitly by using the environment variable `RMS_PROJECT` or by using the `-P` option to `prun` or `allocate`.

A

Compaq AlphaServer SC Interconnect Terms

A.1 Introduction

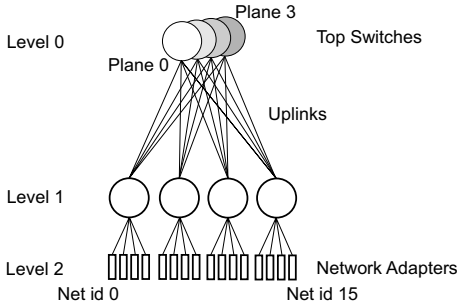
RMS includes support for programs that use Compaq AlphaServer SC Interconnect. This appendix provides an introduction to Compaq AlphaServer SC Interconnect, defining terms used elsewhere in this manual.

Before an application process can use Compaq AlphaServer SC Interconnect, it must be given an Elan capability (see Section C.2), describing the nodes and communications contexts that it is allowed to use. In general, processes present this capability to the kernel as they start.

Having granted a request for CPUs, RMS generates an appropriate capability and pushes it into the RMS kernel module on each of the allocated nodes. The capabilities together with information on the processes that make up the program can then be accessed through the `librmscall` system call library (see Section C.3 for details).

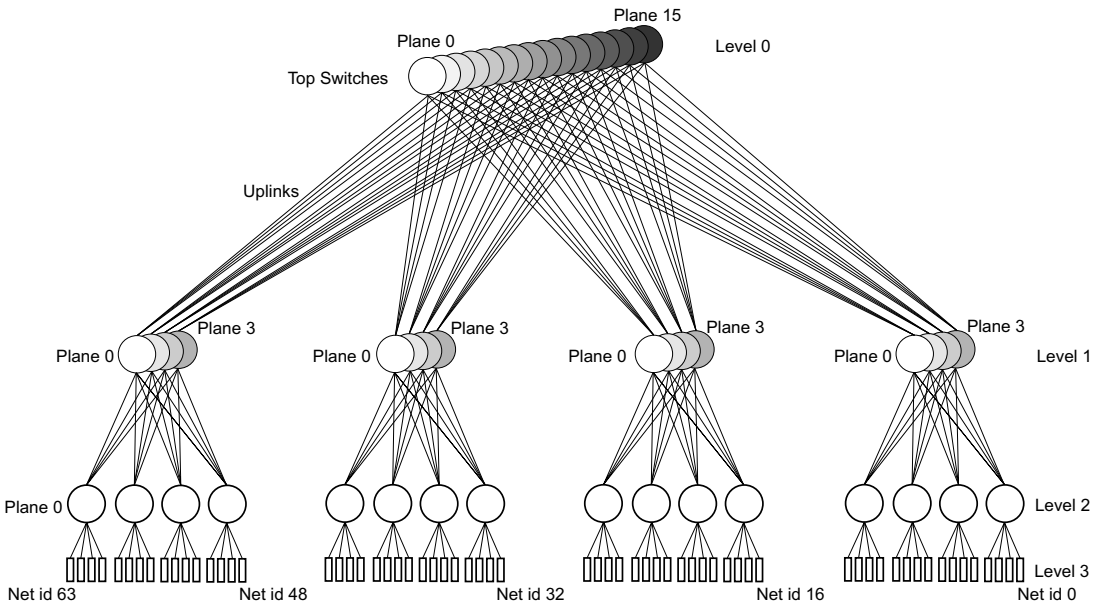
Compaq AlphaServer SC Interconnect is a multistage switch network, also known as a *fat tree network*. It is built from 8-way *crosspoint switches*, known as Elites. Each node is connected to the network by a network adapter, the Elan. The connection of a 2-stage (16-node) switch network is shown in Figure A.1.

Figure A.1: A 2-Stage, 16-Node, Switch Network



The *level* is the index of the stage, starting with 0 at the top. Note that in a 2-stage switch network the Elans are at level 2. Each component has a network ID that describes how to reach it from the top of the network. The *plane* is the index of switches that have the same switch network ID. The interconnection of a 3-stage (64-node) switch network is shown in Figure A.2.

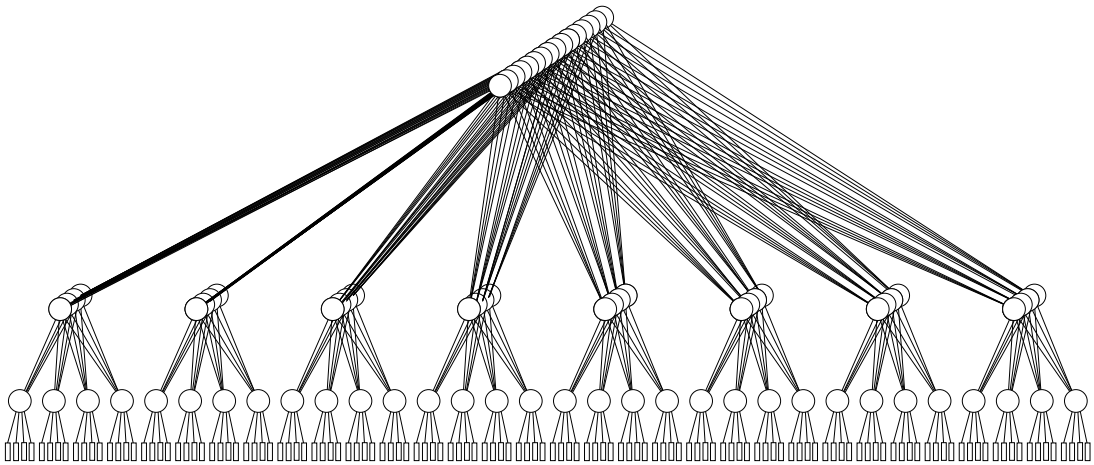
Figure A.2: A 3-Stage, 64-Node, Switch Network



The 3-stage switch network is composed from four 2-stage networks. Each 2-stage network connects sixteen nodes and has sixteen free *uplinks*. These uplinks connect the four 2-stage networks to an additional level of switches to form a 3-stage network, connecting up to 64 nodes.

Four such 64-node networks and an additional stage of switches can be used to construct a 256-way network. Alternatively, the unused uplinks can be used to double the number of nodes a switch can connect. This avoids the need to add an additional switch stage but the resulting network cannot be expanded further. This technique is used in the 128-node network, shown in Figure A.3.

Figure A.3: A 3-Stage, 128-Node, Switch Network



There are switch network modules that connect up to 16 or 128 nodes. The 16-node network is a rack-mountable module containing a single network board. The 128-node network is a rack-mountable chassis containing up to 24 network boards: 8 at the front connecting the nodes to the lower stages; and 16 at the rear providing the upper stages of the network. A central backplane joins the stages. These switch modules may be partially populated for networks containing fewer than 128 nodes.

The number of nodes and switches in these networks is shown in Table A.1. The number of switches refers to the total number of Elite ASICs required to construct the network. The number of hops refers to the maximum number of links traversed for nodes that have to communicate through a top switch. The bidirectional nature of the links means that traffic can be localized to a subtree large enough to span both nodes.

Table A.1: Switch Network Parameters

Name	Levels	Nodes	Switches	Hops
QM-S16	2	16	8	4
QM-S128	3	128	80	6

The Elan performs automatic routing and broadcast communications. Using the switch

network, data can be broadcast directly to a contiguous range of processors: data is routed up to a node in the tree from which all processors can be reached; then the data is routed down to all switch outputs in the broadcast range on the way down. Data can be recombined as it travels through the network to support global *reduction* operations and *barrier synchronization*.

Multiple Elan network adapters may be installed per node, each connected to a different switch network. This replication can increase fault tolerance and bisectional bandwidth, assuming each Elan is attached to a separate PCI bus. Each separate Elan/Elite network attached to a node is known as a *layer* (or a *rail*).

The switch network is described by three tables in the database. The `switch_boards` table (see Section 10.2.22) gives details of each board, its status and its position in the machine. The `elans` table (see Section 10.2.4) and the `elites` table (see Section 10.2.5) describe the position in the switch network of each component, its attributes and its current link state and errors.

RMS includes the control and monitoring daemon, `swmgr` (see Section 4.5), for managing the switch network. `swmgr` probes the switch network control interface for switch boards to determine the size of the network. It then creates or updates the entries in the `elans` table and the `elites` table. Having done this, the `swmgr` uses the switch network control interface to extract error and performance data. This interface is also used for link continuity (boundary scan) testing.

A.2 Link States

The state of each link in the switch network is recorded in the `linkerrors` field in the `elites` table (see Section 10.2.5). Valid values for the states are shown in Section B.4.

Links are normally in the *connected* state (`C`). Unconnected links will be in the *reset* state (`R`). Links will be in the *unknown* state (`U`) if the `swmgr` has not run or if the control cable is not attached to the switch. The states *acking* (`A`) and *nacking* (`N`) are set by the switch control software.

A.3 Link Errors

The `swmgr` logs network errors to the `link_errors` table (see Section 10.2.11). The description contains information that should be used in reporting a problem with the switch network.

B

RMS Status Values

B.1 Overview

This appendix lists the various states that RMS objects can enter. State information is stored in the `status` field of the RMS table for the object in question. For example, the current state of a partition is held in the `partitions` table (see Section 10.2.16). and the current state of a node is entered in the `nodes` table (see Section 10.2.14).

Status changes are recorded in the `events` table (see Section 10.2.6). Entries in the `events` table are identified by `class=`*`X`* and `name=`*`N`* where *`X`* is the class of object and *`N`* is its name. For most tables, the `name` field forms the primary key. In the case of a partition, both the `name` and the `configuration` fields are required to define a unique entry. In the case of an access control, both the `name` and `partition` fields are required.

Status values are shown for the following objects:

- Jobs (see Section B.3)
- Links (see Section B.4)
- Modules (see Section B.5)
- Nodes (see Section B.6)
- Partitions (see Section B.7)
- Resources (see Section B.8)
- Transaction (see Section B.9)

B.2 Generic Status Values

There are three generic status values:

`ok` This state means that an object is functioning correctly as far as the relevant RMS daemon can tell.

`error` This state means that one or more errors have been detected. A description of the problem will be found in the event record.

`unknown` This state means that the RMS daemon responsible for an object either has not run or is unable to determine the state of the object.

Where an object, such as a switch board, has many component status values, the `ok` state means that all component values are `ok`. If one or more of the components are in error then the status will be `error`.

B.3 Job Status Values

The status of each job is stored in the `status` field of the `jobs` table. It is updated by the partition manager when the job is started, suspended, resumed or completed. Valid job status strings are shown in Table B.1.

Table B.1: Job Status Values

Status	Description
`running`	Processes are scheduled
`suspended`	All processes are suspended
`finished`	All processes have exited
`hung`	One or more nodes is not responding
`aborted`	User aborted job
`failed`	Job failed
`killed`	Job was killed by a signal
`expired`	Time limit expired
`syskill`	Job was killed by an administrative user
`unknown`	Partition is blocked or down

If a job is killed because one of the nodes it was running on has crashed or was configured out then its final status value will be `failed`.

B.4 Link Status Values

Each switch (see Appendix A (*Compaq AlphaServer SC Interconnect Terms*)) has an entry in the `elites` table. Each switch has eight links and the state of each of these links is recorded in the `linkstate` field of the `elites` table. The field holds eight characters, one for each link. Valid values for the characters are as shown in Table B.2. See also Section A.2.

Table B.2: Link Status Values

State	Character	Description
Connected	`C`	Normal working state
Reset	`R`	Link is in reset
Acking	`A`	Link generates an ACK for all transactions
Nacking	`N`	Link generates an NACK for all transactions
Not connected	`_`	Link is not connected
Unknown	`U`	`swmgr` cannot determine link state.

B.5 Module Status Values

Module status information for nodes and the switch network is held in the `modules` table. The `estatus` field stores the node's operating environment; specifically, it shows whether the cooling fans and power supply units (PSUs) are working correctly and whether any of the other components are overheating.

Changes in environment status are recorded in the `events` table; entries are keyed by `class=module type=`*X* where *X* is `temperature`, `temphigh`, `psu`, or `fan`. The valid strings and their meaning are shown in Table B.3.

Table B.3: Module Status Values

Status	Description
`ok`	Fans, PSUs and temperature ok
`fan failure`	A fan has failed
`psu failure`	A power supply unit has failed
`node hot`	The enclosure is too hot
`cpu hot`	A CPU is too hot
`psu hot`	A power supply unit is too hot

If the environment status, `estatus`, is recorded as `ok`, the `environment` field of the `modules` table contains temperature readings from the CPU, PSU and enclosure. If a

node has more than one instance of each type of temperature sensor, the maximum of their values is recorded.

Temperature information is recorded as a list of attribute-value pairs, for example:

```
ambient=15 cpu=40 psu=20
```

Note that not all node types support all types of thermistor reading. The `environment` field may contain only a subset of this information.

If an error occurs, the environment string contains details of what has failed. For example, the following string indicates that the CPU fan number 1 has failed on the node.

```
cpu fan 1 failure
```

B.6 Node Status Values

The current state of a node is found in the `status` and `runlevel` fields of the `nodes` table. State changes are logged in the `events` table; entries in the `events` table are identified by `class=node` and `name=`*`N`*, where *`N`* is the name of the node as entered in the `name` field of the `nodes` table.

Provided a node is configured in (`configured` field set to 1), the `status` field contains one of the values shown in Table B.4.

Table B.4: Node Status Values

Status	Description
`not responding`	Machine Manager cannot get response from node
`active`	Node responds to IP requests but RMS is not running
`running`	RMS is active on this node

RMS does not monitor the state of a node while it is configured out (`configured=0`). It determines the status again when the node is configured back in.

As a node boots, its status progresses from `not responding` to `active` and on to `running`. A long delay in reaching the `running` state indicates a problem with booting that should be investigated further. If a node changes from the `running` state to the `active` state and stays there then there is a problem with RMS on that node. If a node changes from the `running` state to the `not responding` state then either the node has crashed or IP communications to the node are failing. In this case, RMS runs the `rmsevent_node` event handler script. This script attempts to determine what went wrong.

The current UNIX run level of a node is held in the nodes table in the runlevel field. This field is updated by the nodestatus program as the run level changes. The valid strings are shown together with their meaning in Table B.5.

Table B.5: Run Level Status Values

Status	Description
run level S	Single user mode
run level 1	UNIX run level 1
run level 2	UNIX run level 2
run level 3	UNIX run level 3

The run level of a node that is configured out will continue to be updated as the node is booted or halted. RMS is started at run level 3.

B.7 Partition Status Values

The current state of a partition is entered in the status field of the partitions table. Changes to a partition's state are logged in the events table. Entries in the events table are identified by class=partition and name=*P*, where *P* is the name of the partition as entered in the name field of the partitions table. All such events refer to partitions in the active configuration.

The partition's status can take one of the values shown in Table B.6.

Table B.6: Partition Status Values

Status	Description
running	The partition is operational
blocked	One or more of the rmsds in the partition is not responding
closing	The partition is in the process of closing down
down	The partition has been shut down successfully

Note that the active field in the partitions table denotes whether or not the partition is part of the active configuration. The database may contain a number of different configurations but only one is active at any time.

B.8 Resource Status Values

The status of each resource request is stored in the status field of the resources table. It is updated by the pmanager when CPUs are allocated and deallocated, and as

jobs using the CPUs complete. While CPUs are allocated, the valid resource status strings are as shown in Table B.7.

Table B.7: Resource Status Values

Status	Description
`blocked`	CPUs cannot be allocated because of a usage limit
`queued`	insufficient CPUs free for the request
`allocated`	CPUs are allocated
`suspended`	CPUs are temporarily deallocated
`finished`	CPUs have been deallocated
`aborted`	CPUs were deallocated when `prun` was killed (for example by `Ctrl/C`)
`killed`	CPUs were deallocated when resource request was killed
`failed`	CPUs were deallocated because node has crashed or was configured out

The final status of a resource is that of the last job to exit (see Table B.1).

B.9 Transaction Status Values

Requests to change the state of the machine are entered in the `transactions` table. The Transaction Log Manager, `tlogmgr`, actions the requests on the users' behalf. This mechanism provides an audit trail of state changes.

Each request has a `status` field associated with it. Valid values for this field are shown in Table B.8.

Table B.8: Transaction Status Values

Status	Description
`submitted`	Client application has submitted transaction
`started`	`tlogmgr` has started to execute the transaction
`complete`	Transaction completed successfully without errors
`error`	Transaction completed with errors
`failed`	Transaction failed

C

RMS Kernel Module

C.1 Introduction

The RMS kernel module supports the operation of RMS on each node in a system. It provides functions that bind together the set of processes that make up a program on each node, allowing RMS to apply scheduling, signal delivery and statistics gathering operations to them collectively. For example, the RMS kernel module allows the `rmsd` daemon or an administrator process to send a signal to all processes in a parallel program at the same time.

The RMS kernel module stores the Elan capabilities assigned to a program, making them available to the processes of that program and their children. The capablilities are only accessible to processes that belong to this one parallel program, they are not available to other processes. The RMS kernel module keeps track of processes through handlers called whenever a process belonging to a parallel program forks or exits.

Note that a parallel program under RMS may consist of one or more UNIX process groups or sessions. It is the ability of a UNIX process to call `setpgrp` or `setsid` at any time that makes the program structure provided by the RMS kernel module necessary; without it, suspend, resume, signal delivery and program cleanup operations on changing sets of processes are unreliable.

C.2 Capabilities

An Elan capability describes a parallel program's rights to use one rail of Compaq AlphaServer SC Interconnect. It specifies the range of nodes that have been allocated

and the Elan hardware context numbers to be used.

```
typedef struct elan_capability
{
    ELAN_USERKEY UserKey;                /* User defined protection      */
    int          Version;                /* Version number               */
    short        Type;                   /* Type                         */
    short        Generation;             /* Generation number            */
    int          LowContext;             /* low context number in block  */
    int          HighContext;            /* high context number in block */
    int          MyContext;              /* my context number            */
    int          LowNode;                /* low elan id of group         */
    int          HighNode;               /* high elan id of group        */
    int          Entries;                /* number of processes          */
    int          RouteTable;             /* route table name to use      */
    unsigned int RailMask;               /* rails this capability is valid for */
    bitmap_t     Bitmap[ELAN_BITMAPSIZE]; /* Bitmap of process to node translation  */
} ELAN_CAPABILITY;
```

Elan capabilities are created on each node allocated to a parallel program and passed to its processes through the RMS kernel module.

C.3 System Call Interface

The RMS kernel module is accessed through its system call interface. This interface allows processes with administrator privileges to create program descriptions, add Elan capabilities to them, collect resource utilization statistics from them and destroy them when their processes have exited. It also allows them to suspend or resume the processes and deliver signals to them.

The RMS system call interface allows user processes to determine how many nodes, CPUs, rails and contexts they have been allocated. This information is primarily (but not exclusively) for use by parallel programming libraries.

NAME

rms_setcorepath, rms_getcorepath – Set, get the path for application core files

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>

int rms_setcorepath(caddr_t path);
int rms_getcorepath(pid_t pid, caddr_t path, int maxlen);
```

PARAMETERS

`path`	Array containing the path name.
`maxlen`	Size of the array pointed to by `path`.
`pid`	Process identifier.

DESCRIPTION

The function `rms_setcorepath()` sets the core file path for the current process. The core file path is set by `rmsd` as it starts a new parallel program and inherited by any child processes. Administrator privileges are required to set a core path.

The function `rms_getcorepath()` returns the core file path of a process. If *`pid`* is negative, it returns the core file path of the current process.

RETURN VALUES

Upon successful completion, `rms_setcorepath()` and `rms_getcorepath()` return `0`. Otherwise, they return `-1` and set `errno` to indicate the error.

`EACCESS`	Caller is not permitted to perform this operation.
`ENOMEM`	Insufficient memory to perform this operation.
`ESRCH`	Process does not exist.
`EEXIST`	The core path has not been set.

NAME

rms_prgcreate, rms_prgdestroy – Create, destroy program descriptions

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>

int rms_prgcreate(int id, uid_t uid, int cpus);
int rms_prgdestroy(int id);
```

PARAMETERS

`id`	Program identifier.
`uid`	Owner of the program.
`cpus`	Number of CPUs allocated.

DESCRIPTION

`rms_prgcreate()` creates a new program description with the current process as its root process. Any children of this process will belong to this program. `rms_prgdestroy()` destroys an existing program description. Calls to `rms_prgdestroy()` will fail if processes belonging to this program are still running. Creating or destroying program descriptions requires administrator privileges. The `rmsd` creates a new program description for each parallel program. The program identifier is the unique identifier for the parallel job, that is to say, it is common across nodes.

RETURN VALUES

Upon successful completion, `rms_prgcreate()` and `rms_prgdestroy()` return `0`. Otherwise, they return `-1` and set `errno` to indicate the error.

`EACCESS`	Caller is not permitted to perform this operation.
`ENOMEM`	Insufficient memory to perform this operation.

EINVAL	Program identifier is in use or the number of CPUs is invalid.
ECHILD	Processes belonging to this program are still running.
EEXIST	Program identifier does not exist.

SEE ALSO

rms_getprgid(3)

NAME

rms_prgids, rms_prginfo, rms_getprgid – Get information on a program or programs

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>
int rms_prgids(int maxids, int *ids, int *nids);
int rms_prginfo(int id, int maxids, pid_t *pids, int nids);
int rms_getprgid(int pid, int *id);
```

PARAMETERS

`id`	Program identifier.
`pid`	Process identifier.
`maxids`	Maximum number of identifiers to be returned.
`ids`	Array of program identifiers.
`pids`	Array of process identifiers.
`nids`	Number of program or process identifiers returned.

DESCRIPTION

`rms_prgids()` returns the identifiers of each active program. `rms_prginfo()` returns the identifiers for each process belonging to a particular parallel program – the current program if *`id`* is negative. `rms_getprgid()` returns the program identifier (if any) for a particular process – the current process if *`pid`* is negative.

RETURN VALUES

Upon successful completion, `rms_prgids()`, `rms_prginfo()` and `rms_getprgid()` return `0`. Otherwise, they return `-1` and set `errno` to indicate the error.

`EACCESS`	Caller is not permitted to perform this operation.

EINVAL	Count of number of array elements is invalid.
EFAULT	Array address is invalid.
ENOMEM	Insufficient kernel memory to perform this operation.
ESRCH	Process or program does not exist.

SEE ALSO

rms_prgcreate(3)

NAME

rms_prgsuspend, rms_prgresume, rms_prgsignal – Suspend or resume the processes in a program, deliver a signal to all processes in a program

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>

int rms_prgsuspend(int id);
int rms_prgresume(int id);
int rms_prgsignal(int id, int signo);
```

PARAMETERS

`id`	Program identifier.
`signo`	Signal number.

DESCRIPTION

`rms_prgsuspend()` suspends all of the processes in a program. The RMS suspends a parallel program by calling `rms_prgsuspend()` on each node that it is using. `rms_prgsuspend()` requires administrator privileges. `rms_prgresume()` resumes all of the processes in a program. The RMS resumes a parallel program by calling `rms_prgresume()` on each node that it is using. `rms_prgresume()` requires administrator privileges.

`rms_prgsignal()` sends a signal to all of the processes in a program. The RMS delivers signals to a parallel program by calling `rms_prgsignal()` on each node that it is using. The function is also used to confirm that all processes belonging to a program have exited. `rms_prgsignal()` can be called by the owner of the program or a process with administrator privileges.

RETURN VALUES

Upon successful completion, `rms_prgsuspend()`, `rms_prgresume()` and `rms_prgsignal()` return `0`. Otherwise, they return `-1` and set `errno` to indicate the error.

EACCESS	Caller is not permitted to perform this operation.
ESRCH	No such program identifier.
EINVAL	Invalid signal number.

SEE ALSO

rms_prgcreate(3)

NAME

rms_prgaddcap, rms_setcap – Associate Elan capabilities with a program or process

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>

int rms_prgaddcap(int id, int index, ELAN_CAPABILITY *cap);
int rms_setcap(int index, int context);
```

PARAMETERS

`id`	Program identifier.
`index`	Index of the capability for this program.
`cap`	Pointer to a capability.
`context`	Context number for this process.

DESCRIPTION

`rms_prgaddcap()` and `rms_setcap()` associate Elan capabilities with a program and its processes. The function `rms_prgaddcap()` adds a new capability to a program. It is called once for each rail in use by the program. Each capability defines the range of node numbers and Elan hardware contexts available to a parallel program. Capabilities are indexed from `0` to *`ncaps`*`-1` where *`ncaps`* is the number of capabilities allocated.

`rms_prgaddcap()` requires administrator privileges. It is called by `rmsd` as it creates a parallel program.

The function `rms_setcap()` assigns Elan hardware context numbers to the current process. It is called by the RMS application loader, `rmsloader`, as it creates each new application process. The contexts assigned must lie within a previously assigned capability for the program.

RETURN VALUES

Upon successful completion, `rms_prgaddcap()` and `rms_setcap()` return `0`. Otherwise, they return `-1` and set `errno` to indicate the error.

EACCESS	Caller is not permitted to perform this operation.
ENOMEM	There was insufficient memory to perform this operation.
ESRCH	Program does not exist.
EFAULT	Capability has invalid address.
EINVAL	Invalid context number (`rms_setcap()` only).

SEE ALSO

rms_ncaps(3)

NAME

rms_ncaps, rms_getcap – Return information on the Elan capabilities allocated to a process in a parallel program

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>

int rms_ncaps(int *ncaps);
int rms_getcap(int index, ELAN_CAPABILITY *cap);
```

PARAMETERS

`ncaps`	Number of capabilities allocated.
`index`	Index of a capability to be returned.
`cap`	Pointer to a capability.

DESCRIPTION

`rms_ncaps()` returns the number of Elan capabilities allocated to a program. `rms_getcap()` returns a specified Elan capability. Capabilities are indexed from `0` to *`ncaps`*`-1`.

RETURN VALUES

Upon successful completion, `rms_ncaps()` and `rms_getcap()` return `0`. Otherwise, they return `-1` and set `errno` to indicate the error.

`EFAULT`	Invalid address.
`EINVAL`	Invalid capability identifier.
`EEXIST`	Calling process is not part of a parallel program.

SEE ALSO

`rms_prgaddcap``(3)`

NAME

rms_prggetstats – Return resource usage information for a program

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmscall [ library ...  ]
#include <rms/rmscall.h>
int rms_prggetstats(int id, prgstats_t *stats);
```

PARAMETERS

`id`	Program identifier.
`stats`	Pointer to a program statistics structure.

DESCRIPTION

`rms_prggetstats()` returns resource usage information for the processes of a parallel program on the calling node. The RMS kernel module sums resource usage over the processes in a program. The statistics returned by `rms_prggetstats()` are the sum over all processes belonging to program *`id`* on this node, including those that have already exited.

Setting *`id`* to `-1` instructs the RMS kernel module to return values for the caller's program. Resource utilization statistics are available to the owner of the program and to any process with administrator privileges.

```
/*
 * program statistics
 */
typedef struct {
    uint64_t etime;             /* elapsed cpu time (millisecs)       */
    uint64_t atime;             /* allocated cpu time (millisecs)     */
    uint64_t utime;             /* user cpu time (millisecs)          */
    uint64_t stime;             /* system cpu time (millisecs)        */
    int ncpus;                  /* number of cpus allocated           */
    int flags;                  /* program status flags               */
    int mem;                    /* max memory size in megabytes       */
    int pageflts                /* number of page faults              */
    uint64_t memint;            /* memory integral                    */
} prgstats_t;
```

The elapsed time statistic `etime` is the time in millisecs since the program was created. The allocated time statistic `atime` is the time in millisecs for which CPUs have been allocated multiplied by the number of CPUs allocated. The `utime` and `etime` statistics are summed over the processes that make up the program (on this node).

If one or more processes belonging to the program is still running, the `flags` field will contain the value `PRG_RUNNING`. This will be replaced by `PRG_ZOMBIE` when the last process has exited. The program description should be destroyed when this value is seen.

The Partition Manager periodically sums these statistics over the nodes used to run a parallel program, updating its entry in the accounting statistics (`acctstats`) table.

RETURN VALUES

Upon successful completion, `rms_prggetstats()` returns `0`. Otherwise, it returns `-1` and sets `errno` to indicate the error.

`EACCESS`	Caller is not permitted to perform this operation.
`EFAULT`	Invalid address for statistics array.
`ESRCH`	No such program.

SEE ALSO

`rms_prginfo``(3)`

D

RMS Application Interface

D.1 Introduction

The RMS application interface is provided so that external scheduling modules can make inquiries about the availability of resources, allocate and deallocate CPUs and perform job control operations.

The application interface is provided as a dynamic library `librmsapi.so`. Function prototypes are defined in the header file `<rms/rmsapi.h>`.

NAME

rms_allocateResource, rms_deallocateResource – Allocate or deallocate a resource

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmsapi -lrms [ library ...  ]
#include <rms/rmsapi.h>
int rms_allocateResource(char *partition, int cpus, int baseNode,
                         int nodes, uid_t uid, char *project,
                         char *requestFlags);
int rms_deallocateResource(int rid);
```

PARAMETERS

`partition` Partition containing the resources.

`cpus` Total number of CPUs to allocate.

`baseNode` ID of the first node to allocate.

`nodes` Number of nodes to allocate.

`uid` User on whose behalf the resource should be allocated.

`project` User's project name.

`requestFlags` The request flags currently supported are as follows:

`immediate=0 | 1`
With a value of 1, this specifies that the request should fail if resources are not available immediately.

`hwbcast=0 | 1` With a value of 1, this specifies a contiguous range of nodes and constrains the scheduler to queue the request until a contiguous range becomes available.

`rails=`*n* In a multirail system, this specifies the number of rails required, where $1 \leq n \leq 32$.

Multiple request flags can be given as a comma-separated list; `immediate=1,hwbcast=1`, for example.

`rid` ID of the resource to deallocate.

DESCRIPTION

`rms_allocateResource()` allocates CPUs from a named partition. If `partition` is `NULL`, the default partition is used, otherwise the named partition must exist. You can optionally specify the base node and the number of nodes (as with the `allocate` and `prun` commands). Alternatively, this can be left to the scheduler by passing the value `RMS_UNASSIGNED`. If the requested CPUs are not available, the request will block unless the `immediate` flag has been entered, in which case it will fail.

If the caller belongs to the `rms` group then `rms_allocateResource()` can be used to allocate CPUs on behalf of another user identified by `uid`. In this case, the CPUs will be available to this user only. If `project` is not null, the request is subject to the usage restrictions of, and is accounted to, the specified project, which must exist. If `project` is null, the user's default project applies.

To run a program on the specified resource, the environment variable `RMS_RESOURCEID` must be set to the value *`partition.rid`* (where *`partition`* is the name of the partition and *`rid`* is the resource id returned by `rms_allocateResource`) before executing `prun`.

`rms_deallocateResource()` deallocates a resource that is no longer in use.

RETURN VALUES

Upon successful completion, `rms_allocateResource()` returns the ID of the resource allocated. This value should be passed to subsequent calls. A negative integer is returned on error. The supported error codes are as follows:

-1 Request cannot be met.
-2 Request cannot be met now and `immediate` was not set to zero.

`rms_deallocateResource()` returns `0` on success and `-1` on error.

SEE ALSO

`rms_suspendResource``(3)`, **`rms_defaultPartition`**`(3)`

NAME

rms_run – Run a program on an allocated resource

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmsapi -lrms [ library ...  ]
#include <rms/rmsapi.h>
int rms_run(int rid, char *cmd, char **args, char *jobFlags);
```

PARAMETERS

`rid`	Resource id.
`cmd`	Command to execute.
`args`	Arguments for the command.
`jobFlags`	The job flags currently supported are as follows:

`tag=0 | 1`	With a value of 1, this specifies that output from each process should be tagged by the process id.
`verbose=n`	Set the level of verbose output from the program. Supported values are `0` quiet, `1` minimal output, and `2` full output.

Multiple request flags can be given as a comma-separated list; `tag=1,verbose=1`, for example.

DESCRIPTION

`rms_run()` starts a parallel program on a previously allocated resource. Any `stdio` to and from the program is forwarded while one or more processes is running.

The call returns when the program completes.

RETURN VALUES

Upon successful completion, `rms_run()` returns the global OR of the exit status values of the processes in the parallel program.

SEE ALSO

rms_allocateResource(3),

NAME

rms_suspendResource, rms_resumeResource, rms_killResource – Job control operations on allocated resources

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmsapi -lrms [ library ...  ]
#include <rms/rmsapi.h>

int rms_suspendResource(int rid);
int rms_resumeResource(int rid);
int rms_killResource(int rid, int signo);
```

PARAMETERS

`rid`	ID of the resource.
`signo`	Signal to send.

DESCRIPTION

`rms_suspendResource()` and `rms_resumeResource()` suspend and resume a resource specified by *`rid`*. The caller must be either the owner of the resource or a member of the `rms` group.

`rms_killResource()` sends a signal to all of the processes in all of the jobs running on a specified resource. The caller must be either the owner of the resource or a member of the `rms` group.

RETURN VALUES

Upon successful completion, `rms_suspendResource()`, `rms_resumeResource()` and `rms_killResource()` return `0`. On error they return a negative integer.

SEE ALSO

`rms_allocateResource``(3)`

NAME

rms_defaultPartition, rms_numCpus, rms_numNodes, rms_freeCpus – Provide information on RMS partitions

SYNOPSIS

```
cc [ flag ...  ]  file ...  -lrmsapi -lrms [ library ...  ]
#include <rms/rmsapi.h>

char *rms_defaultPartition();
int rms_numCpus(char *partition);
int rms_numNodes(char *partition);
int rms_freeCpus(char *partition);
```

PARAMETERS

`partition`	Name of an active partition.

DESCRIPTION

`rms_defaultPartition()` assigns the name of the default partition, if one exists, to *`partition`*. `rms_numCpus()` returns the total number of CPUs in the named partition. `rms_numNodes()` returns the total number of nodes in the named partition. `rms_freeCpus()` returns the number of free CPUs in the named partition.

The calling process must run on a node in the Compaq AlphaServer SC system.

RETURN VALUES

`rms_defaultPartition()` returns `NULL` on error. Other functions return `0` or greater on success or `-1` on error.

SEE ALSO

`rms_allocateResource``(3)`

E

Accounting Summary Script

E.1 Introduction

This appendix describes the example accounting summary script included in `/usr/opt/rms/examples/scripts/accounting_summary` and referred to in Section 9.4.3.

- Section E.2 describes the command line interface.
- Section E.3 shows a sample of output from the script.
- Section E.4 is a listing of the script.

E.2 Command Line Interface

The script has the following command line interface:

```
accounting_summary [ -hd [-u | -p] [-M | -H] ] [days]
```

The options are as follows:

`-h`	Display help on the options.
`-d`	Delete the accounting records of all resource requests that have completed after outputting the accounting summary.
`-u`	Sort the records by user name and then by project name.

`-p`	Sort the records by project name and then by user name. This is the default.
`-M`	Show time in minutes rather than seconds.
`-H`	Show time in hours rather than seconds.
`days`	Show statistics for the specified number of days. By default, statistics are shown for the previous day only.

The script processes the arguments passed to it on the command line and generates a SQL query which it passes to `rmsquery`. The query acts on two tables in the RMS database: the accounting statistics (`acctstats`) table, and the `resources` table. The information returned by the query is formatted to produce output as shown in Section E.3.

If the `-d` option is specified on the command line, after printing the accounting summary, the script generates another SQL query to delete all accounting records that have their `running` field set to 0, denoting that the resource request has completed.

If a query fails, the script outputs an error message.

E.3 Example Output

An example of using the script, together with the output produced, follows. After running the script, all of the accounting records for resource requests that have finished are deleted.

```
# accounting_summary -d

Accounting Summary of Machine atlas at 16:01 Wed 21 Feb 2001
Usage by Project/User For Previous Day

Project    User               CPU          User           Sys  Number
Name       Name              Secs          Secs          Secs Sessions
------------------------------------------------------------------------
default    addy               596           533             6      8
           duncan              58            37             2      6
           johnt              540           227            51     15
           root             29272             2             8     37
           stephen            286            87           134     56
------------------------------------------------------------------------
Total default               30751           885           201    122
------------------------------------------------------------------------
Grand Total                 30751           885           201    122
------------------------------------------------------------------------
```

E.4 Listing of the Script

```
#!/bin/sh
##############################################################################
#
# accounting_summary
#
##############################################################################

usage() {
    echo "Usage : $sname [ -u -p -d [ -M | -H ] ] [ days ]"
}

help() {
   usage
   echo "\t-h\tThis help message"
   echo "\t-p\t'Project' is primary sort field (default)"
   echo "\t-u\t'User' is primary sort field"
   echo "\t-M\tShow time in minutes rather than seconds"
   echo "\t-H\tShow time in hours rather than seconds"
   echo "\t-d\tDelete all 'not running' accounting records after producing summary"
   exit 0
}

#
# main
#
sname="accounting_summary"
OS=`uname`
if [ "$OS" = "Linux" ]; then
    RMSPATH="/usr/bin"
else
    if [ "$OS" = "OSF1" ]; then
        RMSPATH="/usr/opt/rms/bin"
    else
        RMSPATH="/opt/rms/bin"
    fi
fi
RMSGETTIME=${RMSPATH}/rmsgettime
RMSQUERY=${RMSPATH}/rmsquery
tmpfile="/tmp/accounting_summary_$$"
if [ -x /bin/gawk ]; then
    AWK="/bin/gawk"
else
    AWK="/usr/local/bin/gawk"
fi
primary="project"
delete=""
hours=""
minutes=""
```

Listing of the Script

```
#
# parse the options
#

while [ $# -gt 0 ]; do
   option=‘echo $1 | sed "s/^-//"‘
   if [ "$option" = "$1" ]; then
      break
   fi
   if [ "$option" = "p" ]; then
   primary="project"
   elif [ "$option" = "u" ]; then
   primary="user"
   elif [ "$option" = "d" ]; then
   delete="1"
   elif [ "$option" = "M" ]; then
        if [ "$hours" = "1" ]; then
            echo "$sname: ERROR : -M and -H are mutually exclusive"
            exit 1
        fi
   minutes="1"
   elif [ "$option" = "H" ]; then
        if [ "$minutes" = "1" ]; then
            echo "$sname: ERROR : -M and -H are mutually exclusive"
            exit 1
        fi
   hours="1"
   elif [ "$option" = "h" ]; then
   help
   else
      echo "$sname: ERROR : invalid option $1"
      usage
      exit 1
   fi
   shift
done

if [ $# -gt 0 ]; then
    days=$1
    shift
else
    days=1
fi

if [ $# -gt 0 ]; then
    usage
    exit 1
fi

now=‘$RMSGETTIME‘
secsperday=‘expr 60 \* 60 \* 24‘
daysecs=‘expr $secsperday \* $days‘
```

```
starttime=‘expr $now - $daysecs‘

if [ "$primary" = "project" ]; then
    primarytitle="Project"
    secondarytitle="User"
    querystr="select \
    acctstats.project,resources.username, \
    acctstats.atime,acctstats.utime, acctstats.stime \
    from resources,acctstats \
    where acctstats.started > $starttime and resources.name=acctstats.name \
    order by acctstats.project,resources.username"
else
    primarytitle="User"
    secondarytitle="Project"
    querystr="select \
    resources.username,acctstats.project, \
    acctstats.atime,acctstats.utime,acctstats.stime \
    from resources,acctstats \
    where acctstats.started > $starttime and resources.name=acctstats.name \
    order by resources.username,acctstats.project"
fi

machine=‘rinfo -m‘

/bin/rm -f $tmpfile
$RMSQUERY $querystr > $tmpfile
if [ $? -ne 0 ]; then
    echo "$sname : ERROR : $RMSQUERY $querystr FAILED"
    exit 1
fi

cat $tmpfile | \
$AWK ’BEGIN {
    primary     = ""
    secondary   = ""
    nvalues     = 3
    for (i=1; i<=nvalues; i++) {
        values[i]       = 0
        primvalues[i]   = 0
        grandvalues[i]  = 0
    }
    recs        = 0
    primrecs    = 0
    grandrecs   = 0
    printprim   = 1
}

function printsortfields() {
    if (printprim == 1) {
    printf ("%-10.10s %-8.8s ", primary, secondary)
    printprim = 0
    } else {
```

Listing of the Script

```
            printf ("\t   %-8.8s ", secondary)
        }
    }
    function printdashes() {
        printf
       ("-------------------------------------------------------------------\
    ----\n")
    }

    function printvals(vals,   i) {
        for (i=1; i<=nvalues; i++) {
            if (hours == 1 || minutes == 1) {
                printf (" %13.2f", vals[i])
            } else {
                printf (" %13.0f", vals[i])
            }
        }
    }
    NF > 0 {
        if ($1 != primary) {
           if (primary != "") {
               printsortfields()
               printvals(values)
               printf (" %6d\n", recs)
               printdashes()
               printf ("Total %-10.10s    ", primary)
               printvals(primvalues)
               printf (" %6d\n", primrecs)
               printdashes()

            } else {
           datestr = strftime("%H:%M %a %d %b %Y")
           titlestr = sprintf ("Accounting Summary of Machine %s at %s", \
                               machine, datestr)
           print titlestr
           if (days > 1) {
               daystr = sprintf ("%d Days",days)
           } else {
               daystr = "Day"
           }

           printf ("Usage by %s/%s For Previous %s\n", primtitle, sectitle, daystr)
           printf ("\n")
           printf ("%-10.10s %-8.8s       CPU     User       Sys  Number\n", \
                   primtitle, sectitle)
           if (hours == 1) {
                   printf ("Name       Name          Hours    Hours    Hours Sessions\n")
           } else {
               if (minutes == 1) {
                   printf ("Name       Name          Mins     Mins      Mins Sessions\n")
               } else {
```

```
            printf ("Name       Name          Secs     Secs      Secs Sessions\n")
         }
      }
      printdashes()
   }
   primary = $1
   secondary = $2
   for (i=1; i<=nvalues; i++) {
       values[i]       = 0
       primvalues[i]   = 0
   }
   recs      = 0
   primrecs  = 0
   printprim = 1
   } else {
       if ($2 != secondary) {
           printsortfields()
           printvals(values)
           printf (" %6d\n", recs)
           secondary = $2
           for (i=1; i<=nvalues; i++) {
            values[i] = 0
           }
           recs = 0
       }
    }
    for (i=1; i<nvalues; i++) {
        if (hours == 1) {
            val = $(i+2) / 3600
        } else {
            if (minutes == 1) {
                val = $(i+2) / 60
            } else {
                val = $(i+2)
            }
        }
        values[i] = values[i] + val
        primvalues[i] = primvalues[i] + val
        grandvalues[i] = grandvalues[i] + val
    }
    recs++
    primrecs++
    grandrecs++
}
END {
    printsortfields()
    printvals(values)
    printf (" %6d\n", recs)
    printdashes()
    printf ("Total %-10.10s    ", primary)
    printvals(primvalues)
```

Listing of the Script

```
        printf (" %6d\n", primrecs)
        printdashes()
        printf ("Grand Total         ")
        printvals(grandvalues)
        printf (" %6d\n", grandrecs)
        printdashes()
    }' primtitle="$primarytitle" sectitle="$secondarytitle" machine=$machine \
        days=$days hours=$hours minutes=$minutes

    /bin/rm $tmpfile

    if [ "$delete" ]; then
        echo "$sname : Deleting accounting statistics records"
        querystr="delete from acctstats where running=0"
        $RMSQUERY $querystr
        if [ $? -ne 0 ]; then
            echo "$sname : ERROR : $RMSQUERY $querystr FAILED"
            exit 1
        else
            echo "$sname : Accounting statistics records deleted"
        fi
    fi

    exit 0
```

Glossary

Abbreviations

API Application Program Interface — specification of interface to software package (library).

CFS Cluster File System — the file system for Tru64 UNIX clusters.

CGI Common Gateway Interface — a standard method for generating HTML pages dynamically from an application so that a Web server and a Web browser can exchange information. A CGI script can be written in any language and can access various types of data, for example, a SQL database.

CPU Central Processing Unit — the part of the computer that executes the machine instructions that make up the various user and system programs.

CRC Cyclic Redundancy Check — a method of error detection.

CVS Concurrent Versions System — a revision control utility for managing software releases and controlling the concurrent editing of files by multiple software developers.

DIMM Dual In-Line Memory Module.

DMA Direct Memory Access — high performance I/O technique where peripherals read/write memory directly and not through a CPU.

GNU GNU's Not UNIX — A UNIX-like development effort of the Free Software Foundation, headed by Richard Stallman.

HTML HyperText Markup Language — a generic markup language, comprising a set of tags, that enables structured documents to be delivered over the World Wide Web and viewed by a browser.

HTTP HyperText Transfer Protocol — a communications protocol commonly used between a Web server and a Web browser together with a URL (Uniform Resource Locator).

LED Light-Emitting Diode.

MIMD Multiple Instruction, Multiple Data — parallel processing computer architecture characterized as having multiple processors each (potentially) executing a different instruction sequence on different data.

MMU Memory Management Unit — part of CPU that provides protection between user processes and support for virtual memory.

MPI Message Passing Interface — parallel processing API.

MPP Massively Parallel Processing — processing that involves the use of a large number of processors in a coordinated fashion.

PCI Peripheral Component Interconnect — the Elan is connected to a node through this interface.

PDF Portable Document Format — the page description language used by Adobe Acrobat, derived from PostScript, for displaying pages on the screen.

PTE Page Table Entry — an entry in the page table which maps the base address of a page to physical memory.

RISC Reduced Instruction Set Computer — a computer whose machine instructions represent relatively simple operations that can be executed very quickly.

RMS Resource Management System — Quadrics software for managing clusters of UNIX nodes.

SDRAM Synchronous Dynamic Random Access Memory — high performance computer memory architecture.

Shmem A one-sided (put/get) inter-process communication interface used on high-performance parallel systems.

SMP Symmetric MultiProcessor — a computer whose main memory is shared by more than one processor.

SNMP Simple Network Management Protocol — a protocol used to monitor and control devices on the Internet.

SQL Structured Query Language — a database language.

TLB Translation Lookaside Buffer — part of the MMU that caches the result of virtual to physical address translations to minimize translation times in subsequent accesses to the same page.

URL Uniform Resource Locator — a standard protocol for addressing information on the World Wide Web.

UTC Coordinated Universal Time[1] — on UNIX systems it is represented as the time elapsed in seconds since January 1st, 1970 at 00:00:00.

Terms

barrier A synchronization point in a parallel computation that all of the processes must reach before they are allowed to continue.

bisectional bandwidth
The worst case bandwidth across the diameter of the network.

block A thread that blocks without relinquishing the processor until a specified event occurs.

critical section A section of program statements that can yield incorrect results if more than one thread tries to execute the section at the same time.

Elan memory The SDRAM on the Elan card.

event A parallel-processing synchronization primitive implemented by the Elan card.

[1] Used to be called GMT.

Flit A communications cycle unit of information.

HTTP cookies Cookies provide a general mechanism that HTTP server-side connections use to store and to retrieve information on the client side of the connection.

main memory The memory normally associated with the main processor, that is to say, memory on the CPU's high speed memory bus.

main processor The main CPU (or CPUs for a multi-processor) of a node, typically an Alpha™ 21264.

management network

A private network used by the RMS daemons for control and diagnostics.

multirail system

A system that has more than one Elan card connected to each node, each Elan card being connected to a different switch network.

multi-threaded program

A multi-threaded program is one that is constructed such that, during its execution, multiple sequences of instructions are executed concurrently (possibly by different CPUs). Each thread of execution has a separate stack but otherwise they all share the same address space.

node A system with memory, one or more CPUs and one or more Elan cards running an instance of the operating system.

poll Loop and check on each loop whether a specified event has occurred.

rank An integer value that identifies a single process from a set of parallel processes.

reduce Combine the results of a parallel computation into a single value.

remote memory The memory (Elan card or main) of a node when accessed by another node over the network.

resource A set of CPUs allocated to a user to run one or more parallel jobs.

slice A local copy of a global object.

switch network The network constructed from the Elan cards and Elite cards.

thread An independent sequence of execution. Every host process has at least one thread.

virtual memory A feature provided by the operating system, in conjunction with the MMU, that provides each process with a private address space that may be larger than the amount of physical memory accessible to the CPU.

virtual process A (possibly multi-threaded) component of a parallel program executing on a node.

word A 64-bit value.

Index

E

G

I

J

L

M

N

P

R

S

T

U