Bull

LoadLeveler V2R2
Using and Administering

AIX

ORDER REFERENCE
86 A2 14EF 00

Bull

LoadLeveler V2R2
Using and Administering

AIX

Software

October 2000

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 14EF 00

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Year 2000

The product documented in this manual is Year 2000 Ready.

Contents

Who Should Use This Book

This manual is intended for those who are responsible for using and/or administering LoadLeveler.

Tasks involved with using LoadLeveler include submitting parallel, serial, and interactive jobs. Tasks involved with administering Loadleveler include:

- Setting up configuration and administration files
- Maintaining LoadLeveler
- Setting up the distributed environment for allocating batch jobs.

Users and Administrators should be experienced with the UNIX** commands. Administrators should be familiar with system management techniques such as SMIT, as it is used in the AIX* environment. Knowledge of networking and NFS** or AFS** protocols is helpful, as well as knowledge of DCE.

How this Book is Organized

This books contains the following sections:

- "Part 1. Overview of LoadLeveler" on page 1 describes what LoadLeveler is and how it works, and includes an explanation of the LoadLeveler daemons and processes.
- "Part 2. Using LoadLeveler" on page 21 describes how to submit both serial and parallel jobs to LoadLeveler.
- "Part 3. Administering LoadLeveler" on page 69 describes how to perform administration tasks, such as configuring LoadLeveler, gathering accounting data, and routing jobs to NQS.
- "Part 4. Command Reference" on page 165 describes the LoadLeveler commands.
- "Part 5. The LoadLeveler Graphical User Interface" on page 221 describes the LoadLeveler graphical user interface.
- "Part 6. The LoadLeveler Application Programming Interfaces" on page 249 describes LoadLeveler's application programming interfaces.
- The appendices include "Appendix A. Troubleshooting" on page 305, and "Appendix B. Customer Case Studies" on page 319.

A glossary and index are also included.

Users of LoadLeveler should, at a minimum, become familiar with "Part 1. Overview of LoadLeveler" on page 1 and "Part 2. Using LoadLeveler" on page 21. Administrators should, at a minimum, become familiar with "Part 3. Administering LoadLeveler" on page 69, and may find it helpful to read "Troubleshooting LoadLeveler" on page 305.

Typographic Conventions

This book uses the following typographic conventions:

Typographic	Usage
Bold	• **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and path names. • **Bold** words also indicate the first use of a term included in the glossary.
Italic	• *Italic* words or characters represent variable values that you must supply. • *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices. (In other words, it means "or.")
< >	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, **<Enter>** refers to the key on your terminal or workstation that is labeled with the word Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
<Ctrl-*x*>	The notation **<Ctrl-*x*>** indicates a control character sequence. For example, **<Ctrl-c>** means that you hold down the control key while pressing **<c>**.

Related Information

In addition to this publication, the following books are also part of the LoadLeveler library:

- *Diagnosis and Messages Guide* , 86 A2 13EF
- *Installation Memo* , 86 A2 12EF

Information Formats

Documentation supporting RS/6000 SP software licensed programs is no longer available from IBM in hardcopy format. However, you can view, search, and print documentation in the following ways:

- On the World Wide Web
- Online (from the product media or the SP Resource Center)

Accessing This Book off the World Wide Web

You can view or download this book (in PDF format) from the World Wide Web using the following URL:

http://www.rs6000.ibm.com/resource/aix_resource/sp_books/loadleveler

Accessing LoadLeveler Documentation Online

IBM ships on the product media manual pages, HTML files, and PDF files. In order to use these files you must install the appropriate file sets. For more information, see *LoadLeveler Installation Memo* , which is shipped on the product media.

To view the LoadLeveler books in HTML format, you need access to an HTML document browser such as Netscape. Once you install the HTML files, an index to the LoadLeveler books is found in **/usr/lpp/LoadL/html/index.html**.

You can also view the LoadLeveler books from the SP Resource Center, which is available under the Parallel Systems Support Programs (PSSP) or as a separately installed program. You invoke the Resource Center from PSSP by entering **resource_center**. To invoke the Resource Center from the product CD, see the **readme.txt** file.

To view the LoadLeveler books in PDF format, you need access to the Adobe Acrobat Reader 3.0.1 or higher. The Acrobat Reader is shipped with AIX Version 4.3 Bonus Pack and is also freely available for downloading from the Adobe web site at URL **http://www.adobe.com**.

LoadLeveler Man Pages

Manual (man) pages are available for all LoadLeveler commands. You can view the man page for a command by entering **man** and the command name. For example: **man llq**.

The following man pages associated with LoadLeveler APIs (Application Programming Interfaces) are also available to you. You can view these man pages by entering **man** and the name of the man page. For example: **man LoadL_submitapi**.

Man Page Name	What it Describes
LoadL_acctapi	Accounting API
LoadL_ckptapi	Serial Checkpointing API
LoadL_dataapi	Data Access API
LoadL_jobctlapi	Workload Management API
LoadL_parallelapi	Parallel job API
LoadL_queryapi	Query API
LoadL_submitapi	Submit API

What's New in 2.2

The following is a list of new functions added for this release.

gsmonitor Daemon

A new daemon, **gsmonitor**, has been added to monitor machine availability and notify the negotiator when a machine is no longer reachable. For more details, see "The gsmonitor Daemon" on page 18.

Additional Job States

Two new LoadLeveler job states have been added:

- **Cancelled**
- **Terminated**

For more information, see "LoadLeveler Job States" on page 18.

New Job Command File Keywords

Three new job command file keywords have been added:

- **blocking**
- **resources**
- **task_geometry**

See "Job Command File Keywords" on page 36 for more information.

Consumable Resources

Consumable resources allow users to schedule jobs based on the availability of specific resources. For information on how consumable resources is used in the administration file, see "Step 1: Specify Machine Stanzas" on page 75, and "Step 3: Specify Class Stanzas" on page 84. For information about configuring consumable resources, see"Step 4: Define Consumable Resources" on page 104. For information about how consumable resources is used in job command files, see "resources" on page 52, and "Submitting and Managing Jobs that Consume Resources" on page 58.

Task Assignment Section

A new section has been added discussing the assignment of tasks to nodes. For more information see "Task Assignment Considerations" on page 60.

New Adapter Stanza Keyword

A new keyword called **css_type** has been added to the format of an adapter stanza. **css_type** designates the type of switch adapter to be used.

For more information, see "Step 5: Specify Adapter Stanzas" on page 95.

New Machine Stanza Keyword

A new keyword call **schedd_fenced** has been added to the format of a machine stanza. This keyword specifies that the central manager ignores connections from the schedd_daemon running on any machine specifying this keyword. For more information, see 80.

Process Tracking

This new function cancels any processes (throughout the entire cluster) left behind when a job terminates. For more information, see “Step 15: Specify Process Tracking” on page 122.

llctl Command Enhancement

The **purgeschedd** keyword requests that all jobs scheduled by the a specified host machine be purged. For more information, see 176.

Enhanced Support for DCE

LoadLeveler now fully supports DCE security features. Key features of DCE include the ability to authenticate users' identities, authorize users and programs to use LoadLeveler's services, and delegate user credentials to the starter process. For more infomation on enabling DCE, see “Step 16: Configuring LoadLeveler to use DCE Security Services” on page 123. A new command, **lldcegrpmaint**, is provided for setting up DCE groups and principal names. For more information on this command, see “lldcegrpmaint - LoadLeveler DCE group Maintenance Utility” on page 180.

llstatus Command Enhancements

The **llstatus** command now includes options for listing consumable resources. The new options are:

- **-R**, which lists consumable machine resources, and
- **-F**, which lists consumable floating resources

The **-l** option now lists: windows, memory, and connectivity for adapters; the switch fabric connectivity vector; information about free memory and paging, and consumable resource availability and use. For more information on the command, see “llstatus - Query Machine Status” on page 205.

llq Command Enhancements

Device memory for parallel jobs has been added to the Allocated Hosts and Task Instances lists in the **llq -l** output.

Task Guide

Task Guide support has been added for setting up consumable resources, tuning scheduling and communications keywords, and merging SP machine and adapter information. For more information, see 246.

Job Control API Renamed

The **Job Control** API has been renamed in this release. This API is now called the **Workload Management** API and adds a new subroutine, **ll_control**. For more details, see “Workload Management API” on page 283.

Scaling Considerations

Information on running LoadLeveler on a large system configuration and ways to reduce network traffic has been included in a new section, “Scaling Considerations” on page 312. Also, the **SCHEDD_SUBMIT_AFFINITY** configuration file keyword has been added; for more information on this keyword, see 103.

Migration Considerations

This section describes some differences between LoadLeveler 1.3.0 and LoadLeveler 2.1.0., and between LoadLeveler 2.1 and LoadLeveler 2.2. The *LoadLeveler Installation Memo* has more specific information about and procedures for migration.

Moving From 1.3 to 2.1

Resource Manager Functions Now in LoadLeveler

The following functions were previously part of the Parallel System Support Programs (PSSP) Resource Manager and are now part of LoadLeveler.

Table 1. New LoadLeveler Functions Previously Part of the Resource Manager

Resource Manager Function	LoadLeveler Function
Support for pools	The **pool_list** keyword in the machine stanza.
Specifying batch, interactive, or general use for nodes	The **machine_mode** keyword in the machine stanza.
Enabling SP exclusive use accounting for parallel jobs	The **sp_excluse_enable** keyword in the machine stanza.
Controlling user logins	LoadLeveler does not directly interact with the Login Control Facility. LoadLeveler logs into nodes as root and switches to the user's ID. root is never blocked on a node.
Providing node and adapter information for SP nodes	The **llextSDR** extracts information from the SDR that you can use in administration file stanzas.
Requesting dedicated use of nodes	The **node_usage** keyword in the job command file.
Requesting dedicated use of adapters	The *usage* operand on the **network** keyword in the job command file.
Displaying job information with the **jm_status -j** command	The **llq** command.
Displaying pool information with the **jm_status -P** command	The **llstatus -l** command.

Also, the LoadLeveler **rm_host** keyword in the machine stanza is no longer needed.

Keywords Supported for Parallel Jobs

LoadLeveler 2.1.0 includes a new scheduler, the Backfill scheduler, in addition to the default scheduler which existed in LoadLeveler 1.3.0. See Table 4 on page 59 for a list of which keywords associated with parallel jobs are supported by each scheduler.

Migrating Your Existing Adapter Requirements Statements

If you are running the Backfill scheduler with **job_type=parallel**, you should use the 2.1.0 **network** job command file keyword to request adapters. However, if you use the 1.3.0 **Adapter** requirement in a job command file, the requirement is converted to the 2.1.0 **network** statement. Only those requirement statements with one **Adapter** keyword and that use the "==" operator are converted; all other **Adapter** requirements are not allowed.

Table 2 shows how the network type value in an **Adapter** requirement statement is converted. The left column represents the network types you can request using the **Adapter** requirement. The right hand column represents the resulting **network** statement generated by LoadLeveler 2.1.0.

Table 2. How the Backfill Scheduler Handles the Adapter Requirement

Network Type Adapter Requirement	Resulting network Statement
hps_ip	css0,shared,IP
hps_user	css0,shared,US
ethernet	en0,shared,IP
fddi	fi0,shared,IP
tokenring	tr0,shared,IP
fcs	fcs0,shared,IP

Note that any adapter name in a resulting network statement must be specified in the administration file.

Changes in LoadLeveler Command Output

The following are changes in the output produced by LoadLeveler Version 2 Release 1 commands:

- The **llq -l -x** command output now includes task and node information for parallel jobs. For more information, see “Results” on page 195.
- The **llstatus -l** command output includes the following changes:
  - The order of the output fields displayed has changed.
  - The first and last line of output has changed.
  - Job classes are now grouped together and are followed by the number of class instances. For example, **small(2) POE(3)** refers to two small class jobs and three POE class jobs.
  - The Adapter line now contains expanded information.

  For more information, see “Results” on page 206.

Changes in the LoadLeveler Release Directory

The LoadLeveler release directory has changed as follows:

- **/usr/lpp/LoadL/nfs** is now **/usr/lpp/LoadL/full**.
- **/usr/lpp/LoadL/nfs_so** is now **/usr/lpp/LoadL/so**.

The LoadLeveler release directory is set by the **RELEASE_DIR** keyword in the sample LoadLeveler configuration file and the sample program Makefiles.

Changes in the GUI Resource File

The following new resources have been added to **Xloadl**, the GUI resource file:

- New resources ending in **_label** allow you to specify the titles of the columns on the Jobs and Machines windows.
- Additional resources ending in **_len** allow you to add new fields to the Jobs and Machines windows and to specify the size of these fields.
- New resources are available for “widgets,” such as the Job Type cascading menu, and the Nodes, Network, and PVM buttons and windows.

For more information, see **/usr/lpp/LoadL/full/lib/Xloadl**, the GUI resource file.

Moving From 2.1 to 2.2

Keyword Added to Administration File

The **css_type** keyword has been added to adapter stanzas in the administration file. This keyword designates the type of switch adapter to be used; for more information, see “Step 5: Specify Adapter Stanzas” on page 95.

Changes in LoadLeveler Command Output

The **llstatus -l** output now lists: windows, memory, and connectivity for adapters; the switch fabric connectivity vector; information about free memory and paging, and consumable resource availability and use. For more information on the command, see “llstatus - Query Machine Status” on page 205.

Device memory for parallel jobs has been added to the Allocated Hosts and Task Instances lists in the **llq -l** output. For more information, see “llq - Query Job Status” on page 193.

Part 1. Overview of LoadLeveler

Chapter 1. What is LoadLeveler?

LoadLeveler is a job management system that allows users to run more jobs in less time by matching the jobs' processing needs with the available resources. LoadLeveler schedules jobs, and provides functions for building, submitting, and processing jobs quickly and efficiently in a dynamic environment.

Figure 1 shows the different environments to which LoadLeveler can schedule jobs. Together, these environments comprise the *LoadLeveler cluster*. An environment can include heterogeneous clusters, dedicated nodes, and the RISC System/6000 Scalable POWERparallel System (SP).

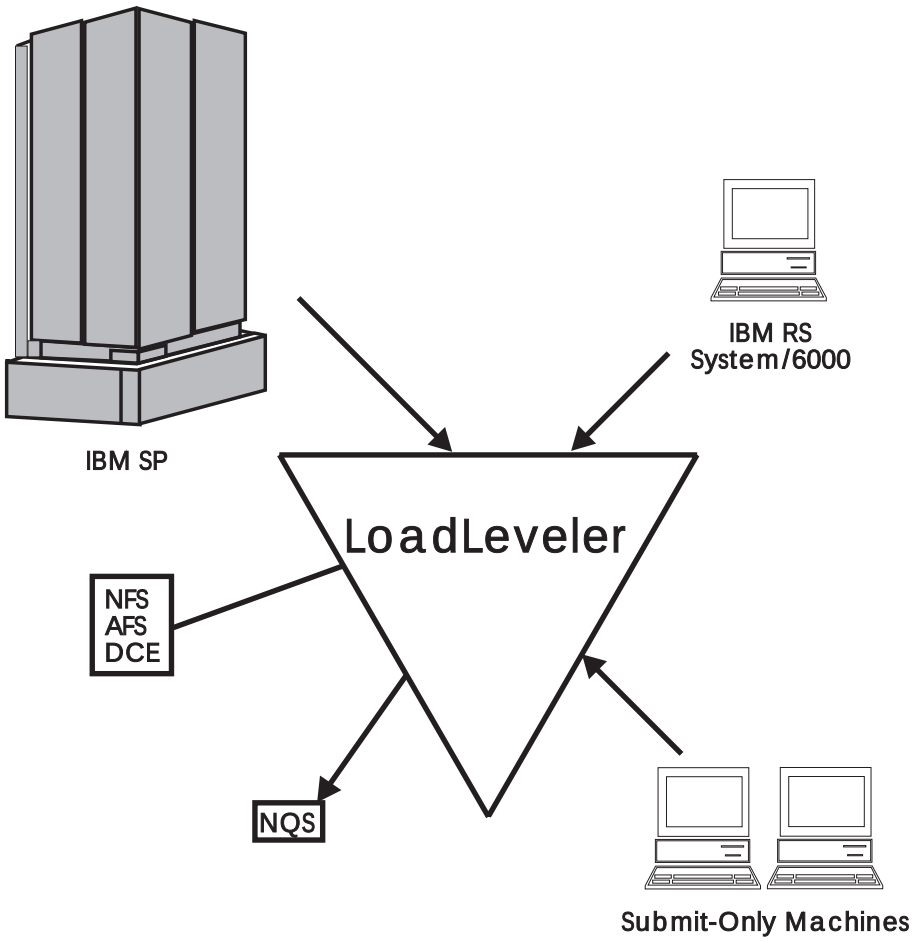

Figure 1. Example of a LoadLeveler Configuration

In addition, LoadLeveler can schedule jobs written for NQS to run on machines outside of the LoadLeveler cluster. As Figure 1 also illustrates, a LoadLeveler cluster can include *submit-only* machines, which allow users to have access to a limited number of LoadLeveler features. This type of machine is further discussed in "Roles of Machines" on page 5.

How LoadLeveler Works

This section introduces some basic job scheduling concepts.

What Does a Network Job Management and Job Scheduling System Do?

A network job management and job scheduling system, such as LoadLeveler, is a software program that schedules and manages jobs that you submit to one or more machines under its control. LoadLeveler accepts jobs that users submit and reviews the job requirements. LoadLeveler then examines the machines under its control to determine which machines are best suited to run each job.

Jobs

LoadLeveler schedules your jobs on one or more machines for processing. The definition of a *job*, in this context, is a set of *job steps*. For each job step, you can specify a different executable (the executable is the part of the job that gets processed). You can use LoadLeveler to submit jobs which are made up of one or more job steps, where each job step depends upon the completion status of a previous job step. For example, Figure 2 illustrates a stream of job steps:

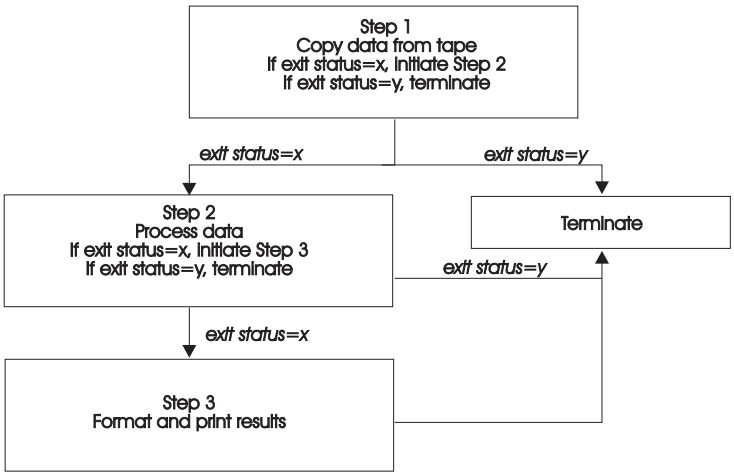

Figure 2. LoadLeveler Job Steps

Each of these job steps is defined in a single *job command file*. A job command file specifies the name of the job, as well as the job steps that you want to submit, and can contain other LoadLeveler statements.

LoadLeveler tries to execute each of your job steps on a machine that has enough resources to support executing and checkpointing each step. If your job command file has multiple job steps, the job steps will not necessarily run on the same machine, unless you explicitly request that they do.

You can submit batch jobs to LoadLeveler for scheduling. Batch jobs run in the background and generally do not require any input from the user. Batch jobs can either be *serial* or *parallel*. A serial job runs on a single machine. A parallel job is a program designed to execute as a number of individual, but related, processes on one or more of your system's nodes. When executed, these related processes can communicate with each other (through message passing or shared memory) to exchange data or synchronize their execution.

LoadLeveler will execute two different types of parallel jobs:

```
job_type = PVM
job_type = parallel
```

With a job_type of PVM, LoadLeveler supports a PVM API to allocate nodes and launch tasks. With a job_type of parallel, LoadLeveler interacts with Parallel Operating Environment (POE) to allocate nodes, assign tasks to nodes, and launch tasks.

Machines and Workstations

In order for LoadLeveler to schedule a job on a machine, the machine must be a valid member of the LoadLeveler cluster. A cluster is the combination of all of the different types of machines that use LoadLeveler. The following types of machines can comprise a LoadLeveler cluster:

- RISC System/6000 (and compatible hardware running AIX)
- SP System

To make a machine a member of the LoadLeveler cluster, the administrator has to install the LoadLeveler software onto the machine and identify the central manager (described in "Roles of Machines"). Once a machine becomes a valid member of the cluster, LoadLeveler can schedule jobs to it.

Roles of Machines: Each machine in the LoadLeveler cluster performs one or more roles in scheduling jobs. These roles are described below:

- *Scheduling Machine:* When a job is submitted, it gets placed in a queue managed by a scheduling machine. This machine contacts another machine that serves as the central manager for the entire LoadLeveler cluster. (This role is described below). This scheduling machine asks the central manager to find a machine that can run the job, and also keeps persistent information about the job. Some scheduling machines are known as *public scheduling machines*, meaning that any LoadLeveler user can access them. These machines schedule jobs submitted from submit-only machines, which are described below.
- *Central Manager Machine:* The role of the Central Manager is to examine the job's requirements and find one or more machines in the LoadLeveler cluster that will run the job. Once it finds the machine(s), it notifies the scheduling machine.
- *Executing Machine:* The machine that runs the job is known as the executing machine.
- *Submitting Machine:* This type of machine is known as a *submit-only* machine. It participates in the LoadLeveler cluster on a limited basis. Although the name implies that users of these machines can only submit jobs, they can also query and cancel jobs. Users of these machines also have their own Graphical User Interface (GUI) that provides them with the submit-only subset of functions. The submit-only machine feature allows workstations that are not part of the LoadLeveler cluster to submit jobs to the cluster.

Keep in mind that one machine can assume multiple roles.

Machine Availability: There may be times when some of the machines in the LoadLeveler cluster are not available to process jobs; for instance, when the owners of the machines have decided to make them unavailable. This ability of LoadLeveler to allow users to restrict the use of their machines provides flexibility and control over the resources.

Machine owners can make their personal workstations available to other LoadLeveler users in several ways. For example, you can specify that:

- The machine will always be available
- The machine will be available only between certain hours
- The machine will be available when the keyboard and mouse are not being used interactively.

Owners can also specify that their personal workstations never be made available to other LoadLeveler users.

LoadLeveler Daemons

This section lists the daemons that LoadLeveler uses to process jobs. For more detailed information, see “Daemons and Processes” on page 13.

LoadL_master
: Referred to as the **master** daemon, this daemon manages all LoadLeveler daemons on its machine. The master daemon runs on all machines in the cluster.

LoadL_schedd
: Referred to as the **schedd** daemon, this daemon manages a list of jobs submitted to the machine. The schedd daemon runs on all scheduling machines in the cluster.

LoadL_startd
: Referred to as the **startd** daemon, this daemon accepts jobs to be run on the machine where startd runs. The startd daemon runs on all executing machines in the cluster.

LoadL_starter
: Spawned by the startd daemon, the **starter** process manages a running job on the executing machine. The starter process runs on all executing machines in the cluster.

LoadL_kbdd
: Referred to as the **keyboard** daemon, this daemon monitors keyboard and mouse activity. The keyboard daemon runs on all executing machines in the cluster.

LoadL_negotiator
: Referred to as the **negotiator** daemon, this daemon collects job status and machine status from all machines in the LoadLeveler cluster, and makes decisions on where the jobs should be run. The negotiator daemon runs on the LoadLeveler central manager machine.

LoadL_GSmonitor
: Referred to as the gsmonitor daemon, this daemon uses the Group Services Application Programming Interface (GSAPI) for monitoring machine availability, and then notifies the negotiator when a machine is no longer reachable. The negotiator will then take the necessary action to remove running job(s) and mark the machine down.

How Does LoadLeveler Schedule Jobs to Run on Machines?

When a user submits a job, LoadLeveler examines the job command file to determine what resources the job will need. LoadLeveler determines which machine, or group of machines, is best suited to provide these resources, then LoadLeveler dispatches the job to the appropriate machine(s). To aid this process, LoadLeveler uses *queues*. A *job queue* is a list of jobs that are waiting to be processed. When a user submits a job to LoadLeveler, the job is entered into an internal database–which resides on one of the machines in the LoadLeveler cluster–until it is ready to be dispatched to run on another machine, as shown in Figure 3 on page 7.

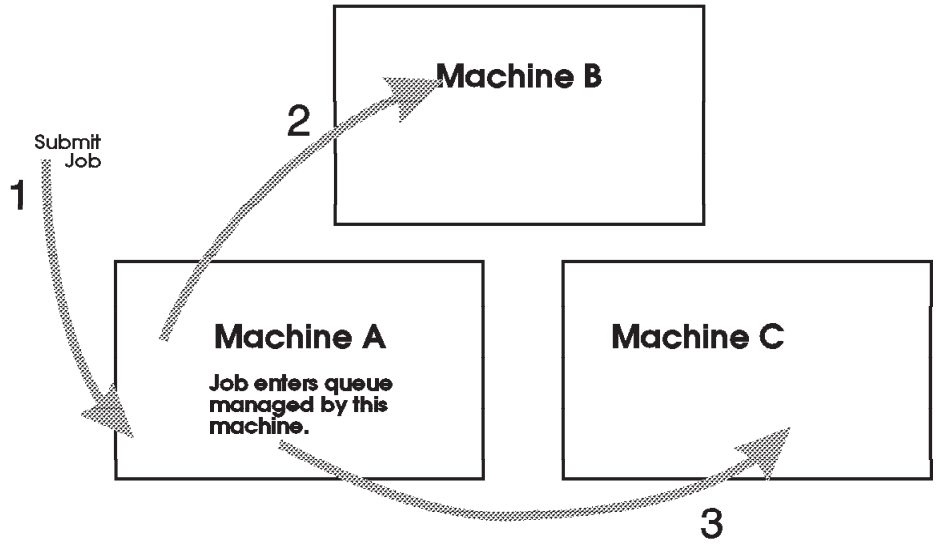

Figure 3. Job Queues

Once LoadLeveler examines a job to determine its required resources, the job is dispatched to a machine to be processed. Arrows 2 and 3 indicate that the job can be dispatched to either one machine, or–in the case of parallel jobs–to multiple machines. Once the job reaches the executing machine, the job runs.

Jobs do not necessarily get dispatched to machines in the cluster on a first-come, first-serve basis. Instead, LoadLeveler examines the requirements and characteristics of the job and the availability of machines, and then determines the best time for the job to be dispatched.

LoadLeveler also uses *job classes* to schedule jobs to run on machines. A job class is a classification to which a job can belong. For example, short running jobs may belong to a job class called `short_jobs`. Similarly, jobs that are only allowed to run on the weekends may belong to a class called `weekend`. The system administrator can define these job classes and select the users that are authorized to submit jobs of these classes. For more information on job classes, see “Step 3: Specify Class Stanzas” on page 84.

You can specify which types of jobs will run on a machine by specifying the type(s) of job classes the machine will support. For more information, see “Step 1: Specify Machine Stanzas” on page 75.

LoadLeveler also examines a job’s *priority* in order to determine when to schedule the job on a machine. A priority of a job is used to determine its position among a list of all jobs waiting to be dispatched. For more information on job priority, see “Setting and Changing the Priority of a Job” on page 28.

The LoadLeveler Job Cycle

Figure 4 on page 8 illustrates the information flow through the LoadLeveler cluster:

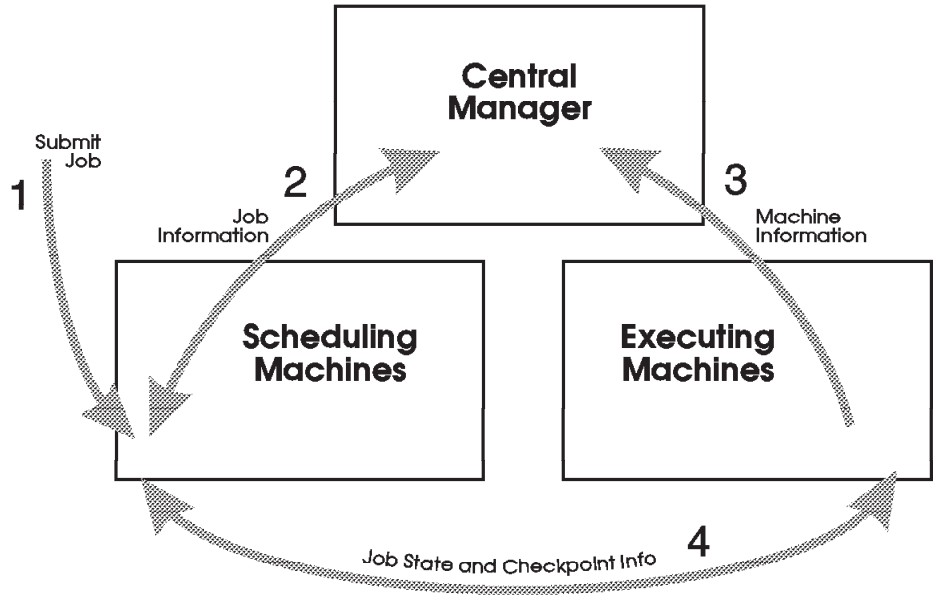

Figure 4. High-Level Job Flow

The managing machine in a LoadLeveler cluster is known as the **central manager**. There are alos machines that act as schedulers, and machines and machines that serve as the executing machines. The arrows in Figure 4 illustrate the following:

- Arrow 1 indicates that a job has been submitted to LoadLeveler.
- Arrow 2 indicates that the scheduling machine contacts the central manager to inform it that a job has been submitted, and to find out if a machine exists that matches the job requirements.
- Arrow 3 indicates that the central manager checks to determine if a machine exists that is capable of running the job. Once a machine is found, the central manager informs the scheduling machine which machine is available.
- Arrow 4 indicates that the scheduling machine contacts the executing machine and provides it with information regarding the job.

Figure 4 is broken down into the following more detailed diagrams illustrating how LoadLeveler processes a job.

1. Submit a LoadLeveler job:

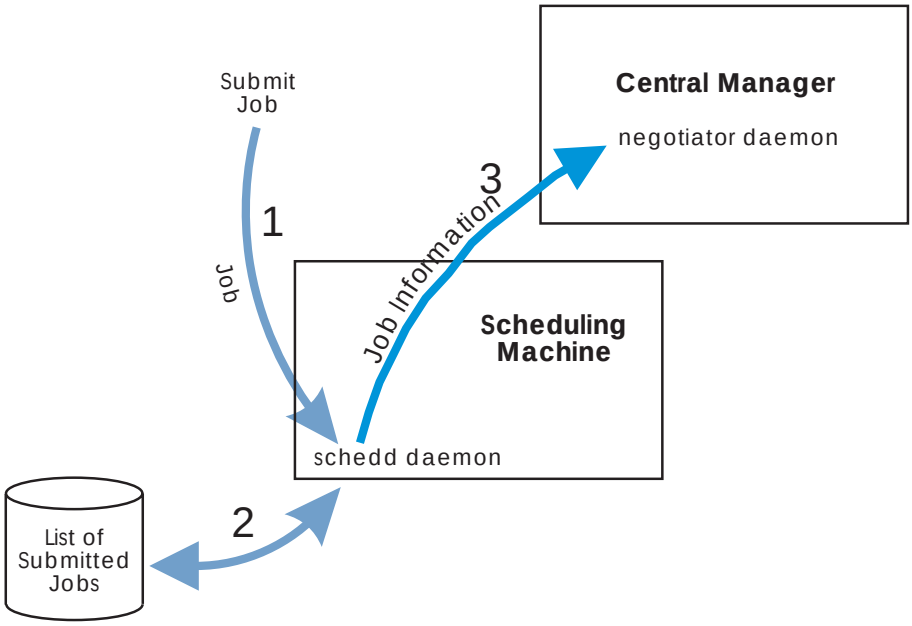

Figure 5. Job is Submitted to LoadLeveler

Figure 5 illustrates that the schedd daemon runs on the scheduling machine. This machine can also have the startd daemon running on it. The negotiator daemon resides on the central manager machine. The arrows in Figure 5 illustrate the following:

- Arrow 1 indicates that a job has been submitted to the scheduling machine.
- Arrow 2 indicates that the schedd daemon, on the scheduling machine, stores all of the relevant job information on local disk.
- Arrow 3 indicates that the schedd daemon sends job description information to the negotiator daemon.

2. Permit to run:

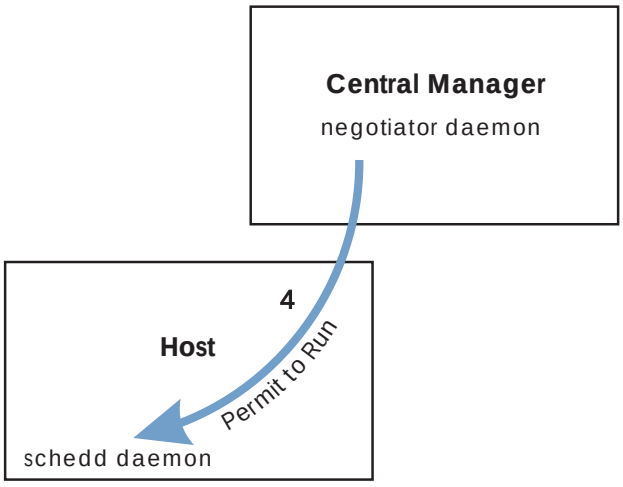

Figure 6. LoadLeveler Authorizes the Job

In Figure 6, arrow 4 indicates that the negotiator daemon authorizes the schedd daemon to begin taking steps to run the job. This authorization is called a

permit to run. Once this is done, the job is considered Pending or Starting. (See “LoadLeveler Job States” on page 18 for more information.)

3. Prepare to run:

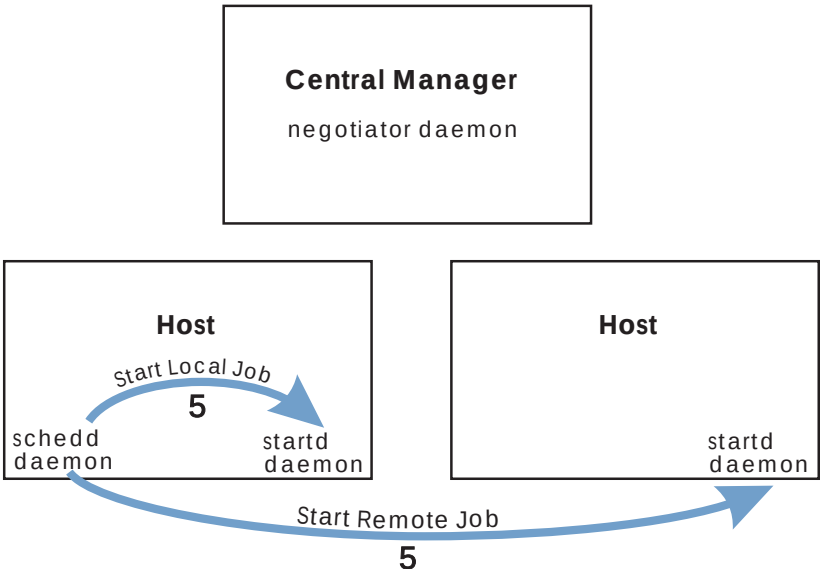

Figure 7. LoadLeveler Prepares to Run the Job

In Figure 7, arrow 5 illustrates that the schedd daemon contacts the startd daemon on the executing machine and requests that it start the job. The executing machine can either be a local machine (the machine from which the job was submitted) or a remote machine (another machine in the cluster).

4. Initiate job:

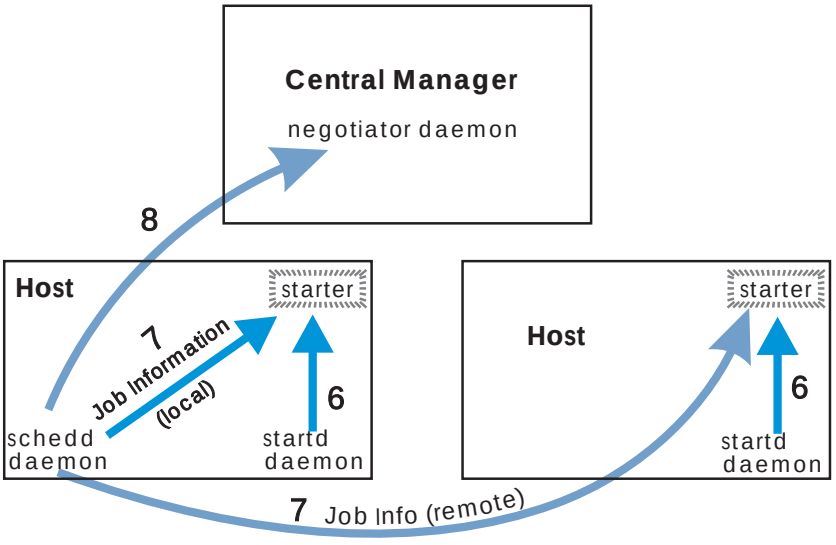

Figure 8. LoadLeveler Starts the Job

The arrows in Figure 8 illustrate the following:

- The two arrows numbered 6 indicate that the **startd** daemon on the executing machine, spawns a **starter** process and awaits more work.

- The two arrows numbered 7 indicate that the schedd daemon sends the starter process the job information and the executable.
- Arrow 8 indicates that the schedd daemon notifies the negotiator daemon that the job has been started and the negotiator daemon marks the job as Running. (See “LoadLeveler Job States” on page 18 for more information.)

The starter forks and executes the user’s job, and the starter parent waits for the child to complete.

5. Complete job:

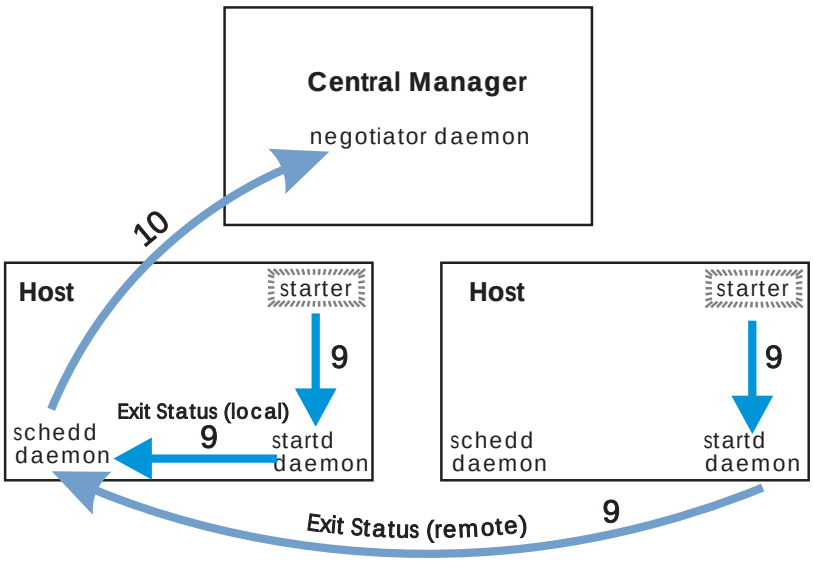

Figure 9. LoadLeveler Completes the Job

The arrows in Figure 9 illustrate the following:

- The arrows numbered 9 indicate that when the job completes, the starter process notifies the startd daemon, and the startd daemon notifies the schedd daemon.
- Arrow 10 indicates that the schedd daemon examines the information it has received and forwards it to the negotiator daemon.

What are Consumable Resources and Why Should I Use Them?

Consumable resources are resources available on machines in your LoadLeveler cluster. They are called ″resources″ because they model quantities of commodities or services available on machines (e.g., cpus, real memory, virtual memory, software licenses, DASD, etc). They are considered ″consumable″ because job steps use some specified amount of these commodities when they are running. Once the step is completed, the resource becomes available for reuse by another job step.

Consumable resources which model the characteristics of a specific machine (e.g., its number of cpus, or the number of a specific software licenses available only on that machine) are called machine resources. Consumable resources which model resources that are available across the LoadLeveler cluster (such as floating software licenses) are called floating resources. For example, consider a configuration with 10 licenses for a given program (which can be used on any

machine in the cluster). If these licenses are defined as floating resources, all 10 can be used on one machine, or they can be spread across as many as 10 different machines.

The LoadLeveler administrator can specify:

- the consumable resources to be considered by LoadLeveler's scheduling algorithms
- the quantity of resources available on specific machines
- the quantity of floating resources available on machines in the cluster
- the consumable resources to be considered in determining the priority of executing machines
- the default amount of resources consumed by a job step of a specified job class

The user submitting jobs can specify the resources consumed by each task of a job step.

The LoadLeveler scheduling algorithms use the availability of the requested consumable resources to determine the machine or machines on which a job will run. Consumable resources are used only for scheduling purposes and are not enforced like other limits, such as wall clock limits. Once a job is scheduled, LoadLeveler does not ensure that the amount of resources used is equal to the amount requested. LoadLeveler's negotiator daemon keeps track of the amounts of consumable resources available, reducing them by amounts requested when a job step is scheduled, and increasing them when a consuming job step completes.

LoadLeveler does not attempt to obtain software licenses or to verify that software licenses have been obtained, when consumable resources are used to model software licenses. By providing a user exit to be invoked as a submit filter, the LoadLeveler administrator may provide code to obtain a software license and run the job step only after a license has been successfully obtained. For more information on filtering job scripts, see "Filtering a Job Script" on page 296.

Chapter 2. LoadLeveler Daemons and Job States

This chapter presents a detailed explanation of LoadLeveler daemons and processes. Included here is a description of job states, which are controlled by certain daemons. See “LoadLeveler Job States” on page 18 for more information.

Daemons and Processes

This section presents a detailed explanation of LoadLeveler daemons and processes. For more information on configuration file keywords mentioned in this section, see “Configuring LoadLeveler” on page 97.

The master Daemon

The **master** daemon runs on every machine in the LoadLeveler cluster, except the submit-only machine. The real and effective user ID of this daemon must be root.

The master daemon determines whether to start any other daemons by checking the **START_DAEMONS** keyword in the global or local configuration file. If the keyword is set to **true**, the daemons are started. If the keyword is set to **false**, the master daemon terminates and generates a message.

On the machine designated as the central manager, the master runs the **negotiator** daemon. The master also controls the central manager backup function. The negotiator runs on either the primary or an alternate central manager. If a central manager failure is detected, one of the alternate central managers becomes the primary central manager by starting the negotiator.

The master daemon starts and if necessary, restarts all the LoadLeveler daemons that the machine it resides on is configured to run. As part of its startup procedure, this daemon executes the **.llrc** file (a dummy file is provided in the **bin** subdirectory of the release directory). You can use this script to customize your local configuration file, specifying what particular data is stored locally. This daemon also runs the **kbdd** daemon, which monitors keyboard and mouse activity.

When the master daemon detects a failure on one of the daemons that it is monitoring, it attempts to restart it. Because this daemon recognizes that certain situations may prevent a daemon from running, it limits its restart attempts to the number defined for the **RESTARTS_PER_HOUR** keyword in the configuration file. If this limit is exceeded, the master aborts and all daemons are killed.

When a daemon must be restarted, the master sends mail to the administrator(s) identified by the **LOADL_ADMIN** keyword in the configuration file. The mail contains the name of the failing daemon, its termination status, and a section of the daemon’s most recent log file. If the master aborts after exceeding **RESTARTS_PER_HOUR**, it will also send that mail before exiting.

The master daemon may perform the following actions in response to an **llctl** command:

- Kill all daemons and exit
- Kill all daemons and execute a new master
- Re-run the **.llrc** file, reread the configuration files, stop or start daemons as appropriate for the new configuration files
- Send drain request to startd and schedd
- Send flush request to startd and send result to caller

- Send suspend request to startd and send result to caller
- Send resume request to startd and schedd, and send result to caller

The schedd Daemon

The **schedd** daemon receives jobs sent by the **llsubmit** command and schedules those jobs to machines selected by the negotiator daemon. The schedd daemon is started, restarted, signalled, and stopped by the master daemon.

The schedd daemon can be in any one of the following states:

Available
This machine is available to schedule jobs.

Draining
The schedd daemon has been drained by the administrator but some jobs are still running. The state of the machine remains Draining until all running jobs complete. At that time, the machine status changes to Drained.

Drained
The schedd machine accepts no more jobs; jobs in the Starting or Running state are allowed to continue running, and jobs in the Idle state are drained, meaning they will not get dispatched.

Down The daemon is not running on this machine. The schedd daemon enters this state when it has not reported its status to the negotiator. This can occur when the machine is actually down, or because there is a network failure.

The schedd daemon performs the following functions:

- Assigns new job ids when requested by the job submission process (for example, by the **llsubmit** command).
- Receives new jobs from the **llsubmit** command. A new job is received as a *job object* for each job step. A job object is the data structure in memory containing all the information about a job step. The schedd forwards the job object to the negotiator daemon as soon as it is received from the submit command.
- Maintains on disk copies of jobs submitted locally (on this machine) that are either waiting or running on a remote (different) machine. The central manager can use this information to reconstruct the job information in the event of a failure. This information is also used for accounting purposes.
- Responds to directives sent by the administrator through the negotiator daemon. The directives include:
  - Run a job.
  - Change the priority of a job.
  - Remove a job.
  - Hold or release a job.
  - Send information about all jobs.
- Sends job events to the negotiator daemon when:
  - schedd is restarting.
  - A new series of job objects are arriving.
  - A job is started.
  - A job was rejected, completed, removed, or vacated. schedd determines the status by examining the exit status returned by the startd.
- Communicates with the Parallel Operating Environment (POE) when you run a POE job.
- Requests that a remote startd daemon kill a job.

- Handles the checkpoint file associated with the job, provided checkpointing has been enabled. For more information, see “Step 14: Enable Checkpointing” on page 117.
- Receives accounting information from startd.

The startd Daemon

The **startd** daemon monitors jobs and machine resources on the local machine and forwards this information to the negotiator daemon. The startd also receives and executes job requests originating from remote machines. The master daemon starts, restarts, signals, and stops the startd daemon.

The startd daemon can be in any one of the following states:

Busy The maximum number of jobs are running on this machine.

Down The daemon is not running on this machine. The startd daemon enters this state when it has not reported its status to the negotiator. This can occur when the machine is actually down, or because there is a network failure.

Drained
The startd machine will not accept any new jobs. However, any jobs that are already running on the startd machine will be allowed to complete.

Draining
The startd daemon has been drained by the administrator, but some jobs are still running. The machine remains in the draining state until all of the running jobs have completed, at which time the machine status changes to drained. The startd daemon will not accept any new jobs while in the draining state.

Flush Any running jobs have been vacated (terminated and returned to the queue to be redispatched). The startd daemon will not accept any new jobs.

Idle The machine is not running any jobs.

None LoadLeveler is running on this machine, but no jobs can run here.

Running
The machine is running one or more jobs and is capable of running more.

Suspend
All LoadLeveler jobs running on this machine are stopped (cease processing), but remain in virtual memory. The startd daemon will not accept any new jobs.

The startd daemon performs these functions:

- Runs a timeout procedure that includes building a snapshot of the state of the machine that includes static and dynamic data. This timeout procedure is run at the following times:
  - After a job completes.
  - According to the definition of the **POLLING_FREQUENCY** keyword in the configuration file.
- Records the following information in LoadLeveler variables and sends the information to the negotiator. These variables are described in “LoadLeveler Variables” on page 132.
  - State (of the startd daemon)
  - EnteredCurrentState
  - Memory
  - Disk

  - KeyboardIdle
  - Cpus
  - LoadAvg
  - Machine
  - Adapter
  - AvailableClasses
- Calculates the SUSPEND, RESUME, CONTINUE, and VACATE expressions. These are described in “Step 8: Manage a Job’s Status Using Control Expressions” on page 109.
- Receives job requests from the schedd daemon to:
  - Start a job
  - Vacate a job
  - Cancel

  When the schedd daemon tells the startd to start a job, the startd determines whether its own state permits a new job to run:

If:	Then this happens:
Yes, it can start a new job	The startd forks a **starter** process.
No, it cannot start a new job	The startd rejects the request for one of the following reasons: – Jobs have been suspended, flushed, or drained – The job limit set for the **MAX_STARTERS** keyword has been reached – There are not enough classes available for the designated job class

- Receives requests from the master (via **llctl**) to do one of the following:
  - Drain
  - Flush
  - Suspend
  - Resume.
- For each request, startd marks its own new state, forwards its new state to the negotiator daemon, and then performs the appropriate action for any jobs that are active.
- Receives notification of keyboard and mouse activity from the kbdd daemon
- Periodically examines the process table for LoadLeveler jobs and accumulates resources consumed by those jobs. This resource data is used to determine if a job has exceeded its job limit and for recording in the history file.
- Send accounting information to schedd.

The starter Process

The startd daemon spawns a **starter** process after the schedd daemon tells the startd to start a job. The starter process manages all the processes associated with a job step. The starter process is responsible for running the job and reporting status back to startd.

The starter process performs these functions:

- Processes the prolog and epilog programs as defined by the **JOB_PROLOG** and **JOB_EPILOG** keywords in the configuration file. The job will not run if the prolog program exits with a return code other than zero.
- Handles authentication. This includes:
  - Authenticates AFS, if necessary

  - Verifies that the submitting user is *not* root
  - Verifies that the submitting user has access to the appropriate directories in the local file system.
- Runs the job by forking a child process that runs with the user id and all groups of the submitting user. The starter child creates a new process group of which it is the process group leader, and executes the user's program or a shell. The starter parent is responsible for detecting the termination of the starter child. LoadLeveler does not monitor the children of the parent.
- Responds to vacate and suspend orders from the startd.
- Periodically generates a new checkpoint file, provided checkpointing has been enabled, and sends it to the scheduling machine.

The negotiator Daemon

The **negotiator** daemon maintains status of each job and machine in the cluster and responds to queries from the **llstatus** and **llq** commands. The negotiator daemon runs on a single machine in the cluster (the central manager machine). This daemon is started, restarted, signalled, and stopped by the master daemon.

The negotiator daemon receives status messages from each schedd and startd daemon running in the cluster. The negotiator daemon tracks:
- Which schedd daemons are running
- Which startd daemons are running, and the status of each startd machine.

If the gsmonitor daemon does not send the negotiator an update about a machine within the time period defined by the **MACHINE_UPDATE_INTERVAL** keyword, then the negotiator assumes that the machine is down, and therefore the schedd and startd daemons are also down.

The negotiator also maintains in its memory several queues and tables which determine where the job should run.

The negotiator performs the following functions:
- Receives and records job status changes from the schedd daemon.
- Schedules jobs based on a variety of scheduling criteria and policy options. Once a job is selected, the negotiator contacts the schedd that originally created the job.
- Handles requests to:
  - Set priorities
  - Query about jobs
  - Remove a job
  - Hold or release a job
  - Favor or unfavor a user or a job.
- Receives notification of schedd resets indicating that a schedd has restarted.

The kbdd Daemon

The **kbdd** daemon monitors keyboard and mouse activity. The kbdd daemon is spawned by the master daemon if the **X_RUNS_HERE** keyword in the configuration file is set to **true**.

The kbdd daemon notifies the startd daemon when it detects keyboard or mouse activity; however, kbdd is *not* interrupt driven. It sleeps for the number of seconds defined by the **POLLING_FREQUENCY** keyword in the LoadLeveler configuration file, and then determines if X events, in the form of mouse or keyboard activity,

have occurred. For more information on the configuration file, see “Chapter 5. Administering and Configuring LoadLeveler” on page 71.

The gsmonitor Daemon

The negotiator daemon monitors for down machines based on the heartbeat responses of the **MACHINE_UPDATE_INTERVAL** time period. If the negotiator has not received an update after two **MACHINE_UPDATE_INTERVAL** periods, then it marks the machine as down, and notifies the schedd to remove any jobs running on that machine. The gsmonitor daemon (LoadL_GSmonitor) allows this cleanup to occur more reliably. The gsmonitor daemon uses the Group Services Application Programming Interface (GSAPI) to monitor machine availability and notify the negotiator quickly when a machine is no longer reachable. Because it uses the GSAPI, the gsmonitor daemon requires that the Group Services subsytem, which is provided by the IBM Parallel System Support Programs (PSSP), be installed and operational.

The gsmonitor daemon should be run on one or two nodes in each of the Group Services domains. By running LoadL_GSmonitor on two nodes, this allows for a backup in case one of the nodes goes down. A Group Services domain consists of the set of nodes that makes up a system partition. LoadL_GSmonitor subscribes to the Group Services system-defined host membership group, which is represented by the **HA_GS_HOST_MEMBERSHIP** Group Services keyword. This group monitors every configured node in the system, including those that are not in the LoadLeveler cluster.

To start the gsmonitor daemon, set **GSMONITOR_RUNS_HERE** to True in the local config file. The default for **GSMONITOR_RUNS_HERE** is False.

Notes:

The Group Services routines need to be run as root, so the LoadL_GSmonitor executable must be owned by root and have the setuid permission bit enabled.

It will not cause a problem to run more than one LoadL_GSmonitor daemon per SP System Partition, this will just cause the negotiator to be notified by each running daemon.

For more information about the Group Services subsystem, see *PSSP: Administration Guide*, SA22-7348. For more information about GSAPI, see *Group Services Programming Guide and Reference*, SA22-7355-00.

LoadLeveler Job States

As LoadLeveler processes a job, the job moves into various states. Some states are unique to specific daemons; for example, only the negotiator places a job in the NotQueued state. For more information on daemons, see “Daemons and Processes” on page 13. Possible job states are:

Cancelled
: The job was cancelled either by a user or by an administrator.

Completed
: The job has completed.

Deferred
: The job will not be assigned to a machine until a specified date. This date may have been specified by the user in the job command file, or may have

been generated by the negotiator because a parallel job did not accumulate enough machines to run the job. (Only the negotiator places a job in the Deferred state.)

Idle The job is being considered to run on a machine, though no machine has been selected.

NotQueued
The job is not being considered to run on a machine. A job can enter this state because the associated schedd is down, the user or group associated with the job is at its maximum **maxqueued** or **maxidle** value, or because the job has a dependency which cannot be determined. For more information on these keywords, see “Controlling the Mix of Idle and Running Jobs” on page 314. (Only the negotiator places a job in the NotQueued state.)

Not Run
The job will never be run because a dependency associated with the job was found to be false.

Pending
The job is in the process of starting on one or more machines. (The negotiator indicates this state until the schedd acknowleges that it has received the request to start the job. Then the negotiator changes the state of the job to Starting. The schedd indicates the Pending state until all startd machines have acknowledged receipt of the start request. The schedd then changes the state of the job to Starting.)

Reject Pending
The job did not start. Possible reasons why a job is rejected are: job requirements were not met on the target machine, or the user ID of the person running the job is not valid on the target machine. After a job leaves the Reject Pending state, it is moved into one of the following states: Idle, User Hold, or Removed.

Removed
The job was stopped by LoadLeveler.

Remove Pending
The job is in the process of being removed, but not all associated machines have acknowledged the removal of the job.

Running
The job is running: the job was dispatched and has started on the designated machine.

Starting
The job is starting: the job was dispatched, was received by the target machine, and LoadLeveler is setting up the environment in which to run the job. For a parallel job, LoadLeveler sets up the environment on all required nodes. See the description of the “Pending” state for more information on when the negotiator or the schedd daemon moves a job into the Starting state.

System Hold
The job has been put in system hold.

System User Hold
The job has been put in system hold and user hold.

Terminated
If the negotiator and schedd daemons experience communication problems,

they may be temporarily unable to exchange information concerning the status of jobs in the system. During this period of time, some of the jobs may actually complete and therefore be removed from the schedd's list of active jobs. When communication resumes between the two daemons, the negotiator will move such jobs to the Terminated state, where they will remain for a set period of time (specified by the NEGOTIATOR_REMOVE_COMPLETED keyword in the configuration file). When this time has passed, the negotiator will remove the jobs from its active list.

User Hold
The job has been put in user hold.

Vacated
The job started but did not complete. The negotiator will reschedule the job (provided the job is allowed to be rescheduled). Possible reasons why a job moves to the Vacated state are: the machine where the job was running was flushed, the VACATE expression in the configuration file evaluated to True, or LoadLeveler detected a condition indicating the job needed to be vacated. For more information on the VACATE expression, see "Step 8: Manage a Job's Status Using Control Expressions" on page 109.

You may also see other states that include "Pending," such as Complete Pending and Vacate Pending. These are intermediate, temporary states usually associated with parallel jobs.

Part 2. Using LoadLeveler

Chapter 3. Submitting and Managing Jobs

This chapter tells you how to submit jobs to LoadLeveler. In general, the information in this chapter applies both to serial jobs and to parallel jobs. For more specific information on parallel jobs, see “Chapter 4. Submitting and Managing Parallel Jobs” on page 59.

Many LoadLeveler actions, such as submitting a job, can be done in either of the following ways:

- Using LoadLeveler commands. This method is discussed in this chapter.
- Using the LoadLeveler graphical user interface (GUI). This method is discussed in “Building and Submitting Jobs Using the Graphical User Interface” on page 225.

Building a Job Command File

Before you can submit a job or perform any other job related tasks, you need to build a job command file. A job command file describes the job you want to submit, and can include LoadLeveler keyword statements. For example, to specify a binary to be executed, you can use the **executable** keyword, which is described later in this section. To specify a shell script to be executed, the **executable** keyword can be used; if it is not used, LoadLeveler assumes that the job command file itself is the executable.

The job command file can include the following:

- LoadLeveler keyword statements: A *keyword* is a word that can appear in job command files. A *keyword statement* is a statement that begins with a LoadLeveler keyword. These keywords are described in “Job Command File Keywords” on page 36.
- Comment statements: You can use comments to document your job command files. You can add comment lines to the file as you would in a shell script.
- Shell command statements: If you use a shell script as the executable, the job command file can include shell commands.
- LoadLeveler Variables: See “Job Command File Variables” on page 56 for more information.

You can build a job command file either by using the Build a Job window on the GUI or by using a text editor.

Job Command File Syntax

The following general rules apply to job command files.

- Keyword statements begin with # @. There can be any number of blanks between the # and the @.
- Comments begin with #. Any line whose first non-blank character is a pound sign (#) and is not a LoadLeveler keyword statement is regarded as a comment.
- Statement components are separated by blanks. You can use blanks before or after other delimiters to improve readability but they are not required if another delimiter is used.
- The back-slash (\) is the line continuation character. Note that the continued line must not begin with # @. See Figure 15 on page 34 for an example of using the back-slash.

- Keywords are *not* case sensitive. This means you can enter them in lower case, upper case, or mixed case.

Serial Job Command File

Figure 10 is an example of a simple serial job command file which is run from the current working directory. The job command file reads the input file, **longjob.in1**, from the current working directory and writes standard output and standard error files, **longjob.out1** and **longjob.err1**, respectively, to the current working directory.

```
#   The name of this job command file is file.cmd.
#   The input file is longjob.in1 and the error file is
#   longjob.err1. The queue statement marks the end of
#   the job step.
#
# @ executable = longjob
# @ input = longjob.in1
# @ output = longjob.out1
# @ error = longjob.err1
# @ queue
```

Figure 10. Serial Job Command File

Using Multiple Steps in a Job Command File

To specify a stream of job steps, you need to list each job step in the job command file. You must specify one **queue** statement for each job step. Also, the executables for all job steps in the job command file must exist when you submit the job. All information in the first step is inherited by all succeeding steps.

LoadLeveler treats all job steps as independent job steps unless you use the **dependency** keyword. If you use the **dependency** keyword, LoadLeveler determines whether a job step should run based upon the exit status of the previously run job step.

For example, Figure 11 contains two separate job steps. Notice that `step1` is the first job step to run and that `step2` is a job step that runs only if `step1` exits with the correct exit status.

```
# This job command file lists two job steps called "step1"
# and "step2".  "step2" only runs if "step1" completes
# with exit status = 0.  Each job step requires a new
# queue statement.
#
# @ step_name = step1
# @ executable = executable1
# @ input = step1.in1
# @ output = step1.out1
# @ error = step2.err1
# @ queue
# @ dependency = (step1 == 0)
# @ step_name = step2
# @ executable = executable2
# @ input = step2.in1
# @ output = step2.out1
# @ error = step2.err1
# @ queue
```

Figure 11. Job Command File with Multiple Steps

In Figure 11 on page 24, `step1` is called the *sustaining* job step. `step2` is called the *dependent* job step because whether or not it begins to run is dependent upon the exit status of `step1`. A single sustaining job step can have more than one dependent job steps and a dependent job step can also have job steps dependent upon it.

In Figure 11 on page 24, each job step has its own **executable**, **input**, **output**, and **error** statements. Your job steps can have their own separate statements, or they can use those statements defined in a previous job step. For example, in Figure 12, `step2` uses the **executable** statement defined in `step1`:

```
# This job command file uses only one executable for
# both job steps.
#
# @ step_name = step1
# @ executable = executable1
# @ input = step1.in1
# @ output = step1.out1
# @ error = step1.err1
# @ queue
# @ dependency = (step1 == 0)
# @ step_name = step2
# @ input = step2.in1
# @ output = step2.out1
# @ error = step2.err1
# @ queue
```

Figure 12. Job Command File with Multiple Steps and One Executable

See “Additional Job Command File Examples” on page 32 for more information.

Parallel Job Command File

In addition to building job command files to submit serial jobs, you can also build job command files to submit parallel jobs. Before constructing parallel job command files, consult your LoadLeveler system administrator to see if your installation is configured for parallel batch job submission.

For more information on submitting parallel jobs, see “Chapter 4. Submitting and Managing Parallel Jobs” on page 59.

Submitting a Job Command File

After building a job command file, you can submit it for processing either to a machine in the LoadLeveler cluster or one outside of the cluster. (See “Querying Multiple LoadLeveler Clusters” on page 27 for information on submitting a job to a machine outside the cluster.) You can submit a job command file either by using the GUI or the **llsubmit** command.

When you submit a job, LoadLeveler assigns the job a three part identifier and also sets environment variables for the job.

The identifier consists of the following:

- Machine name: the name of the machine that schedules the job. This is not necessarily the name of the machine that runs the job.
- Job ID: an identifier given to a group of job steps that were initiated from the same job command file. For example, if you created a job command file that submitted the same program five times (using five queue statements) possibly with different input and output, each program would have the same job ID.

- Step ID: an identifier that is unique for every job step in the job you submit. If a job command file contains multiple job steps, every job step will have a unique step ID but the same job ID.

For an example of submitting a job, see “Step 3: Submit a Job” on page 30.

Submitting a Job Command File to be Routed to NQS Machines: When submitting a job command file to be routed to an NQS machine for processing, the job command file must contain the shell script to be submitted to the NQS node. NQS accepts only shell scripts; binaries are not allowed. All options in the command file pertaining to scheduling are used by LoadLeveler to schedule the job. When the job is dispatched to the node running the specified NQS class, the LoadLeveler options pertaining to the runtime environment are converted to NQS options and the job is submitted to the specified NQS queue. For more information on submitting jobs to NQS, see Figure 31 on page 159. For more information on the **llsubmit** command, see “llsubmit - Submit a Job” on page 213.

Submitting a Job Command File Using a Submit-Only Machine: You can submit jobs from submit-only machines. Submit-only machines allow machines that do not run LoadLeveler daemons to submit jobs to the cluster. You can submit a job using either the submit-only version of the GUI or the **llsubmit** command.

To install submit-only LoadLeveler, follow the procedure in the *LoadLeveler Installation Memo* , or consult the appropriate README file.

In addition to allowing you to submit jobs, the submit-only feature allows you to cancel and query jobs from a submit-only machine.

Managing Jobs

This sections tells you how to edit a job command file, query the status of a job, place and release a hold on a job, cancel a job, change the priority of a job, checkpoint a step, and display machine status.

Editing a Job Command File

After you build a job command file, you can edit it using the editor of your choice. You may want to change the name of the executable or add or delete some statements.

Querying the Status of a Job

Once you submit a job, you can query the status of the job to determine, for example, if it is still in the queue or if it is running. You also receive other job status related information such as the job ID and job owner. You can query the status of a LoadLeveler job either by using the GUI or the **llq** command. For an example of querying the status of a job, see “Step 4: Display the Status of a Job” on page 30.

Querying the Status of a Job Running on an NQS Machine: If your job command file was routed to an NQS machine for processing, you can obtain its status by using either the GUI or the **llq** command. Keep in mind that a machine in the LoadLeveler cluster monitors the NQS machine where your job is running. The status you see on the GUI (or from **llq**) is generated by the machine in the LoadLeveler cluster. Since LoadLeveler only checks the NQS machine for status periodically, the status of the job on the NQS machine may change before LoadLeveler has an opportunity to update the GUI. If this happens, NQS will notify you, before LoadLeveler notifies you, regarding the status of the job.

Querying the Status of a Job Using a Submit-Only Machine: A submit-only machine, in addition to allowing you to submit and cancel jobs, allows you to query the status of jobs. You can query a job using either the submit-only version of the GUI or by using the **llq** command. For information on **llq**, see “llq - Query Job Status” on page 193.

Querying Multiple LoadLeveler Clusters

This section applies only to those installations having more than one LoadLeveler cluster.

Using the **LOADL_CONFIG** environment variable, you can query, submit, or cancel jobs in multiple LoadLeveler clusters. The **LOADL_CONFIG** environment variable allows you to specify that the master configuration file be located in a directory other than the home directory of the **loadl** user ID. The file that **LOADL_CONFIG** points to must be in the **/etc** directory.

You need to set up your own master configuration file to point to the location of the LoadLeveler user ID, group ID, and configuration files. By default, the location of the master file is **/etc/LoadL.cfg**.

The following example explains how you can set up a machine to query multiple clusters:

You can configure **/etc/LoadL.cfg** to point to the “default” configuration files, and you can configure **/etc/othercluster.cfg** to point to the configuration files of another cluster which the user can select.

For example, you can enter the following query command:

```
$ llq
```

The above command uses the configuration from **/etc/LoadL.cfg** (this is determined by the **LOADL_CONFIG** environment variable). To send a query to the scheduler defined in the configuration file of **/etc/othercluster.cfg**, enter:

```
$ env LOADL_CONFIG=/etc/othercluster.cfg llq
```

Note that the machine from which you issue the **llq** command is considered as a submit-only machine by the other cluster.

Placing and Releasing a Hold on a Job

You may place a hold on a job and thereby cause the job to remain in the queue until you release it.

There are two types of holds: a user hold and a system hold. Both you and your LoadLeveler administrator can place and release a user hold on a job. Only a LoadLeveler administrator, however, can place and release a system hold on a job.

You can place a hold on a job or release the hold either by using the GUI or the **llhold** command. For examples of holding and releasing jobs, see “Step 6: Hold a Job” on page 31 and “Step 7: Release a Hold on a Job” on page 31.

As a user or an administrator, you can also use the **startdate** keyword described in “startdate” on page 53 to place a hold on a job. This keyword allows you to specify when you want to run a job.

Cancelling a Job

You can cancel one of your jobs that is either running or waiting to run by using either the GUI or the **llcancel** command. You can use **llcancel** to cancel LoadLeveler jobs and jobs routed to NQS. Note that you can also cancel jobs from a submit-only machine.

Checkpointing a Job

Checkpointing is a method of periodically saving the state of a job so that, if for some reason, the job does not complete, it can be restarted from the saved state. For a detailed explanation of checkpointing, see “Step 14: Enable Checkpointing” on page 117.

Setting and Changing the Priority of a Job

LoadLeveler uses the priority of a job to determine its position among a list of all jobs waiting to be dispatched. You can use the **llprio** command to change job priorities. See “llprio - Change the User Priority of Submitted Job Steps” on page 191 for more information. This section discusses the different types of priorities and how LoadLeveler uses these priorities when considering jobs for dispatch.

User Priority

Every job has a user priority associated with it. This priority, which can be specified by the user in the job command file, is a number between 0 and 100 inclusively. A job with a higher priority runs before a job with a lower priority (when both jobs are owned by the same user). The default user priority is 50. Note that this is not the UNIX *nice* priority.

System Priority

Every job has a system priority associated with it. This priority is specified in LoadLeveler’s configuration file using the **SYSPRIO** expression.

Understanding the SYSPRIO Expression: SYSPRIO is evaluated by LoadLeveler to determine the overall system priority of a job. A system priority value is assigned when the negotiator adds the new job to the queue of jobs eligible for dispatch.

The **SYSPRIO** expression can contain class, group, and user priorities, as shown in the following example:

```
SYSPRIO : (ClassSysprio * 100) + (UserSysprio * 10) + (GroupSysprio * 1) - (QDate)
```

For more information on the system priority expression, including all the variable you can use in this expression, see “Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105.

How Does a Job’s Priority Affect Dispatching Order?

LoadLeveler schedules jobs based on the *adjusted system priorty*, which takes in account both system priority and user priority. Jobs with a higher adjusted system priority are scheduled ahead of jobs with a lower adjusted system priority. In determining which jobs to run first, LoadLeveler does the following:

1. Assigns all jobs a SYSPRIO at job submission time.
2. Orders jobs first by SYSPRIO.
3. Assigns jobs belonging to the same user and the same class an adjusted system priority, which takes all the system priorities and orders them by user priority.

For example, Table 3 represents the priorities assigned to jobs submitted by two users, Rich and Joe. Two of the jobs belong to Joe, and three belong to Rich. User Joe has two jobs (Joe1 and Joe2) in Class A with SYSPRIOs of 9 and 8 respectively. Since Joe2 has the higher user priority (20), and because both of Joe's jobs are in the same class, Joe2's priority is swapped with that of Joe1 when the adjusted system priority is calculated. This results in Joe2 getting an adjusted system priority of 9, and Joe1 getting an adjusted system priority of 8. Similarly, the Class A jobs belonging to Rich (Rich1 and Rich3) also have their priorities swapped. The priority of the job Rich2 does not change, since this job is in a different class (Class B).

Table 3. How LoadLeveler Handles Job Priorities

Job	User Priority	System Priority (SYSPRIO)	Class	Adjusted System Priority
Rich1	50	10	A	6
Joe1	10	9	A	8
Joe2	20	8	A	9
Rich2	100	7	B	7
Rich3	90	6	A	10

Working with Machines

Throughout this book, the terms *workstation*, *machine*, and *node* refer to the machines in your cluster. See "Machines and Workstations" on page 5 for information on the roles these machines can play.

You can perform the following types of tasks related to machines:

- Display machine status: when you submit a job to a machine, the status of the machine automatically appears in the Machines window on the GUI. This window displays machine related information such as the names of the machines running jobs, as well as the machine's architecture and operating system. For detailed information on one or more machines in the cluster, you can use the Details option on the Actions pull-down menu. This will provide you with a detailed report that includes information such as the machine's state and amount of installed memory.

  For an example of displaying machine status, see "Step 8: Display the Status of a Machine" on page 31.
- Display central manager: the LoadLeveler administrator designates one of the machines in the LoadLeveler cluster as the central manager. When jobs are submitted to any machine, the central manager is notified and decides where to schedule the jobs. In addition, it keeps track of the status of machines in the cluster and jobs in the system by communicating with each machine. LoadLeveler uses this information to make the scheduling decisions and to respond to queries.

  Usually, the system administrator is more concerned about the location of the central manager than the typical end user but you may also want to determine its location. One reason why you might want to locate the central manager is if you want to browse some configuration files that are stored on the same machine as the central manager.
- Display public scheduling machines: public scheduling machines are machines that participate in the scheduling of LoadLeveler jobs on behalf of users at

submit-only machines and users at other workstations that are not running the schedd daemon. You can find out the names of all these machines in the cluster.

Submit-only machines allow machines that are not part of the LoadLeveler cluster to submit jobs to the cluster for processing.

A Simple Task Scenario Using Commands

The section presents a series of simple tasks which a user might perform using commands. This section is meant for new users of LoadLeveler. More experienced users may want to continue on to “Additional Job Command File Examples” on page 32.

Step 1: Build a Job

Since you are not using the GUI, you have to build your job command file by using a text editor to create a script file. Into the file enter the name of the executable, other keywords designating such things as output locations for messages, and the necessary LoadLeveler statements, as shown in Figure 13:

```
# This job command file is called longjob.cmd.  The
# executable is called longjob, the input file is longjob.in,
# the output file is longjob.out, and the error file is
# longjob.err.
#
# @ executable = longjob
# @ input      = longjob.in
# @ output     = longjob.out
# @ error      = longjob.err
# @ queue
```

Figure 13. Building a Job Command File

Step 2: Edit a Job

You can optionally edit the job command file you created in step 1.

Step 3: Submit a Job

To submit the job command file that you created in step 1, use the **llsubmit** command:

```
llsubmit longjob.cmd
```

LoadLeveler responds by issuing a message similar to:

```
submit: The job "wizard.22" has been submitted.
```

where `wizard` is the name of the machine to which the job was submitted and `22` is the job identifier (ID). You may want to record the identifier for future use (although you can obtain this information later if necessary).

For more information on **llsubmit**, see “llsubmit - Submit a Job” on page 213

Step 4: Display the Status of a Job

To display the status of the job you just submitted, use the **llq** command. This command returns information about all jobs in the LoadLeveler queue:

```
llq wizard.22
```

where `wizard` is the machine name to which you submitted the job, and `22` is the job ID. You can also query this job using the command **llq wizard.22.0**, where `0` is the step ID. For more information, see “llq - Query Job Status” on page 193.

Step 5: Change the Priorities of Jobs in the Queue

You can change the user priority of a job that is in the queue or one that is running. This only affects jobs belonging to the same user and the same class. If you change the priority of a job in the queue, the job’s priority increases or decreases in relation to your other jobs in the queue. If you change the priority of a job that is running, it does not affect the job while it is running. It only affects the job if the job re-enters the queue to be dispatched again. For more information, see “How Does a Job’s Priority Affect Dispatching Order?” on page 28.

To change the priority of a job, use the **llprio** command. To increase the priority of the job you submitted by a value of 10, enter:

```
llprio +10 wizard.22.0
```

For more information, see “llprio - Change the User Priority of Submitted Job Steps” on page 191.

Step 6: Hold a Job

To place a temporary hold on a job in a queue, use the **llhold** command. This command only takes effect if jobs are in the Idle or NotQueued state. To place a hold on `wizard.22.0`, enter:

```
llhold wizard.22.0
```

For more information, see “llhold - Hold or Release a Submitted Job” on page 187.

Step 7: Release a Hold on a Job

To release the hold you placed in step 6, use the **llhold** command:

```
llhold -r wizard.22.0
```

For more information, see “llhold - Hold or Release a Submitted Job” on page 187.

Step 8: Display the Status of a Machine

To display the status of the machine to which you submitted a job, use the **llstatus** command:

```
llstatus -l wizard
```

For more information, see “llstatus - Query Machine Status” on page 205.

Step 9: Cancel a Job

To cancel *wizard.22.0*, use the **llcancel** command:

```
llcancel wizard.22.0
```

For more information, see “llcancel - Cancel a Submitted Job” on page 170.

Step 10: Find the Location of the Central Manager

Enter the **llstatus** command with the appropriate options to display the machine on which the central manager is running. For more information, see “llstatus - Query Machine Status” on page 205.

Step 11: Find the Location of the Public Scheduling Machines

Public scheduling machines are those machines that participate in the scheduling of LoadLeveler jobs. The **llstatus** command can also be used to display the public scheduling machines.

Additional Job Command File Examples

"Serial Job Command File" on page 24 gives you an example of a simple job command file. This section contains examples of building and submitting more complex job command files.

Example 1: Generating Multiple Jobs With Varying Outputs

To run a program several times, varying the initial conditions each time, you could can multiple LoadLeveler scripts, each specifying a different input and output file as described in Figure 15 on page 34. It would probably be more convenient to prepare different input files and submit the job only once, letting LoadLeveler generate the output files and do the multiple submissions for you.

Figure 14 illustrates the following:

- You can refer to the LoadLeveler name of your job symbolically, using **$(jobid)** and **$(stepid)** in the LoadLeveler script file.
- **$(jobid)** refers to the job identifier.
- **$(stepid)** refers to the job step identifier and increases after each **queue** command. Therefore, you only need to specify input, output, and error statements once to have LoadLeveler name these files correctly.

Assume that you created five input files and each input file has different initial conditions for the program. The names of the input files are in the form **longjob.in.***x*, where *x* is 0–4.

Submitting the LoadLeveler script shown in Figure 14 results in your program running five times, each time with a different input file. LoadLeveler generates the output file from the LoadLeveler job step IDs. This ensures that the results from the different submissions are not merged.

```
# @ executable = longjob
# @ input = longjob.in.$(stepid)
# @ output = longjob.out.$(jobid).$(stepid)
# @ error = longjob.err.$(jobid).$(stepid)
# @ queue
# @ queue
# @ queue
# @ queue
# @ queue
```

Figure 14. Job Command File with Varying input Statements

To submit the job, type the command:

```
llsubmit longjob.cmd
```

LoadLeveler responds by issuing the following:

```
submit: The job "ll6.23" with 5 job steps has been submitted.
```

The following table shows you the standard input files, standard output files, and standard error files for the five job steps:

Job Step	Standard Input	Standard Output	Standard Error
ll6.23.0	longjob.in.0	longjob.out.23.0	longjob.err.23.0
ll6.23.1	longjob.in.1	longjob.out.23.1	longjob.err.23.1
ll6.23.2	longjob.in.2	longjob.out.23.2	longjob.err.23.2
ll6.23.3	longjob.in.3	longjob.out.23.3	longjob.err.23.3
ll6.23.4	longjob.in.4	longjob.out.23.4	longjob.err.23.4

Example 2: Using LoadLeveler Variables in a Job Command File

Figure 15 on page 34 shows how you can use LoadLeveler variables in a job command file to assign different names to input and output files. This example assumes the following:

- The name of the machine from which the job is submitted is `lltest1`
- The user's home directory is **/u/rhclark** and the current working directory is **/u/rhclark/OSL**
- LoadLeveler assigns a value of 122 to **$(jobid)**.

In Job Step 0:

- LoadLeveler creates the subdirectories **oslsslv_out** and **oslsslv_err** if they do not exist at the time the job step is started.

In Job Step 1:

- The character string **rhclark** denotes the home directory of user **rhclark** in **input**, **output**, **error**, and **executable** statements.
- The **$(base_executable)** variable is set to be the "base" portion of the **executable**, which is **oslsslv**.
- The **$(host)** variable is equivalent to **$(hostname)**. Similarly, **$(jobid)** and **$(stepid)** are equivalent to **$(cluster)** and **$(process)**, respectively.

In Job Step 2:

- This job step is executed only if the return codes from Step 0 and Step 1 are both equal to zero.
- The initial working directory for Step 2 is explicitly specified.

```
# Job step 0 ============================================================
#   The names of the output and error files created by this job step are:
#
#     output: /u/rhclark/OSL/oslsslv_out/lltest1.122.0.out
#     error : /u/rhclark/OSL/oslsslv_err/lltest1_122_0_err
#
# @ job_name = OSL
# @ step_name = step_0
# @ executable = oslsslv
# @ arguments = -maxmin=min -scale=yes -alg=dual
# @ environment = OSL_ENV1=20000; OSL_ENV2=500000
# @ requirements = (Arch == "R6000") && (OpSys == "AIX43")
# @ input  = test01.mps.$(stepid)
# @ output = $(executable)_out/$(host).$(jobid).$(stepid).out
# @ error  = $(executable)_err/$(host)_$(jobid)_$(stepid)_err
# @ queue
#
# Job step 1 ============================================================
#   The names of the output and error files created by this job step are:
#
#     output: /u/rhclark/OSL/oslsslv_out/lltest1.122.1.out
#     error : /u/rhclark/OSL/oslsslv_err/lltest1_122_1_err
#
# @ step_name = step_1
# @ executable = rhclark/$(job_name)/oslsslv
# @ arguments = -maxmin=max -scale=no -alg=primal
# @ environment = OSL_ENV1=60000; OSL_ENV2=500000; \
                  OSL_ENV3=70000; OSL_ENV4=800000;
# @ input  = rhclark/$(job_name)/test01.mps.$(stepid)
# @ output = rhclark/$(job_name)/$(base_executable)_out/$(hostname).$(cluster).$(process).out
# @ error  = rhclark/$(job_name)/$(base_executable)_err/$(hostname)_$(cluster)_$(process)_err
# @ queue
#
# Job step 2 ============================================================
#   The names of the output and error files created by this job step are:
#
#     output: /u/rhclark/OSL/oslsslv_out/lltest1.122.2.out
#     error : /u/rhclark/OSL/oslsslv_err/lltest1_122_2_err
#
# @ step_name = OSL
# @ dependency = (step_0 == 0) && (step_1 == 0)
# @ comment = oslsslv
# @ initialdir = /u/rhclark/$(step_name)
# @ arguments = -maxmin=min -scale=yes -alg=dual
# @ environment = OSL_ENV1=300000; OSL_ENV2=500000
# @ input  = test01.mps.$(stepid)
# @ output = $(comment)_out/$(host).$(jobid).$(stepid).out
# @ error  = $(comment)_err/$(host)_$(jobid)_$(stepid)_err
# @ queue
```

Figure 15. Using LoadLeveler Variables in a Job Command File

Example 3: Using the Job Command File as the Executable

The name of the sample script shown in Figure 16 on page 36 is **run_spice_job**. This script illustrates the following:

- The script does not contain the **executable** keyword. When you do not use this keyword, LoadLeveler assumes that the script is the executable. (Since the name of the script is **run_spice_job**, you can add the **executable = run_spice_job** statement to the script, but it is not necessary.)

- The job consists of four job steps (there are 4 **queue** statements). The **spice3f5** and **spice2g6** programs are invoked at each job step using different input data files:
  - **spice3f5:** Input for this program is from the file **spice3f5_input**_*x* where *x* has a value of 0, 1, and 2 for job steps 0, 1, and 2, respectively. The name of this file is passed as the first argument to the script. Standard output and standard error data generated by **spice3f5** are directed to the file **spice3f5_output**_*x*. The name of this file is passed as second argument to the script. In job step 3, the names of the input and output files are **spice3f5_input_benchmark1** and **spice3f5_output_benchmark1**, respectively.
  - **spice2g6:** Input for this program is from the file **spice2g6_input**_*x*. Standard output and standard error data generated by **spice2g6** together with all other standard output and standard error data generated by this script are directed to the files **spice_test_output**_*x* and **spice_test_error**_*x*, respectively. In job step 3, the name of the input file is **spice2g6_input_benchmark1**. The standard output and standard error files are **spice_test_output_benchmark1** and **spice_test_error_benchmark1**.

  All file names that are not fully qualified are relative to the initial working directory **/home/loadl/spice**. LoadLeveler will send the job steps 0 and 1 of this job to a machine for that has a real memory of 64 MB or more for execution. Job step 2 most likely will be sent to a machine that has more that 128 MB of real memory and has the ESSL library installed since these preferences have been stated using the LoadLeveler **preferences** keyword. LoadLeveler will send job step 3 to the machine `ll5.pok.ibm.com` for execution because of the explicit requirement for this machine in the **requirements** statement.

```
#!/bin/ksh
# @ job_name = spice_test
# @ account_no = 99999
# @ class = small
# @ arguments = spice3f5_input_$(stepid) spice3f5_output_$(stepid)
# @ input  = spice2g6_input_$(stepid)
# @ output = $(job_name)_output_$(stepid)
# @ error  = $(job_name)_error_$(stepid)
# @ initialdir = /home/loadl/spice
# @ requirements = ((Arch == "R6000") && (OpSys == "AIX43") && (Memory > 64))
# @ queue
# @ queue
# @ preferences  = ((Memory > 128)  && (Feature == "ESSL"))
# @ queue
# @ class = large
# @ arguments = spice3f5_input_benchmark1 spice3f5_output_benchmark1
# @ requirements = (Machine == "ll5.pok.ibm.com")
# @ input  = spice2g6_input_benchmark1
# @ output = $(job_name)_output_benchmark1
# @ error  = $(job_name)_error_benchmark1
# @ queue
OS_NAME=  `uname`

case $OS_NAME in
   AIX)
      echo "Running $OS_NAME version of spice3f5" > $2
      AIX_bin/spice3f5 < $1  >> $2  2>&1
      echo "Running $OS_NAME version of spice2g6"
      AIX_bin/spice2g6
      ;;
   *)
      echo "spice3f5 for $OS_NAME is not available" > $2
      echo "spice2g6 for $OS_NAME is not available"
      ;;
esac
```

Figure 16. Job Command File Used as the Executable

Job Command File Keywords

This section provides an alphabetical list of the keywords you can use in a LoadLeveler script. It also provides examples of statements that use these keywords. For most keywords, if you specify the keyword in a job step of a multi-step job, its value is inherited by all proceeding job steps. Exceptions to this are noted in the keyword description.

account_no

Supports centralized accounting. Allows you to specify an account number to associate with a job. This account number is stored with job resource information in local and global history files. It may also be validated before LoadLeveler allows a job to be submitted. For more information, see “Chapter 7. Gathering Job Accounting Data” on page 153.

The syntax is:

account_no = *string*

where *string* is a text string that can consist of a combination of numbers and letters. For example, if the job accounting group charges for job time based upon the department to which you belong, your account number would be similar to:

```
account_no = dept34ca
```

arguments

Specifies the list of arguments to pass to your program when your job runs.

The syntax is:

arguments = *arg1 arg2 ...*

For example, if your job requires the numbers 5, 8, 9 as input, your arguments keyword would be similar to:

```
arguments = 5 8 9
```

blocking

Blocking specifies that tasks be assigned to machines in multiples of a certain integer. Unlimited blocking specifies that tasks be assigned to the each machine until it runs out of initiators, at which time tasks will be assigned to the machine which is next in the order of priority. If the total number of tasks are not evenly divisible by the blocking factor, the remainder of tasks are allocated to a single node.

The syntax is:

blocking = *integer*|**unlimited**

Where:

integer
: specifies the blocking factor to be used. The blocking factor must be a positive integer. With a blocking factor of 4, LoadLeveler will allocate 4 tasks at a time to each machine with at least 4 initiators available. This keyword must be specified with the total_tasks keyword. For example:

  ```
  blocking = 4
  total_tasks = 17
  ```

  LoadLeveler will allocate tasks to machines in an order based on the values of their MACHPRIO expressions (beginning with the highest MACHPRIO value). In cases where total_tasks is not a multiple of the blocking factor, LoadLeveler assigns the remaining number of tasks as soon as possible (even if that means assigning the remainder to a machine at the same time as it assigns another block).

unlimited
: Specifies that LoadLeveler allocate as many tasks as possible to each machine, until all of the tasks have been allocated. LoadLeveler will prioritize machines based on the number of initiators each machine currently has available. Unlimited blocking is the only means of allocating tasks to nodes that does not prioritize machines primarily by MACHPRIO expression.

checkpoint

Specifies whether you want to checkpoint your program.

The syntax is:

checkpoint = user_initiated | system_initiated | no

Specify **user_initiated** if you want to determine when the checkpoint is taken. User initiated checkpointing is available to both serial jobs and parallel POE jobs. (Checkpointing is not supported for parallel PVM jobs.) Serial jobs must use the LoadLeveler **ckpt** API call to request user initiated checkpointing. See “Serial Checkpointing API” on page 253 for more information. POE jobs must use the Parallel Environment (PE) parallel checkpointing API to enable user initiated checkpointing. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1* for more information.

Specify **system_initiated** if you want LoadLeveler to automatically checkpoint your program at preset intervals. System initiated checkpointing is available only to serial jobs. To cause both user initiated and system initiated checkpoints to occur, specify **system_initiated** and have your program use the appropriate **ckpt** API call.

Specify **no** if you do not want your program to be checkpointed. This is the default.

To restart a program for which a checkpoint file exists, you must set the CHKPT_STATE environment variable to **restart**. For more information on environment variables associated with checkpointing, see “Set the Appropriate Environment Variables” on page 118. For information on setting environment variables for a job, see “environment” on page 41. Note that it is not necessary to set the **restart** job command language keyword for a checkpointing job. For more information, see “restart” on page 52.

To use checkpointing, your program must be linked with the appropriate LoadLeveler libraries. See “Ensure all User’s Jobs are Linked to Checkpointing Libraries” on page 120 for more information. For more detailed information on checkpointing, see “Step 14: Enable Checkpointing” on page 117.

class

Specifies the name of a job class defined locally in your cluster. If not specified, the default job class, **No_Class**, is assigned. You can use the **llclass** command to find out information on job classes.

The syntax is:

class = *name*

For example, if you are allowed to submit jobs belonging to a class called “largejobs”, your class keyword would look like the following:

```
class = largejobs
```

comment

Specifies text describing characteristics or distinguishing features of the job.

core_limit

Specifies the hard limit and/or soft limit for the size of a core file. This is a per process limit.

The syntax is:

core_limit = *hardlimit*,*softlimit*

Some examples are:

```
core_limit = 125621,10kb
core_limit = 5621kb,5000k
core_limit = 2mb,1.5mb
core_limit = 2.5mw
core_limit = unlimited
core_limit = rlim_infinity
core_limit = copy
```

See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

cpu_limit

Specifies the hard limit and/or soft limit for the amount of CPU time that a submitted job step can use. This is a per process limit.

The syntax is:

cpu_limit = *hardlimit,softlimit*

For example:

```
cpu_limit = 12:56:21,12:50:00
cpu_limit = 56:21.5
cpu_limit = 1:03,21
cpu_limit = unlimited
cpu_limit = rlim_infinity
cpu_limit = copy
```

See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

data_limit

Specifies the hard limit and/or soft limit for the size of the data segment to be used by the job step. This is a per process limit.

The syntax is:

data_limit = *hardlimit,softlimit*

For example:

```
data_limit = ,125621
data_limit = 5621kb
data_limit = 2mb
data_limit = 2.5mw,2mb
```

See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

dependency

Specifies the dependencies between job steps. A job dependency, if used in a given job step, must be explicitly specified for that step.

The syntax is:

dependency = *expression*

where the syntax for the *expression* is:

```
step_name operator
value
```

where *step_name* (as described in “step_name” on page 53) must be a previously defined job step and *operator* can be one of the following:

== equal to
!= not equal to
<= less than or equal to
>= greater than or equal to
< less than
> greater than
&& and
|| or

The *value* is usually a number which specifies the job return code to which the *step_name* is set. It can also be one of the following LoadLeveler defined job step return codes:

CC_NOTRUN
The return code set by LoadLeveler for a job step which is not run because the dependency is not met. The value of CC_NOTRUN is 1002.

CC_REMOVED
The return code set by LoadLeveler for a job step which is removed from the system (because, for example, **llcancel** was issued against the job step). The value of CC_REMOVED is 1001.

Examples: The following are examples of dependency statements:

Example 1: In the following example, the step that contains this dependency statement will run if the return code from step 1 is zero:

```
dependency = (step1 == 0)
```

Example 2: In the following example, step1 will run with the executable called **myprogram1**. Step2 will run only if LoadLeveler removes step1 from the system. If step2 does run, the executable called **myprogram2** gets run.

```
# Beginning of step1
# @ step_name = step1
# @ executable = myprogram1
# @ ...
# @ queue
# Beginning of step2
# @ step_name = step2
# @ dependency = step1 == CC_REMOVED
# @ executable = myprogram2
# @ ...
# @ queue
```

Example 3: In the following example, step1 will run with the executable called **myprogram1**. Step2 will run if the return code of step1 equals zero. If the return code of step1 does not equal zero, step2 does not get executed. If step2 is not run, the dependency statement in step3 gets evaluated and it is determined that step2 did not run. Therefore, **myprogram3** gets executed.

```
# Beginning of step1
# @ step_name = step1
# @ executable = myprogram1
# @ ...
# @ queue
# Beginning of step2
# @ step_name = step2
# @ dependency = step1 == 0
# @ executable = myprogram2
# @ ...
```

```
# @ queue
# Beginning of step3
# @ step_name = step3
# @ dependency = step2 == CC_NOTRUN
# @ executable = myprogram3
# @ ...
# @ queue
```

Example 4: In the following example, the step that contains step2 returns a non-negative value if successful. This step should take into account the fact that LoadLeveler uses a value of 1001 for CC_REMOVED and 1002 for CC_NOTRUN. This is done with the following dependency statement:

```
dependency = (step2 >= 0) && (step2 < CC_REMOVED)
```

environment

Specifies your initial environment variables when your job step starts. Separate environment specifications with semicolons. An environment specification may be one of the following:

COPY_ALL
: Specifies that all the environment variables from your shell be copied.

$var
: Specifies that the environment variable *var* be copied into the environment of your job when LoadLeveler starts it.

!var
: Specifies that the environment variable *var* not be copied into the environment of your job when LoadLeveler starts it. This is most useful in conjunction with COPY_ALL.

var=value
: Specifies that the environment variable *var* be set to the value “value” and copied into the environment of your job when LoadLeveler starts it.

The syntax is:

environment = *env1 ; env2 ; ...*

For example:

```
environment = COPY_ALL; !env2;
```

error

Specifies the name of the file to use as standard error (stderr) when your job step runs. If you do no specify this keyword, the file **/dev/null** is used.

The syntax is:

error = *filename*

For example:

```
error = $(jobid).$(stepid).err
```

executable

For serial jobs, **executable** identifies the name of the program to run. The program can be a shell script or a binary. For parallel jobs, **executable** can be a shell script or the following:

- For Parallel Operating Environment (POE) jobs – specifies the full path name of the POE executable.
- For Parallel Virtual Machine (PVM) jobs – specifies the name of your parallel job.

If you do not include this keyword and the job command file is a shell script, LoadLeveler uses the script file as the executable.

The syntax is:

executable = *name*

Examples:

```
# @ executable = a.out
# @ executable = /usr/bin/poe (for POE jobs)
# @ executable = my_parallel_job (for PVM jobs)
```

Note that the **executable** statement automatically sets the **$(base_executable)** variable, which is the file name of the executable without the directory component. See Figure 15 on page 34 for an example of using the **$(base_executable)** variable.

file_limit

Specifies the hard limit and/or soft limit for the size of a file. This is a per process limit.

The syntax is:

file_limit = *hardlimit*,*softlimit*

For example:

```
file_limit = 120mb,100mb
```

See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

group

Specifies the LoadLeveler group. If not specified, this defaults to the default group, **No_Group**. The syntax is:

group = *group_name*

For example:

```
group = my_group_name
```

hold

Specifies whether you want to place a hold on your job step when you submit it. There are three types of holds:

user Specifies user hold

system
Specifies system hold

usersys
Specifies user and system hold

The syntax is:

hold = user|system|usersys

For example, to put a user hold on a job, the keyword statement would be:

```
hold = user
```

To remove the hold on the job, you can use either the GUI or the **llhold -r** command.

image_size

Maximum virtual image size, in kilobytes, to which your program will grow during execution. LoadLeveler tries to execute your job steps on a machine that has enough resources to support executing and checkpointing your job step. If your job command file has multiple job steps, the job steps will not necessarily run on the same machine, unless you explicitly request that they do.

If you do not specify the image size of your job command file, the image size is that of the executable. If you underestimate the image size of your job step, your job step may crash due to the inability to acquire more address space. If you overestimate the image size, LoadLeveler may have difficulty finding machines that have the required resources.

The syntax is:

image_size = *number*

Where *number* must be a positive integer. For example, to set an image size of 11 KB, the keyword statement would be:

```
image_size = 11
```

initialdir

The path name of the directory to use as the initial working directory during execution of the job step. If none is specified, the initial directory is the current working directory at the time you submitted the job. File names mentioned in the command file which do not begin with a **/** are relative to the initial directory. The initial directory must exist on the submitting machine as well as on the machine where the job runs.

The syntax is:

initialdir = *pathname*

For example:

```
initialdir  = /var/home/mike/ll_work
```

input

Specifies the name of the file to use as standard input (stdin) when your job step runs. If not specified, the file **/dev/null** is used.

The syntax is:

input = *filename*

For example:

```
input = input.$(process)
```

job_cpu_limit

Specifies the hard limit and/or soft limit for the CPU time to be used by all processes of a job step. For example, if a job step forks to produce multiple processes, the sum total of CPU consumed by all of the processes is added and controlled by this limit.

The syntax is:

job_cpu_limit = *hardlimit,softlimit*

For example:

```
job_cpu_limit = 12:56,12:50
```

See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

job_name

Specifies the name of the job. This keyword must be specified in the first job step. If it is specified in other job steps in the job command file, it is ignored. You can name the job using any combination of letters and/or numbers.

The syntax is:

job_name = *job_name*

For example:

```
job_name = my_first_job
```

The job_name only appears in the long reports of the **llq**, **llstatus**, and **llsummary** commands, and in mail related to the job.

job_type

Specifies the type of job step to process. Valid entries are:

pvm3 For PVM jobs with a non-SP architecture.

parallel
For other parallel jobs, including PVM 3.3.11+ (SP architecture).

serial For serial jobs. This is the default.

Note that when you specify **job_type=pvm3** or **job_type=serial**, you cannot specify the following keywords: **node**, **tasks_per_node**, **total_tasks**, **network.LAPI**, and **network.MPI**.

The syntax is:

job_type = *string*

For example:

```
job_type = pvm3
```

max_processors

Specifies the maximum number of nodes requested for a parallel job, regardless of the number of processors contained in the node.

This keyword is equivalent to the maximum value you specify on the new **node** keyword. In any new job command files you create for non-PVM jobs, you should use the **node** keyword to request nodes/processors. The **max_processors** keyword should be used by existing job command files and new PVM job command files. Note that if you specify in a job command file both the **max_processors** keyword and the **node** keyword, the job is not submitted.

The syntax is:

max_processors = *number*

For example:

```
max_processors = 6
```

min_processors

Specifies the minimum number of nodes requested for a parallel job, regardless of the number of processors contained in the node.

This keyword is equivalent to the minimum value you specify on the new **node** keyword. In any new job command files you create for non-PVM jobs, you should use the **node** keyword to request nodes/processors. The **min_processors** keyword should be used by existing job command files and new PVM job command files. Note that if you specify in a job command file both the **min_processors** keyword and the **node** keyword, the job is not submitted.

The syntax is:

min_processors = *number*

For example:

```
min_processors = 4
```

network

Specifies communication protocols, adapters, and their characteristics. You need to specify this keyword when you want a task of a parallel job step to request a specific adapter that is defined in the LoadLeveler administration file. You do not need to specify this keyword when you want a task to access a shared, default adapter via TCP/IP. (A default adapter is an adapter whose name matches a machine stanza name.)

Note that you cannot specify both the **network** statement and the **Adapter** requirement (or the **Adapter** preference) in a job command file. Also, the value of the **network** keyword applies only to the job step in which you specify the keyword. (That is, this keyword in not inherited by other job steps.)

The syntax is:

network.*protocol* = *network_type* [**,** [*usage*] [**,** *mode* [**,** *comm_level*]]]

Where:

protocol
: Specifies the communication protocol(s) that are used with an adapter, and can be the following:

  MPI Specifies the Message Passing Interface. You can specify in a job step both **network.MPI** and **network.LAPI**.

  LAPI Specifies the Low-level Application Programming Interface. You can specify in a job step both **network.MPI** and **network.LAPI**.

  PVM Specifies a Parallel Virtual Machine job. When you specify in a job step **network.PVM**, you cannot specify any other network statements in that job step. Also, the adapter *mode* must be **IP**.

network_type
: Specifies either an adapter name or a network type.This field is required. The possible values for adapter name are the names associated with the

interface cards installed on a node (for example, en0, tk1, and css0). The possible values for network type are installation-defined; the LoadLeveler administrator must specify them in the adapter stanza of the LoadLeveler administration file using the **network_type** keyword. For example, an installation can define a network type of "switch" to identify css0 adapters. When a switch adapter exists on a node, the network_type can be specified as csss, which indicates that the fastest switch communication path should be used. For more information, see "Step 5: Specify Adapter Stanzas" on page 95.

usage Specifies whether the adapter can be shared with tasks of other job steps. Possible values are **shared**, which is the default, or **not_shared**.

mode Specifies the communication subsystem mode used by the communication protocol that you specify, and can be either **IP** (Internet Protocol), which is the default, or **US** (User Space). Note that each instance of the US mode requested by a task running on the SP switch requires an adapter window. For example, if a task requests both the MPI and LAPI protocols such that both protocol instances require US mode, two adapter windows will be used. For more information on adapter windows, see *Parallel System Support Programs for AIX Administration Guide*.

comm_level
The **comm_level** keyword should be used to suggest the amount of inter-task communication that users *expect* to occur in their parallel jobs. This suggestion is used to allocate adapter device resources. For more information on device resources, consult the PSSP Admin Guide. Specifying a level that is higher than what the job actually needs will not speed up communication, but may make it harder to schedule a job (because it requires more resources). The **comm_level** keyword can only be specified with **US** mode. The three communication levels are:

LOW Implies that minimal inter-task communication will occur.

AVERAGE
This is the default value. Unless you know the specific communication characteristics of your job, the best way to determine the comm_level is through trial-and-error.

HIGH Implies that a great deal of inter-task communication will occur.

Example 1: To use the MPI protocol with an SP switch adapter in User Space mode without sharing the adapter, enter the following:

```
network.MPI = css0,not_shared,US,HIGH
```

Example 2: To use the MPI protocol with a shared SP switch adapter in IP mode, enter the following:

```
network.MPI = css0,,IP
```

Because a shared adapter is the default, you do not need to specify **shared**.

Example 3: A communication level can only be specified if User Space mode is also specified:

```
network.MPI = css0,,US,AVERAGE
```

Note that LoadLeveler can ensure that an adapter is dedicated (not shared) if you request the adapter in US mode, since any user who requests a user space adapter must do so using the **network** statement. However, if you request a

dedicated adapter in IP mode, the adapter will only be dedicated if all other LoadLeveler users who request this adapter do so using the **network** statement.

node

Specifies the minimum and maximum number of nodes requested by a job step. You must specify at least one of these values. The value of the **node** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.)

The syntax is:

```
node = [min][,max]
```

Where:

min Specifies the minimum number of nodes requested by the job step. The default is 1.

max Specifies the maximum number of nodes requested by the job step. The default is the *min* value of this keyword. The maximum number of nodes a job step can request is limited by the **max_node** keyword in the administration file (provided this keyword is specified). That is, the maximum must be less than or equal to any **max_node** value specified in a user, group, or class stanza.

For example, to specify a range of six to twelve nodes, enter the following:

```
node = 6,12
```

To specify a maximum of seventeen nodes, enter the following:

```
node =  ,17
```

When you use the **node** keyword together with the **total_tasks** keyword, the *min* and *max* values you specify on the **node** keyword must be equal, or you must specify only one value. For example:

```
node = 6
total_tasks = 12
```

For information on specifying the number of tasks you want to run on a node, see “Task Assignment Considerations” on page 60,“tasks_per_node” on page 54, and “total_tasks” on page 55.

node_usage

Specifies whether this job step shares nodes with other job steps.

The syntax is:

```
node_usage = shared | not_shared
```

Where:

shared
Specifies that nodes can be shared with other tasks of other job steps. This is the default.

not_shared
Specifies that nodes are not shared: no other job steps are scheduled on this node.

notification

Specifies when the user specified in the **notify_user** keyword is sent mail. The syntax is:

```
notification = always|error|start|never|complete
```

Where:

always
Notify the user when the job begins, ends, or if it incurs error conditions.

error Notify the user only if the job fails.

start Notify the user only when the job begins.

never Never notify the user.

complete
Notify the user only when the job ends. This is the default.

For example, if you want to be notified with mail only when your job step completes, your notification keyword would be:

```
notification = complete
```

When a LoadLeveler job ends, you may receive UNIX mail notification indicating the job exit status. For example, you could get the following mail message:

```
Your LoadLeveler job
myjob1
exited with status 4.
```

The return code 4 is from the user's job. LoadLeveler retrieves the return code and returns it in the mail message, but it is not a LoadLeveler return code.

notify_user

Specifies the user to whom mail is sent based on the **notification** keyword. The default is the submitting user and the submitting machine.

The syntax is:

notify_user = *userID*

For example, if you are the job step owner but you want a co-worker whose name and user ID is **bob**, to receive mail regarding the job step, your notify keyword would be:

```
notify_user = bob
```

output

Specifies the name of the file to use as standard output (stdout) when your job step runs. If not specified, the file **/dev/null** is used.

The syntax is:

output = *filename*

For example:

```
output = out.$(jobid)
```

parallel_path

Specifies the path that should be used when starting a PVM 3.3 slave process. This is used for PVM 3.3 only and is translated into the **ep** keyword as defined in the PVM 3.3 **hosts** file.

For example:

```
parallel_path = /home/userid/cmds/pvm3/$PVM_ARCH:$PVM_ROOT/lib/$PVM_ARCH
```

The **parallel_path** statement above has two components, separated by a colon. The first component points to the location of the user's programs. The second component points to the location of the **pvmgs** routine – required if the job uses PVM 3.3 group support – assuming PVM 3.3 is installed "normally". Note that your installation must install PVM 3.3 to include group support in order for you to use group support within LoadLeveler. $PVM_ARCH will be replaced by the architecture of the machine, as defined by PVM 3.3. This will specify the path to be searched for executables when the user's job issues a **pvm_spawn()** command.

$PVM_ARCH, and $PVM_ROOT are PVM environment variables. For more information, see the appropriate PVM 3.3 documentation.

preferences

Specifies the characteristics that you prefer be available on the machine that executes the job steps. LoadLeveler attempts to run the job steps on machines that meet your preferences. If such a machine is not available, LoadLeveler will then assign machines which meet only your requirements.

The values you can specify in a **preferences** statement are the same values you can specify in a **requirements** statement, with the exception of the **Adapter** requirement. See "requirements" for more information.

The syntax is:

preferences = *Boolean_expression*

Some examples are::

```
preferences = (Memory <=16) && (Arch == "R6000")
```

```
preferences = Memory >= 64
```

queue

Places one copy of the job step in the queue. This statement is required. The **queue** statement essentially marks the end of the job step. Note that you can specify statements between **queue** statements.

The syntax is:

queue

requirements

Specifies the requirements which a machine in the LoadLeveler cluster must meet to execute any job steps. You can specify multiple requirements on a single requirements statement.

The syntax is:

requirements = *Boolean_expression*

When strings are used as part of a Boolean expression that must be enclosed in double quotes. Sample requirement statements are included following the descriptions of the supported requirements.

The requirements supported are:

Adapter
Specifies the pre-defined type of network you want to use to run a parallel job step. In any new job command files you create, you should use the **network** keyword to request adapters and types of networks. The **Adapter** requirement is provided for compatibility with Version 1.3 job command files. Note that you cannot specify both the **Adapter** requirement and the **network** statement in a job command file.

The pre-defined network types are:

hps_ip
Refers to an SP switch in IP mode.

hps_us
Refers to an SP switch in user space mode. If the switch in user mode is requested by the job, no other jobs using the switch in user mode will be allowed on nodes running that job.

ethernet
Refers to Ethernet.

fddi Refers to Fiber Distributed Data Interface (FDDI).

tokenring
Refers to Token Ring.

fcs Refers to Fiber Channel Standards.

Note that LoadLeveler converts the above network types to the **network** statement. For more information, see “Migrating Your Existing Adapter Requirements Statements” on page xix.

Arch
Specifies the machine architecture on which you want your job step to run. It describes the particular kind of UNIX platform for which your executable has been compiled. The default is the architecture of the submitting machine.

Disk
Specifies the amount of disk space in kilobytes you believe is required in the LoadLeveler **execute** directory to run the job step.

Feature
Specifies the name of a feature defined on a machine where you want your job step to run. Be sure to specify a feature in the same way in which the feature is specified in the machine stanza of the administration file. To find out what features are available, use the **llstatus** command.

LL_Version
Specifies the LoadLeveler version, in dotted decimal format, on which you want your job step to run. For example, LoadLeveler Version 2 Release 1 (with no modification levels) is written as 2.1.0.0.

Machine
Specifies the name(s) of machines on which you want the job step to run. Be sure to specify a machine in the same way in which it is specified in the machine configuration file.

Memory
Specifies the amount of physical memory required in megabytes in the machine where you want your job step to run.

OpSys
Specifies the operating system on the machine where you want your job step to run. It describes the particular kind of UNIX platform for which your executable has been compiled. The default is the operating system of the submitting machine. The executable must be compiled on a machine that matches these requirements.

Pool
Specifies the number of a pool where you want your job step to run.

Example 1: To specify a memory requirement and a machine architecture requirement, enter:

```
requirements = (Memory >=16) && (Arch == "R6000")
```

Example 2: To specify that your job requires multiple machines for a parallel job, enter:

```
requirements = (Machine == { "ll6" "ll5" "ll0" })
```

Example 3: You can set a machine equal to a job step name. This means that you want the job step to run on the same machine on which the previous job step ran. For example:

requirements = (Machine == machine.*step_name***)**

where *step_name* is a step name previously defined in the job command file. The use of **Machine == machine.***step_name* is limited to serial jobs.

For example:

```
# @ step_name    = step1
# @ executable   = c1
# @ output       = $(executable).$(jobid).$(step_name).out
# @ queue
# @ step_name    = step2
# @ dependency   = (step1 == 0)
# @ requirements = (Machine == machine.step1)
# @ executable   = c2
# @ output       = $(executable).$(jobid).$(step_name).out
# @ queue
```

Example 4: To specify a requirement for an SP switch adapter in IP mode, enter:

```
requirements = (Adapter == "hps_ip")
```

Example 5: To specify a requirement for a specific pool number, enter:

```
requirements = (Pool == 7)
```

Example 6: To specify a requirement that the job runs on LoadLeveler Version 2 Release 1 or any follow-on release, enter:

```
requirements = (LL_Version >= "2.1")
```

Note that the statement **requirements = (LL_Version == "2.1")** matches only the value 2.1.0.0.

resources

Specifies quantities of the consumable resources ″consumed″ by each task of a job step. The resources may be machine resources or floating resources. The syntax is:

resources=*name(count) name(count) ... name(count)*

where *name(count)* is an administrator-defined name and count, or could also be **ConsumableCpus***(count)*, **ConsumableMemory***(count units)*, or **ConsumableVirtualMemory***(count units)*. **ConsumableMemory** and **ConsumableVirtualMemory** are the only two consumable resources that can be specified with both a count and units. The count for each specified resource must be an integer greater than or equal to zero, with three exceptions: **ConsumableCpus**, and **ConsumableMemory** must be specified with a value which is greater than zero, and **ConsumableVirtualMemory** must be specified with a value greater than 0, and greater than or equal to the **image_size**. If the count is not valid, then LoadLeveler will issue an error message, and will not submit the job. The allowable units are those normally used with LoadLeveler data limits:

```
b bytes
w words
kb kilobytes (2** 10 bytes)
kw kilowords (2** 10 words)
mb megabytes (2** 20 bytes)
mw megawords (2**20 words)
gb gigabytes (2** 30 bytes)
gw gigawords (2** 30 words)
```

ConsumableMemory and **ConsumableVirtualMemory** values are stored in mb (megabytes) and rounded up. Therefore, the smallest amount of **ConsumableMemory** or **ConsumableVirtualMemory** which you can request is one megabyte. If no units are specified, then megabytes are assumed. However, **image_size** units are in kilobytes. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not affect the scheduling of the job. If the **resources** keyword is not specified in the job step, then the **default_resources** (if any) defined in the administration file for the class will be used for each task of the job step.

restart

Specifies whether LoadLeveler considers a job “restartable.” The syntax is:

restart = yes|no

If **restart=yes**, which is the default, and the job is vacated from its executing machine before completing, the central manager requeues the job. It can start running again when a machine on which it can run becomes available. If **restart=no**, a vacated job is cancelled rather than requeued.

Note that this keyword is different from the **restart** state associated with checkpointing jobs. This state tells LoadLeveler to restart a job from an existing checkpoint file. (Checkpoint jobs are always considered “restartable.”) For more information, see “Set the Appropriate Environment Variables” on page 118.

rss_limit

Specifies the hard limit and/or soft limit for the resident set size.

The syntax is:

rss_limit = *hardlimit,softlimit*

For example:

```
rss_limit=12mb,10mb
```

The above example specifies the limits in megabytes, but If no units are specified, then bytes are assumed. See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

shell

Specifies the name of the shell to use for the job step. If not specified, the shell used in the owner’s password file entry is used. If none is specified, the /bin/sh is used.

The syntax is:

shell = *name*

For example, if you wanted to use the Korn shell, the shell keyword would be:

```
shell = /bin/ksh
```

stack_limit

Specifies the hard limit and/or soft limit for the size of the stack that is created.

The syntax is:

stack_limit = *hardlimit,softlimit*

For example:

```
stack_limit = 120000,100000
```

Because no units have been specified in the above example, LoadLeveler assumes that the figure represents a number of bytes. See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

startdate

Specifies when you want to run the job step. If not specified, the current date and time are used.

The syntax is:

startdate = *date time*

date is expressed as *MM/DD/YYYY*, and *time* is expressed as *HH:mm(:ss)*.

For example, if you want the job to run on August 28th, 2000 at 1:30 PM, issue:

```
startdate = 08/28/2000 13:30
```

If you specify a start date that is in the future, your job is kept in the Deferred state until that start date.

step_name

Specifies the name of the job step. You can name the job step using any combination of letters, numbers, underscores (_) and periods (.). You cannot, however, name it T or F, or use a number in the first position of the step name. The step name you use must be unique and can be used only once. If you don’t specify a step name, by default the first job step is named the character string ″0″, the second is named the character string ″1″, and so on.

The syntax is:

step_name = *step_name*

For example:

```
step_name = step_3
```

task_geometry

The **task_geometry** keyword allows you to group tasks of a parallel job step to run together on the same node. Although task_geometry allows for a great deal of flexibility in how tasks are grouped, you cannot specify the particular nodes that these groups run on; the scheduler will decide which nodes will run the specified groupings. The syntax is:

task_geometry={(*task id***,***task id***,...)(***task id***,***task id***, ...) ... }**

In this example, a job with 6 tasks will run on 4 different nodes:

```
task_geometry={(0,1) (3) (5,4) (2)}
```

Each number in the example above represents a task id in a job, each set of parenthesis contains the task ids assigned to one node. The entire range of tasks specified must begin with 0, and must be complete; no number can be skipped (the largest task id number should end up being the value that is one less than the total number of tasks). The entire statement following the keyword must be enclosed in braces, and each grouping of nodes must be enclosed in parentheses. Commas can only appear between task ids, and spaces can only appear between nodes and task ids.

The **task_geometry** keyword cannot be specified under any of the following conditions: (a) the step is serial, (b) **job_type** is anything other than ″parallel″, or (c) any of the following keywords are specified: **tasks_per_node**, **total_tasks**, **node**, **min_processors**, **max_processors**, **blocking**. For more information, see “Task Assignment Considerations” on page 60.

tasks_per_node

Specifies the number of tasks of a parallel job you want to run per node. Use this keyword in conjunction with the **node** keyword. The value you specify on the **node** keyword can be a range or a single value. If the node keyword is not specified, then the default value is one node.

The maximum number of tasks a job step can request is limited by the **total_tasks** keyword in the administration file (provided this keyword is specified). That is, the maximum must be less than any **total_tasks** value specified in a user, group, or class stanza.

The value of the **tasks_per_node** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.)

Also, you cannot specify both the **tasks_per_node** keyword and the **total_tasks** keyword within a job step.

The syntax is:

tasks_per_node = *number*

Where *number* is the number of tasks you want to run per node. The default is one task per node.

For example, to specify a range of seven to 14 nodes, with four tasks running on each node, enter the following:

```
node = 7,14
tasks_per_node = 4
```

The above job step runs 28 to 56 tasks, depending on the number of nodes allocated to the job step.

total_tasks

Specifies the total number of tasks of a parallel job you want to run on all available nodes. Use this keyword in conjunction with the **node** keyword. The value you specify on the **node** keyword must be a single value rather than a range of values. If the node keyword is not specified, then the default value is one node.

The maximum number of tasks a job step can request is limited by the **total_tasks** keyword in the administration file (provided this keyword is specified). That is, the maximum must be less than any **total_tasks** value specified in a user, group, or class stanza.

The value of the **total_tasks** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.)

Also, you cannot specify both the **total_tasks** keyword and the **tasks_per_node** keyword within a job step.

The syntax is:

total_tasks = *number*

Where *number* is the total number of tasks you want to run.

For example, to run two tasks on each of 12 available nodes for a total of 24 tasks, enter the following:

```
node = 12
total_tasks = 24
```

If you specify an unequal distribution of tasks per node, LoadLeveler allocates the tasks on the nodes in a round-robin fashion. For example, if you have three nodes and five tasks, two tasks run on the first two nodes and one task runs on the third node.

user_priority

Sets the initial priority of your job step. Priority only affects your job steps. It orders job steps you submitted with respect to other job steps submitted by you, not with respect to job steps submitted by other users.

The syntax is:

user_priority = *number*

where *number* is a number between 0 and 100, inclusive. A higher number indicates the job step will be selected before a job step with a lower number. The default priority is 50. Note that this is not the UNIX *nice* priority.

This priority guarantees the order the jobs are considered for dispatch. It does not guarantee the order in which they will run.

wall_clock_limit

Sets the hard limit and/or soft limit for the elapsed time for which a job can run. In computing the elapsed time for a job, LoadLeveler considers the start time to be the time the job is dispatched.

If you are running the LoadLeveler Backfill scheduler, either users must set a wall clock limit in their job command file or the administrator must define a wall clock limit value for the class to which a job is assigned. In most cases, this wall clock limit value should not be **unlimited**. For more information, see “Choosing a Scheduler” on page 100.

The syntax is:

wall_clock_limit = *hardlimit*,*softlimit*

An example is:

```
wall_clock_limit = 5:00,4:30
```

See “Limit Keywords” on page 88 for more information on the values and units you can use with this keyword.

Job Command File Variables

LoadLeveler has several variables you can use in a job command file. These variables are useful for distinguishing between output and error files.

You can refer to variables in mixed case, but you must specify them using the following syntax:

$(*variable_name***)**

The following variables are available to you:

$(host)
The hostname of the machine from which the job was submitted. In a job command file, the **$(host)** variable and the **$(hostname)** variable are equivalent.

$(domain)
The domain of the host from which the job was submitted.

$(jobid)
The sequential number assigned to this job by the submitting machine. The **$(jobid)** variable and the **$(cluster)** variable are equivalent.

$(stepid)
The sequential number assigned to this job step when multiple queue statements are used with the job command file. The **$(stepid)** variable and the **$(process)** variable are equivalent.

In addition, the following keywords are also available as variables. However, you must define them in the job command file. These keywords are described in detail in “Job Command File Keywords” on page 36.

$(executable)
$(class)
$(comment)

$(job_name)
$(step_name)

Note that for the **$(comment)** variable, the keyword definition must be a single string with no blanks. Also, the **executable** statement automatically sets the **$(base_executable)** variable, which is the file name of the executable without the directory component. See Figure 15 on page 34 for an example of using the **$(base_executable)** variable.

Example 1

The following job command file creates an output file called **stance.78.out**, where `stance` is the host and `78` is the jobid.

```
# @ executable = my_job
# @ arguments  = 5
# @ output     = $(host).$(jobid).out
# @ queue
```

Example 2

The following job command file creates an **output** file called **computel.step1.March05**.

```
# @ comment    = March05
# @ job_name   = computel
# @ step_name  = step1
# @ executable = my_job
# @ output     = $(job_name).$(step_name).$(comment)
# @ queue
```

Run-time Environment Variables

The following environment variables are set by LoadLeveler for all jobs. These environment variables are also set before running prolog and epilog programs. For more information on prolog and epilog programs, see “Writing Prolog and Epilog Programs” on page 297.

LOADLBATCH
: Set to **yes** to indicate the job is running under LoadLeveler.

LOADL_ACTIVE
: The LoadLeveler version.

LOADL_JOB_NAME
: The three part job identifier.

LOADL_PID
: The process ID of the starter process.

LOADL_PROCESSOR_LIST
: A Blank-delimited list of hostnames allocated for the step. This environment variable is limited to 128 hostnames. If the value is greater than the 128 limit, the environment variable is not set.

LOADL_STARTD_PORT
: The port number where the startd daemon runs.

LOADL_STEP_ACCT
: The account number of the job step owner.

LOADL_STEP_ARGS
: Any arguments passed by the job step.

LOADL_STEP_CLASS
: The job class for serial jobs.

LOADL_STEP_COMMAND
: The name of the executable (or the name of the job command file if the job command file is the executable).

LOADL_STEP_ERR
The file used for standard error messages (stderr).
LOADL_STEP_GROUP
The UNIX group name of the job step owner.
LOADL_STEP_ID
The job step ID.
LOADL_STEP_IN
The file used for standard input (stdin).
LOADL_STEP_INITDIR
The initial working directory.
LOADL_STEP_NAME
The name of the job step.
LOADL_STEP_NICE
The UNIX *nice* value of the job step. This value is determined by the **nice** keyword in the class stanza. For more information, see “Step 3: Specify Class Stanzas” on page 84.
LOADL_STEP_OUT
The file used for standard output (stdout).
LOADL_STEP_OWNER
The job step owner.
LOADL_STEP_TYPE
The job type (SERIAL, PARALLEL, PVM3, or NQS)

Submitting and Managing Jobs that Consume Resources

Specifying the Consumption of Resources by a Job Step

The LoadLeveler user may use the **resources** keyword in the job command file to specify the resources to be consumed by each task of a job step. If the **resources** keyword is specified in the job command file, it overrides any **default_resources** specified by the administrator for the job step’s class.

For example, the following job requests one CPU and one FRM license for each of its tasks:

```
resources = ConsumableCpus(1) FRMlicense(1)
```

If this were specified in a serial job step, one CPU and one FRM license would be consumed while the job step runs. If this were a parallel job step, then the number of CPUs and FRM licenses consumed while the job step runs would depend upon how many tasks were running on each machine. For more information on assigning tasks to nodes, see “Task Assignment Considerations” on page 60.

Displaying Currently Available Resources

The LoadLeveler user can get information about currently available resources by using the **llstatus** command with either the **-F**, or **-R** options. The **-F** option displays a list of all of the floating resources associated with the LoadLeveler cluster. The **-R** option list all of the consumable resources associated with all of the machines in the LoadLeveler cluster. The user can specify a hostlist with the **llstatus** command to display only the consumable resources associated with specific hosts.

Chapter 4. Submitting and Managing Parallel Jobs

This chapter tells you how to submit and manage parallel jobs. For information on setting up and planning for parallel jobs, see “Chapter 6. Administration Tasks for Parallel Jobs” on page 149.

Supported Parallel Environments

LoadLeveler allows you to schedule parallel batch jobs that have been written using the following:

- IBM Parallel Environment Library* (POE/MPI/LAPI) 2.4.0
- Parallel Virtual Machine (PVM) 3.3 (RS6K architecture)
- Parallel Virtual Machine (PVM) 3.3.11+ (SP2MPI architecture)

Note that for parallel batch jobs, LoadLeveler no longer interacts with the PSSP Resource Manager, since all Resource Manager function has been incorporated into LoadLeveler. For more information, see “Resource Manager Functions Now in LoadLeveler” on page xix.

Keyword Considerations for Parallel Jobs

Scheduler Considerations

Several LoadLeveler job command language keywords are associated with parallel jobs. Whether a keyword is appropriate is dependent upon the type of job and the type of LoadLeveler scheduler you are running.

Table 4 shows you the parallel keywords supported by the LoadLeveler Backfill scheduler, based on the type of job you are running.

Table 4. Parallel Keywords Supported by the Backfill Scheduler

job_type=parallel	job_type=pvm3
network node node_usage tasks_per_node total_tasks task_geometry blocking All keywords supported for job_type=pvm3 (supported for compatibility reasons)	Adapter requirement max_processors min_processors network parallel_path

Table 5 shows you the parallel keywords supported by the default LoadLeveler scheduler, based on the type of job you are running.

Table 5. Parallel Keywords Supported by the Default Scheduler

job_type=parallel	job_type=pvm3
max_processors min_processors Adapter requirement	max_processors min_processors parallel_path Adapter requirement

These keywords are used in the examples in this chapter, and are described in more detail in “Job Command File Keywords” on page 36.

If you disable the default LoadLeveler scheduler to run an external scheduler, see “Usage Notes” on page 290 for an explanation of which keywords are supported.

Task Assignment Considerations

You can use the following keywords to specify how LoadLeveler assigns tasks to nodes. With the exception of unlimited blocking, each of these methods prioritizes machines in an order based on their MACHPRIO expressions. Various task assignment keywords can be used in combination, and others are mutually exclusive.

Table 6. Valid Combinations of Task Assignment Keywords

Keyword	Valid Combinations							
total_tasks	X	X						
tasks_per_node			X	X				
node = <min, max>			X					
node = <number>	X			X				
min_processors					X		X	
max_processors						X	X	
task_geometry								X
blocking		X						

The following examples show how each allocation method works. For each example, consider a 3-node SP with machines named ″N1,″ ″N2,″ and ″N3″. The machines’ order of priority, according to the values of their MACHPRIO expressions, is: N1, N2, N3. N1 has 4 initiators available, N2 has 6, and N3 has 8.

node and total_tasks

When you specify the node keyword with the total_tasks keyword, the assignment function will allocate all of the tasks in the job step evenly among however many nodes you have specified. If the number of total_tasks is not evenly divisible by the number of nodes, then the assignment function will assign any larger groups to the first node(s) on the list that can accept them. In this example, 14 tasks must be allocated among 3 nodes:

```
# @ node=3
# @ total_tasks=14
```

Table 7. node and total_tasks

Machine	Available Initiators	Assigned Tasks
N1	4	4
N2	6	5
N3	8	5

The assignment function divides the 14 tasks into groups of 5, 5, and 4, and begins at the top of the list, to assign the first group of 5. The assignment function starts at N1, but because there are only 4 available initiators, cannot assign a block of 5

tasks. Instead, the function moves down the list and assigns the two groups of 5 to N2 and N3, the assignment function then goes back and assigns the group of 4 tasks to N1.

node and tasks_per_node

When you specify the node keyword with the tasks_per_node keyword, the assignment function will assign tasks in groups of the specified value among the specified number of nodes.

```
# @ node = 3
# @ tasks_per_node = 4
```

blocking

When you specify blocking, tasks are allocated to machines in groups (blocks) of the specified number (blocking factor). The assignment function will assign one block at a time to the machine which is next in the order of priority until all of the tasks have been assigned. If the total number of tasks are not evenly divisible by the blocking factor, the remainder of tasks are allocated to a single node. The blocking keyword must be specified with the total_tasks keyword. For example:

```
# @ blocking = 4
# @ total_tasks = 17
```

Where **blocking** specifies that a job's tasks will be assigned in blocks, and **4** designates the size of the blocks. Here is how a blocking factor of 4 would work with 17 tasks:

Table 8. blocking

Machine	Available Initiators	Assigned Tasks
N1	4	4
N2	6	5
N3	8	8

The assignment function first determines that there will be 4 blocks of 4 tasks, with a remainder of one task. Therefore, the function will allocate the remainder with the first block that it can. N1 gets a block of four tasks, N2 gets a block, plus the remainder, then N3 gets a block. The assignment function begins again at the top of the priority list, and N3 is the only node with enough initiators available, so N3 ends up with the last block.

unlimited blocking

When you specify unlimited blocking, the assignment function will allocate as many jobs as possible to each node; the function prioritizes nodes primarily by how many initiators each node has available, and secondarily on their MACHPRIO expressions. This method allows you to allocate tasks among as few nodes as possible. To specify unlimited blocking, specify ″unlimited″ as the value for the blocking keyword. The total_tasks keyword must also be specified with unlimited blocking. For example:

```
# @ blocking = unlimited
# @ total_tasks = 17
```

Table 9. unlimited blocking

Machine	Available Initiators	Assigned Tasks
N3	8	8
N2	6	6
N1	4	3

The assignment function begins with N3 (because N3 has the most initiators available), and assigns 8 tasks, N2 takes six, and N1 takes the remaining 3.

task_geometry

The task_geometry keyword allows you to specify which tasks run together on the same machines, although you cannot specify which machines. In this example, the task_geometry keyword groups 7 tasks to run on 3 nodes:

```
# @ task_geometry = {(5,2)(1,3)(4,6,0)}
```

The entire task_geometry expression must be enclosed within braces. The task IDs for each node must be enclosed within parentheses, and must be separated by commas. The entire range of task IDs that you specify must begin with zero, and must end with the task ID which is one less than the total number of tasks. You can specify the task IDs in any order, but you cannot skip numbers (the range of task IDs must be complete). Commas may only appear between task IDs, and spaces may only appear between nodes and task IDs.

Running Interactive POE Jobs

POE will accept LoadLeveler job command files; however, you can still set the following environment variables to define specific LoadLeveler job attributes before running an interactive POE job:

LOADL_ACCOUNT_NO
The account number associated with the job.

LOADL_INTERACTIVE_CLASS
The class to which the job is assigned.

For information on other POE environment variables, see *IBM Parallel Environment for AIX; Operation and Use, Volume 1*.

Job Command File Examples

This section contains sample job command files for the following parallel environments:

- IBM AIX Parallel Operating Environment (POE) 2.4.0
- Parallel Virtual Machine (PVM) 3.3 (RS6K architecture)
- Parallel Virtual Machine (PVM) 3.3.11+ (SP2MPI architecture)

POE 2.4.0

Figure 17 on page 63 is a sample job command file for POE 2.4.0.

```
#
# @ job_type = parallel
# @ environment = COPY_ALL
# @ output = poe.out
# @ error = poe.error
# @ node = 8,10
# @ tasks_per_node = 2
# @ network.LAPI = switch,shared,US
# @ network.MPI = switch,shared,US
# @ wall_clock_limit = 60
# @ executable = /usr/bin/poe
# @ arguments = /u/richc/My_POE_program -euilib "us"
# @ class = POE
# @ queue
```

Figure 17. POE 2.4.0 Job Command File – Multiple Tasks Per Node

Figure 17 shows the following:

- The total number of nodes requested is a minimum of eight and a maximum of 10 (**node=8,10**). Two tasks run on each node (**tasks_per_node=2**). Thus the total number of tasks can range from 16 to 20.
- Each task of the job can run using the LAPI protocol in US mode with an SP switch adapter (**network.LAPI=switch,shared,US**), and/or using the MPI protocol in US mode with an HPS adapter (**network.MPI=switch,shared,US**). Note that “switch” is an installation-defined network type which is used for css0 adapters in these examples.
- The maximum run time allowed for the job is 60 seconds (**wall_clock_limit=60**).

Figure 18 is a second sample job command file for POE 2.4.0.

```
#
# @ job_type = parallel
# @ input = poe.in.1
# @ output = poe.out.1
# @ error = poe.err
# @ node = 2,8
# @ network.MPI = switch,shared,IP
# @ wall_clock_limit = 60
# @ class = POE
# @ queue
/usr/bin/poe /u/richc/my_POE_setup_program -infolevel 2
/usr/bin/poe /u/richc/my_POE_main_program -infolevel 2
```

Figure 18. POE Sample Job Command File – Invoking POE Twice

Figure 18 shows the following:

- POE is invoked twice, via **my_POE_setup_program** and **my_POE_main_program**.
- The job requests a minimum of two nodes and a maximum of eight nodes (**node=2,8**).
- The job by default runs one task per node.
- The job uses the MPI protocol with an SP switch adapter in IP mode (**network.MPI=switch,shared,IP**).
- The maximum run time allowed for the job is 60 seconds (**wall_clock_limit=60**).

PVM 3.3 (Non-SP)

Figure 19 shows a sample job command file for PVM 3.3 (RS6K architecture). Before using PVM, users should contact their administrator to determine which PVM architecture has been installed.

```
# @ executable    = my_PVM_program
# @ job_type      = pvm3
# @ parallel_path = /home/LL_userid/cmds/pvm3/$PVM_ARCH:$PVM_ROOT/lib/$PVM_ARCH
# @ class         = PVM3
# @ requirements  = (Pool == 4)
# @ output = my_PVM_program.$(cluster).$(process).out
# @ error  = my_PVM_program.$(cluster).$(process).err
# @ min_processors = 8
# @ max_processors = 10
# @ queue
```

Figure 19. Sample PVM 3.3 Job Command File

Note the following requirements for PVM 3.3 (RS6K architecture) jobs:

- The job must have **job_type = pvm3**.
- You must specify the parallel executable as the executable.

PVM 3.3.11+ (SP2MPI architecture)

Figure 20 on page 65 shows a sample job command file for PVM 3.3.11+ (SP2MPI architecture). Before using PVM, users should contact their administrator to determine which PVM architecture has been installed. The SP2MPI architecture version should be used when users require that their jobs run in user space.

```
#!/bin/ksh
# @ job_type      = parallel
# @ class         = PVM3
# @ requirements  = (Adapter == "hps_us")
# @ output = my_PVM_program.$(cluster).$(process).out
# @ error  = my_PVM_program.$(cluster).$(process).err
# @ node = 3,3
# @ queue

# Set PVM daemon and starter path dictated by LoadLeveler administrator
starter_path=/home/userid/loadl/pvm3/bin/SP2MPI
daemon_path=/home/userid/loadl/pvm3/lib/SP2MPI

# Export "MP_EUILIB" before starting PVM3 (default is "ip")
export MP_EUILIB=us
echo MP_EUILIB=$MP_EUILIB

# Clean up old PVM log and daemon files belonging to user
filelog=/tmp/pvml.id | awk -F'=' '{print $2}' | awk -F'(' '{print $1}'
filedaemon=/tmp/pvmd.id | awk -F'=' '{print $2}' | awk -F'(' '{print $1}'
rm -f $filelog > /dev/null
rm -f $filedaemon > /dev/null

# Start PVM daemon in background
$daemon_path/pvmd3 &
echo "pvm background pid=$!"
echo "Sleep 2 seconds"
sleep 2
echo "PVM daemon started"

# Start parallel executable
llnode_cnt='echo "$LOADL_PROCESSOR_LIST" | awk '{print NF}''
actual_cnt=expr "$llnode_cnt" - 1
$starter_path/starter -n $actual_cnt /home/userid/my_PVM_program
echo "Parallel executable starting"

# Check processes running and halt PVM daemon
echo "ps -a" | /home/userid/loadl/pvm3/lib/SP2MPI/pvm
echo "Halt PVM daemon"
echo "halt" | /home/userid/loadl/pvm3/lib/SP2MPI/pvm
wait
echo "PVM daemon completed"
```

Figure 20. Sample PVM 3.3.11+ (SP2MPI Architecture) Job Command File

Note the following requirements for PVM 3.3.11+ (SP2MPI architecture) jobs:

- The job must have **job_type = parallel**.
- You must specify one more processor then you actually need to run the parallel job. PVM spawns an additional task to relay messages to and from the PVM daemon. Parallel tasks cannot communicate with PVM daemon directly. The additional task will be spawned on the last processor in the LOADL_PROCESSOR_LIST. For more information on this environment variable set by LoadLeveler see “Obtaining Allocated Host Names” on page 67.
- You must use the PVM daemon and starter path dictated by the LoadLeveler administrator. The **parallel_path** keyword is ignored.
- You must export MP_EUILIB as **us** when running in user space over the switch. MP_PROCS, MP_RMPOOL and MP_HOSTFILE are ignored when running under LoadLeveler.
- You should clean up any temporary PVM log or daemon files before starting the PVM daemon.

- You must start the PVM daemon in the job script, and you must start it in the background (**$daemon_path/pvmd3 &**).
- You must compile your parallel program following the PVM guidelines for PVM 3.3.11+ (SP2MPI architecture).
- You must start the parallel executable through the PVM starter program. The PVM starter program has no relationship to the LoadLeveler starter daemon.
- You must specify the parallel executable as an argument to the PVM starter program.
- You must specify the actual number of parallel tasks to the PVM starter program. This number must be one less then the number of processors allocated through LoadLeveler.
- You must halt the PVM daemon when the PVM starter program completes.
- You can invoke the PVM starter program only once.

Sequence of Events in a PVM 3.3.11+ Job

This example demonstrates the sequence of events that occur when you submit the sample job command file shown in Figure 20 on page 65.

Figure 21 on page 67 illustrates the following:

- From the job command file, **(1)** the PVM daemon, pvmd3, and **(2)** the PVM starter are started under the LoadLeveler starter. The PVM starter tells the PVM daemon to start two tasks (**my_PVM_program**).
- **(3)** The PVM daemon starts the POE Partition Manager, which in turn **(4)** starts the POE daemons, (represented as pvmd2) on all three nodes.
- **(5)** The POE daemons (pvmd2) start the parallel tasks, **my_PVM_program**, on all nodes under the LoadLeveler starter. The last parallel task, **my_PVM_program** on Node 3, is the additional task which relays messages between the PVM daemon and the parallel tasks.

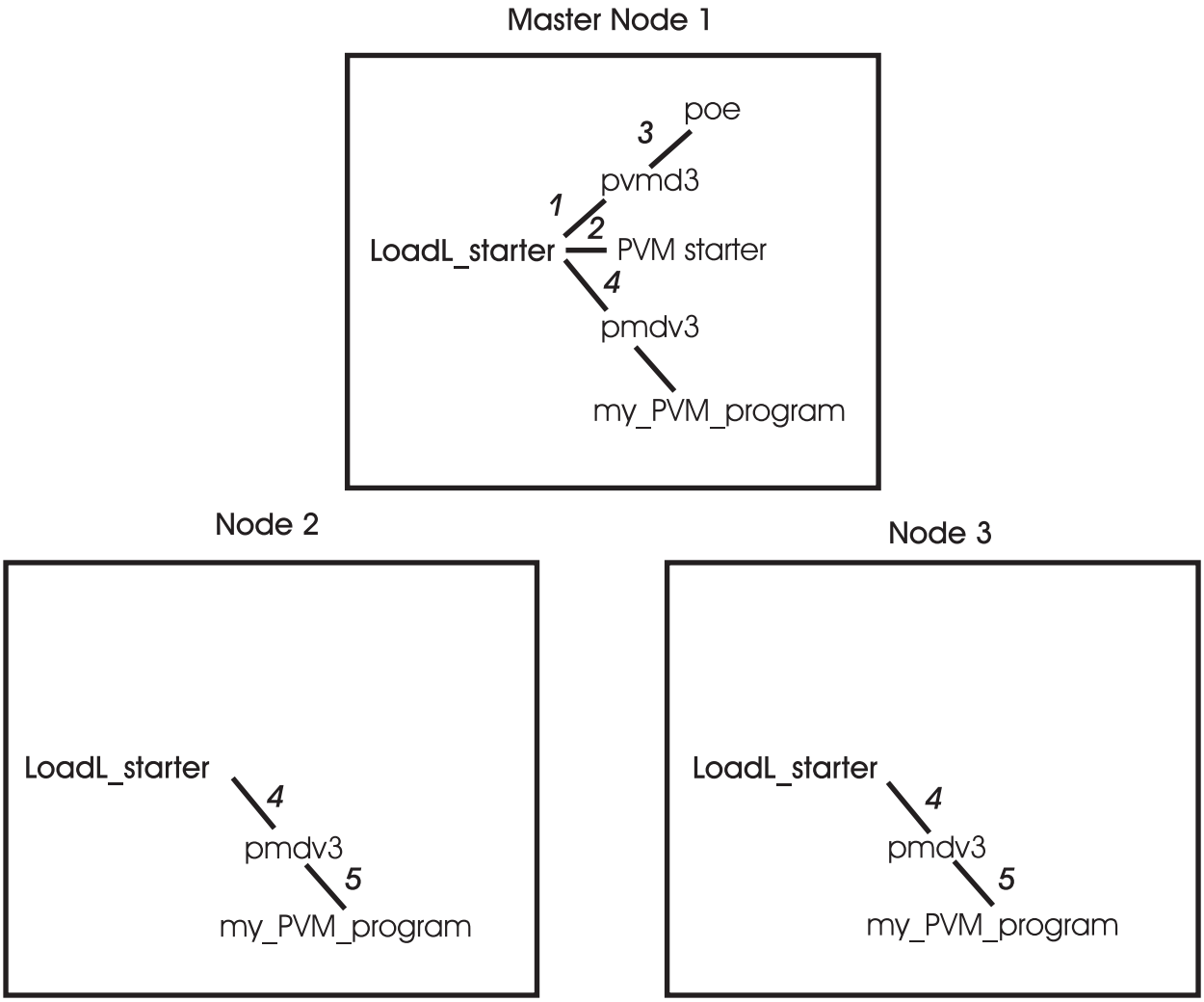

Figure 21. Sequence of Events in a PVM 3.3.11+ Job

Obtaining Status of Parallel Jobs

Both end users and LoadLeveler administrators can obtain status of parallel jobs in the same way as they obtain status of serial jobs – either by using the **llq** command or by viewing the Jobs window on the graphical user interface (GUI). By issuing **llq -l**, or by using the Job Details selection in the GUI, users get a list of machines allocated to the parallel job. See “llq - Query Job Status” on page 193 for sample output from an **llq -l** command issued to query a parallel job.

Also, administrators can create a class for parallel jobs. Users can check the status of their parallel jobs by specifying this class in the Class field on the Jobs window of the GUI.

Obtaining Allocated Host Names

llq -l output includes information on allocated host names. Another way to obtain the allocated host names is with the **LOADL_PROCESSOR_LIST** environment variable, which you can use from a shell script in your job command file as shown in Figure 22 on page 68.

This example uses **LOADL_PROCESSOR_LIST** to perform a remote copy of a local file to all of the nodes, and then invokes POE. Note that the processor list contains an entry for each task running on a node. If two tasks are running on a node, **LOADL_PROCESSOR_LIST** will contain two instances of the host name where the tasks are running. The example in Figure 22 removes any duplicate entries.

Note that **LOADL_PROCESSOR_LIST** is set by LoadLeveler, not by the user. This environment variable is limited to 128 hostnames. If the value is greater than the 128 limit, the environment variable is not set.

```
#!/bin/ksh
# @ output     =  my_POE_program.$(cluster).$(process).out
# @ error      =  my_POE_program.$(cluster).$(process).err
# @ class      =  POE
# @ job_type   =  parallel
# @ node = 8,12
# @ network.MPI = css0,shared,US
# @ queue

tmp_file="/tmp/node_list"
rm -f $tmp_file

# Copy each entry in the list to a new line in a file so
# that duplicate entries can be removed.
for node in $LOADL_PROCESSOR_LIST
        do
                echo $node >> $tmp_file
        done

# Sort the file removing duplicate entries and save list in variable
nodelist= sort -u /tmp/node_list

for node in $nodelist
        do
                rcp localfile $node:/home/userid
        done

rm -f $tmp_file


/usr/bin/poe /home/userid/my_POE_program
```

Figure 22. Using LOADL_PROCESSOR_LIST in a Shell Script

Part 3. Administering LoadLeveler

Chapter 5. Administering and Configuring LoadLeveler

This chapter tells you how to administer and configure LoadLeveler. In general, the information in this chapter applies to both serial and parallel jobs. For more specific information on parallel jobs, see “Chapter 6. Administration Tasks for Parallel Jobs” on page 149.

Overview

After installing LoadLeveler, you need to customize it by modifying both the *administration* file and the *configuration* file. The administration file optionally lists and defines the machines in the LoadLeveler cluster and the characteristics of classes, users, and groups. The configuration file contains many parameters that you can set or modify that will control how LoadLeveler operates.

In order to easily manage LoadLeveler, you should have only one administration file and one global configuration file, centrally located on a machine in the LoadLeveler cluster. Every other machine in the cluster must be able to read the administration and configuration file that are located on the central machine. LoadLeveler does not prevent you from having multiple copies of administration files but you need to be sure to update all the copies whenever you make a change to one. Having only one administration file prevents any confusion.

You can, however, have multiple local configuration files that specify information specific to individual machines. For more information on the global and local configuration files, refer to “Configuring LoadLeveler” on page 97.

Before working with these two files, you should read the following planning considerations to help you decide how to modify the files.

Planning Considerations

Node availability

Some workstation owners might agree to accept LoadLeveler jobs only when they are not using the workstation themselves. Using LoadLeveler keywords, these workstations can be configured to be available at designated times only.

Common name space

To run jobs on any machine in the LoadLeveler cluster, a user needs the same uid (the system ID number for a user) and gid (the system ID number for a group) on every machine in the cluster. The term cluster refers to all machines mentioned in the configuration file.

For example, if there are two machines in your LoadLeveler cluster, *machine_1* and *machine_2*, user `john` must have the same user ID and login group ID in the **/etc/passwd** file on both machines. If user `john` has user ID 1234 and login group ID 100 on *machine_1*, then user `john` must have the same user ID and login group ID in **/etc/passwd** on *machine_2*. This ensures that the **getuid** system call returns the same user ID on both systems. (This allows a job to run with the same group ID and user ID of the person who submitted the job.)

If you do not have a user ID on one machine, your jobs will not run on that machine. Also, many commands, such as **llq**, will not work correctly if a user does not have a user ID on the central manager machine.

However, there are cases where you may choose to not give a user a login ID on a particular machine. For example, a user does not need an ID on every submit-only machine; the user only needs to be able to submit jobs from at least one such machine. Also, you may choose to restrict a user's access to a schedd machine that is not a public scheduler; again, the user only needs access to at least one schedd machine.

Performance
You should keep the **log**, **spool**, and **execute** directories in a local file system in order to maximize performance. Also, to measure the performance of your network, consider using one of the available products, such as Toolbox/6000.

Management
Managing distributed software systems is a primary concern for all system administrators. Allowing users to share filesystems to obtain a single, network-wide image, is one way to make managing LoadLeveler easier.

Resource Handling
Some nodes in the LoadLeveler cluster might have special software installed that users might need to run their jobs successfully. You should configure LoadLeveler to distinguish those nodes from other nodes using, for example, machine features.

Where to Begin?

Setting up LoadLeveler involves defining machines, users, and how they interact, in such a way that LoadLeveler is able to run jobs quickly and efficiently. If you have a good deal of experience in system administration and job scheduling, you should begin by reading "Expert". If you are relatively new to job scheduling tasks, begin by reading "Intermediate or Beginner".

No matter what your level of experience, it will prove worthwhile to read all the information in this chapter at some point to help you optimize LoadLeveler's performance.

Intermediate or Beginner

If you are experienced in UNIX system administration but are unfamiliar with job scheduling systems or your experience is limited, you may want to start with the section "Administration File Structure and Syntax" on page 74 and read to the end of this chapter. This section provides a relatively slow, step-by-step approach to administering LoadLeveler. If you would rather start up LoadLeveler quickly using mostly default characteristics, follow the procedures in "Quick Set Up" on page 73.

Expert

If you are very familiar with UNIX system administration and job scheduling, and have some idea how you want to distribute your workload, go to "Quick Set Up" on page 73. Each step in this short procedure refers you to a detailed discussion of the task at hand. The sample configuration and administration files included in the samples subdirectory also provide assistance.

If you plan to run interactive jobs using the Parallel Operating Environment (POE) running under LoadLeveler, see "Allowing Users to Submit Interactive POE Jobs" on page 149.

Quick Set Up

If you are very familiar with UNIX system administration and job scheduling, follow the steps listed in this section to get LoadLeveler up and running on your network quickly in a default configuration. This default configuration will merely enable you to submit serial jobs; for a more complex setup, you will have to consult the rest of this manual. This section also does not address how to configure DCE. For more information about configuring DCE for LoadLeveler, see “Step 16: Configuring LoadLeveler to use DCE Security Services” on page 123. For this set up, it is recommended that you use **loadl** as the LoadLeveler user ID. Afterward, you can fine tune your configuration for greater efficiency when you become more familiar with the details of LoadLeveler.

1. Ensure that the installation procedure has completed successfully and that the configuration file, **LoadL_config**, exists in LoadLeveler's home directory or in the directory specified in **/etc/LoadL.cfg** (if this file exists). See “Configuring LoadLeveler” on page 97 for more information.
2. Identify yourself as the LoadLeveler administrator in the **LoadL_config** file using the **LOADL_ADMIN** keyword. The syntax of this keyword is:

   LOADL_ADMIN = *list of user names* **(required)**
   where *list of user names* is a blank-delimited list of those individuals who will have administrative authority.

   Refer to “Step 1: Define LoadLeveler Administrators” on page 99 for more information.
3. Define a machine to act as the LoadLeveler central manager by coding one machine stanza as follows in the administration file, which is called **LoadL_admin**. (Replace *machinename* with the actual name of the machine.)

   ```
   machinename: type = machine
   central_manager = true
   ```

   Do not specify more than one machine as the central manager. Also, if during installation, you ran **llinit** with the **-cm** flag, the central manager is already defined in the **LoadL_admin** file because the **llinit** command takes parameters you entered and updates the administration and configuration files. See “Step 1: Specify Machine Stanzas” on page 75 for more information.
4. Issue the following command for each machine to be included in the LoadLeveler cluster. (Replace *hostname* with the actual name of the machine.)

   ```
   llctl -h hostname start
   ```

   Issue this command for the central manager machine first. See “llctl - Control LoadLeveler Daemons” on page 175 for more information.

   You can also issue the following command to start LoadLeveler on all machines beginning with the central manager. Before you issue this command, make sure all the machines are listed in the administration file. This command only affects machines that are defined in the administration file.

   ```
   llctl -g start
   ```

llctl uses **rsh** or **remsh** to start LoadLeveler on the target machine. Therefore, the administrator using **llctl** must have rsh authority on the target machine.

Administering LoadLeveler

This section explains how to perform administration tasks, and includes a step-by-step approach to administering LoadLeveler in “Customizing the Administration File” on page 75.

Administration File Structure and Syntax

The administration file is called **LoadL_admin** and it lists and defines the *machine*, *user*, *class*, *group*, and *adapter* stanzas.

Machine stanza
: Defines the roles that the machines in the LoadLeveler cluster play. See “Step 1: Specify Machine Stanzas” on page 75 for more information.

User stanza
: Defines LoadLeveler users and their characteristics. See “Step 2: Specify User Stanzas” on page 81 for more information.

Class stanza
: Defines the characteristics of the job classes. See “Step 3: Specify Class Stanzas” on page 84 for more information.

Group stanza
: Defines the characteristics of a collection of users that form a LoadLeveler group. See “Step 4: Specify Group Stanzas” on page 93 for more information.

Adapter stanza
: Defines the network adapters available on the machines in the LoadLeveler cluster. See “Step 5: Specify Adapter Stanzas” on page 95 for more information.

Stanzas have the following general format:

```
label: type = type_of_stanza
keyword1 = value1
keyword2 = value2
  ...
```

Figure 23. Format of Administration File Stanzas

The following is a simple example of an administration file illustrating several stanzas:

```
machine_a: type = machine
      central_manager = true       # defines this machine as the central manager
      adapter_stanzas = adapter_a  # identifies an adapter stanza

class_a: type = class
      priority = 50    # priority of this class

user_a: type  = user
      priority  = 50   # priority of this user

group_a: type = group
      priority  = 50   # priority of this group

adapter_a: type = adapter
      adapter_name = en0  #defines an adapter
```

Figure 24. Sample Administration File Stanzas

The characteristics of a stanza are:

- Every stanza has a label associated with it. The label specifies the name you give to the stanza.
- Every stanza has a **type** field that specifies it as a user, class, machine, group, or adapter stanza.
- New line characters are ignored. This means that separate parts of a stanza may be included on the same line. However, it is not recommended to have parts of a stanza cross line boundaries.
- White space is ignored, other than to delimit keyword identifiers. This eliminates confusion between tabs and spaces at the beginning of lines.
- A cross-hatch sign (#) identifies a comment and may appear anywhere on the line. All characters following this sign on that line are ignored.
- Multiple stanzas of the same label are allowed, but only the first label is used.
- Default stanzas specify the default values for any keywords which are not otherwise specified. Each stanza type can have an associated default stanza. A default stanza must appear in the administration file ahead of any specific stanza entries of the same type. For example, a default class stanza must appear ahead of any specific class stanzas you enter.

Customizing the Administration File

You can add as many stanzas as you would like to the administration file. This section tells you how to modify this file in a step-by-step manner. You do not have to perform the steps in the order that they appear here.

Step 1: Specify Machine Stanzas

The information in a machine stanza defines the characteristics of that machine. You do not have to specify a machine stanza for every machine in the LoadLeveler cluster but you must have one machine stanza for the machine that will serve as the central manager.

If you do not specify a machine stanza for a machine in the cluster, the machine and the central manager still communicate and jobs are scheduled on the machine but the machine is assigned the default values specified in the default machine stanza. If there is no default stanza, the machine is assigned default values set by LoadLeveler.

Any machine name used in the stanza must be a name which can be resolved to an IP address. This name is referred to as an interface name because the name can be used for a program to interface with the machine. Generally, interface names match the machine name, but they do not have to.

By default, LoadLeveler will append the DNS domain name to the end of any machine name without a domain name appended before resolving its address. If you specify a machine name without a domain name appended to it and you do not want LoadLeveler to append the DNS domain name to it, specify the name using a trailing period. You may have a need to specify machine names in this way if you are running a cluster with more than one nameserving technique. For example, if you are using a DNS nameserver and running NIS, you may have some machine names which are resolved by NIS which you do not want LoadLeveler to append DNS names to. In situations such as this, you also want to specify **name_server** keyword in your machine stanzas.

Under the following conditions, you must have a machine stanza for the machine in question:

- If you set the **MACHINE_AUTHENTICATE** keyword to **true** in the configuration file, then you must create a machine stanza for each node that LoadLeveler includes in the cluster.
- If the machine's hostname (the name of the machine returned by the UNIX hostname command) does not match an interface name. In this case, you must specify the interface name as the machine stanza name and specify the machine's hostname using the **alias** keyword.
- If the machine's hostname does match an interface name but not the correct interface name.

Machine stanzas take the following format. Default values for keywords appear in bold:

```
label: type = machine
adapter_stanzas = stanza_list
alias = machine_name
central_manager = true | false | alt
cpu_speed_scale = true | false
dce_host_name = dce hostname
machine_mode = batch | interactive | general
master_node_exclusive = true | false
max_adapter_windows = [all | none | <+> n | -n ]
max_jobs_scheduled = number
name_server = list
pvm_root = pathname
pool_list = pool_numbers
resources = name(count) name(count) ... name(count)
schedd_fenced = true | false
schedd_host = true | false
spacct_excluse_enable = true | false
speed = number
submit_only = true | false
```

Figure 25. Format of a Machine Stanza

You can specify the following keywords in a machine stanza:

adapter_stanzas = *stanza_list*
where *stanza_list* is a blank-delimited list of one or more adapter stanza names

which specify adapters available on this machine. All adapter stanzas you define must be specified on this keyword.

alias = *machine_name*
where *machine_name* is a blank-delimited list of one or more machine names. Depending upon your network configurations, you may need to add **alias** keywords for machines that have multiple interfaces.

Note: In general, if your cluster is configured with machine hostnames which match the hostnames corresponding to the IP address configured for the LAN adapters which LoadLeveler is expected to use, you will not have to specify the **alias** keyword. For example, if all of the machines in your cluster are configured like this sample machine, you should not have to specify the **alias** keyword.

```
Machine porsche.kgn.ibm.com
```

- The hostname command returns porsche.kgn.ibm.com.
- The Ethernet adapter address 129.40.8.20 resolves to hostname porsche.kgn.ibm.com.

However, if any machine in your cluster is configured like either of the following two sample machines, then you will have to specify the **alias** keyword for those machines:

1. Machine `yugo.kgn.ibm.com`
   - The hostname command returns yugo.kgn.ibm.com.
   - The Ethernet adapter address 129.40.8.21 resolves to hostname chevy.kgn.ibm.com.
   - No adapter address resolves to yugo.

   You need to code the machine stanza as:

   ```
   chevy: type = machine
   alias = yugo
   ```
2. Machine `rover.kgn.ibm.com`
   - The hostname command returns rover.kgn.ibm.com.
   - The FDDI adapter address 129.40.9.22 resolves to hostname rover.kgn.ibm.com.
   - The Ethernet adapter address 129.40.8.22 resolves to hostname bmw.kgn.ibm.com.
   - No route exists via the FDDI adapter to the clusters central manager machine.
   - A route exists from this machine to the central manager via the Ethernet adapter.

   You need to code the machine stanza as:

   ```
   bmw:   type = machine
   alias = rover
   ```

central_manager = true**| false |** alt
where **true** designates this machine as the LoadLeveler central manager host, where the negotiator daemon runs. You must specify one and only one machine stanza identifying the central manager. For example:

```
machine_a: type = machine
central_manager = true
```

false specifies that this machine is not the central manager.

alt specifies that this machine can serve as an alternate central manager in the event that the primary central manager is not functioning. For more information on recovering if the primary central manager is not operating, refer to “What Happens if the Central Manager Isn’t Operating?” on page 309. Submit-only machines cannot have their machine stanzas set to this value.

If you are going to select machines to serve as alternate central managers, you should look at the following keywords in the configuration file:

- **CENTRAL_MANAGER_HEARTBEAT_INTERVAL**
- **CENTRAL_MANAGER_TIMEOUT**

For information on setting these keywords, see “Step 10: Specify Alternate Central Managers” on page 111.

cpu_speed_scale = true**| false**
where **true** specifies that CPU time (which is used, for example, in setting limits, in accounting information, and reported by the **llq -x** command), is in normalized units for each machine. **false** specifies that CPU time is in native units for each machine. For an example of using this keyword to normalize accounting information, see “Task 5: Specifying Machines and Their Weights” on page 157.

dce_host_name = *dce hostname*
where *dce hostname* is the dce hostname of this machine. Execute either ″**SDRGetObjects Node dcehostname,**″ or ″**llextSDR**″ to obtain a listing of DCE hostnames of nodes on an SP system.

machine_mode = batch **|** interactive **| general**
Specifies the type of job this machine can run. Where:

batch Specifies this machine can run only batch jobs.

interactive
Specifies this machine can run only interactive jobs. Only POE is currently enabled to run interactively.

general
Specifies this machine can run both batch jobs and interactive jobs.

master_node_exclusive = true**| false**
where **true** specifies that this machine is used only as a master node for parallel jobs.

max_adapter_windows = [all | none | <+>*n* **|** *-n* **]**
This keyword specifies how many of a machine’s available adapter windows LoadLeveler can use. The default value is **all**, which specifies that LoadLeveler can reserve all of the windows which are not already reserved by other applications. The value **none** indicates that LoadLeveler can not use any windows (consequently, no user space jobs will be dispatched to that machine). A positive number (specified, with or without the plus sign), means that LoadLeveler can use no more than the specified number of windows; however, LoadLeveler may use less than the specified number if fewer windows are actually available on the machine’s adapter. A negative number means that LoadLeveler will use all but the specified number of the available windows (e.g., *-n* means that LoadLeveler will reserve *n* windows for use by other applications).

max_jobs_scheduled = *number*
where *number* is the maximum number of jobs submitted from this scheduling (schedd) machine that can run (or start running) in the LoadLeveler cluster at one time. If *number* of jobs are already running, no other jobs submitted from

this machine will run, even if resources are available in the LoadLeveler cluster. When one of the running jobs completes, any waiting jobs then become eligible to be run. The default is -1, which means there is no maximum.

name_server = *list*
where *list* is a blank-delimited list of character strings that is used to specify which nameserver(s) are used for the machine. Valid strings are DNS, NIS, and LOCAL. LoadLeveler uses the list to determine when to append a DNS domain name for machine names specified in LoadLeveler commands issued from the machine described in this stanza.

If DNS is specified alone, LoadLeveler will always append the DNS domain name to machine names specified in LoadLeveler commands. If NIS or LOCAL is specified, LoadLeveler will never append a DNS domain name to machine names specified in LoadLeveler commands. If DNS is specified with either NIS or LOCAL, LoadLeveler will always look up the name in the administration file to determine whether to append a DNS domain name. If the name is specified with a trailing period, it doesn't append the domain name.

pvm_root = *pathname*
Where *pathname* specifies the location of the directory in which PVM is installed. The default pathname is **$HOME/pvm3**.

pool_list = *pool_numbers*
Where *pool_numbers* is a blank-delimited list of non-negative numbers identifying pools to which the machine belongs. These numbers may be any positive integers including zero. This keyword provides compatability with function that was previously part of the Resource Manager.

resources = *name(count) name(count) ... name(count)*

Specifies quantities of the consumable resources initially available on the machine. Where *name(count)* is an administrator-defined name and count, or could also be **ConsumableCpus***(count)*, **ConsumableMemory***(count units)*, or **ConsumableVirtualMemory***(count units)*. **ConsumableMemory** and **ConsumableVirtualMemory** are the only two consumable resources that can be specified with both a count and units. The count for each specified resource must be an integer greater than or equal to zero, with three exceptions: **ConsumableCpus**, and **ConsumableMemory** must be specified with a value which is greater than zero, and **ConsumableVirtualMemory** must be specified with a value greater than 0, and greater than or equal to the **image_size**. The allowable units are those normally used with LoadLeveler data limits:

```
b bytes
w words
kb kilobytes (2** 10 bytes)
kw kilowords (2** 10 words)
mb megabytes (2** 20 bytes)
mw megawords (2**20 words)
gb gigabytes (2** 30 bytes)
gw gigawords (2** 30 words)
```

ConsumableMemory and **ConsumableVirtualMemory** values are stored in mb (megabytes) and rounded up. Therefore, the smallest amount of **ConsumableMemory** or **ConsumableVirtualMemory** which you can request is one megabyte. If no units are specified, then megabytes are assumed. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not effect the scheduling of the job.

schedd_fenced = true **| false**
where **true** specifies that the central manager ignores connections from the schedd daemon running on this machine. Use the **true** setting in conjunction with the **llctl -h host purgeschedd** command when you want to attempt to recover resources lost when a node running the schedd daemon fails. A **true** setting prevents conflicts from arising when a schedd machine is restarted while a purge is taking place. For more information, see "How Do I Recover Resources Allocated by a schedd Machine?" in the *LoadLeveler Diagnosis and Messages Guide*.

schedd_host = true **| false**
where **true** designates this as a public scheduling machine used to receive job submissions from submit-only machines, or for accepting jobs from machines which run stard but not schedd daemons. Submit-only machines do not run LoadLeveler jobs.

spacct_excluse_enable = true **| false**
Where **true** specifies that the accounting function on an SP system is informed that a job step has exclusive use of this machine. Note that your SP system must have exclusive user accounting enabled in order for this keyword to have an effect. For more information on SP accounting, see *Parallel System Support Programs for AIX: Administration Guide*, GC23-3899.

speed = *number*
where *number* is a floating point number that is used for machine scheduling purposes in the **MACHPRIO** expression. For more information on machine scheduling and the MACHPRIO expression, see "Step 7: Prioritize the Order of Executing Machines Maintained by the Negotiator" on page 106. In addition, the **speed** keyword is also used to define the weight associated with the machine. This weight is used when gathering accounting data on a machine basis. The default is 1.0.

The following example illustrates how the **speed** keyword can be used for assigning weights to machines.

If your cluster consisted of five RISC System/6000 machines that you want to have the same weight, you would not have to specify this keyword in the administration file. By default, all machines would have a weight of 1.0. If, however, you add an SP system to your cluster for parallel job processing, you may want to update the local configuration file for each node of the SP system to charge differently for resource consumption on those nodes. You would need to set the **speed** keyword to something other than 1.0 to make the SP nodes have a different weight.

For information on how the **speed** keyword can be used to schedule machines, refer to "Step 7: Prioritize the Order of Executing Machines Maintained by the Negotiator" on page 106.

submit_only = true**| false**
where **true** designates this as a submit-only machine. If you set this keyword to **true**, in the administration file set **central_manager** and **schedd_host** to **false**.

Examples of Machine Stanzas

Example 1: In this example, the machine is being defined as the central manager.

```
#
machine_a: type = machine
central_manager = true    # central manager runs here
```

Example 2: This example sets up a submit-only node. Note that the **submit-only** keyword in the example is set to **true**, while the **schedd_host** keyword is set to **false**. You must also ensure that you set the **schedd_host** to **true** on at least one other node in the cluster.

```
#
machine_b: type = machine
central_manager = false   # not the central manager
schedd_host = false       # not a scheduling machine
submit_only = true        # submit only machine
alias = machineb          # interface name
```

Example 3: In the following example, machine_c is the central manager, has an alias associated with it, and can run parallel PVM jobs:

```
#
machine_c: type = machine
central_manager = true    # central manager runs here
schedd_host = true        # defines a public scheduler
alias = brianne
pvm_root = /u/brianne/loadl/1.2.0/aix32/pvm3
```

Step 2: Specify User Stanzas

The information specified in a user stanza defines the characteristics of that user. You can have one user stanza for each user but this is not necessary. If an individual user does not have their own user stanza, that user uses the defaults defined in the default user stanza.

User stanzas take the following format:
You can specify the following keywords in a user stanza:

label: **type = user**
account = *list*
default_class = *list*
default_group = *group name*
default_interactive_class = *class name*
maxidle = *number*
maxjobs = *number*
maxqueued = *number*
max_node = *number*
max_processors = *number*
priority = *number*
total_tasks = *number*

Figure 26. Format of a User Stanza

account =*list*
where *list* is a blank-delimited list of account numbers that identifies the account numbers a user may use when submitting jobs. The default is a null list.

default_class = *list*
where *list* is a blank-delimited list of class names used for jobs which do not include a **class** statement in the job command file. If you specify only one default class name, this class is assigned to the job. If you specify a list of default class names, LoadLeveler searches the list to find a class which satisfies the resource limit requirements. If no class satisfies these requirements, LoadLeveler rejects the job.

Suppose a job requests a CPU limit of 10 minutes. Also, suppose the default class list is `default_class = short long`, where `short` is a class for jobs up to

five minutes in length and `long` is a class for jobs up to one hour in length. LoadLeveler will select the `long` class for this job because the `short` class does not have sufficient resources.

If no **default_class** is specified in the user stanza, or if there is no user stanza at all, then jobs submitted without a **class** statement are assigned to the **default_class** that appears in the default user stanza. If you do not define a **default_class**, jobs are assigned to the class called **No_Class**.

default_group = *group_name*
where *group_name* is the default group assigned to jobs submitted by the user. If a **default_group** statement does not appear in the user stanza, or if there is no user stanza at all, then jobs submitted by the user without a **group** statement are assigned to the **default_group** that appears in the default user stanza. If you do not define a **default_group**, jobs are assigned to the group called **No_Group**.

If you specify **default_group = Unix_Group**, LoadLeveler sets the user's LoadLeveler group to his or her primary UNIX group (as defined in the **/etc/passwd** file).

default_interactive_class = *class_name*
where *class_name* is the class to which an interactive job submitted by this user is assigned if the user does not specify a class using the LOADL_INTERACTIVE_CLASS environment variable. You can specify only one default interactive class name.

If you do not set a **default_interactive_class** value in the user stanza, or if there is no user stanza at all, then interactive jobs submitted without a **class** statement are assigned to the **default_interactive_class** that appears in the default user stanza. If you do not define a **default_interactive_class**, interactive jobs are assigned to the class called **No_Class**.

See "Example 2" on page 84 for more information on how LoadLeveler assigns a default interactive class to jobs.

maxidle = *number*
where *number* is the maximum number of idle jobs this user can have in queue. That is, *number* is the maximum number of jobs which the negotiator will consider for dispatch for the user. Jobs above this maximum are placed in the NotQueued state. This prevents individual users from dominating the number of jobs that are either running or are being considered to run. If the user stanza does not specify **maxidle** or if there is no user stanza at all, the maximum number of jobs that can be simultaneously in queue for the user is defined in the default stanza. If no value is found, or the limit found is -1, then no limit is placed on the number of jobs that can be simultaneously idle for the user.

For more information, see "Controlling the Mix of Idle and Running Jobs" on page 314.

maxjobs = *number*
where *number* is the maximum number of jobs this user can run at any time. If the user stanza does not specify **maxjobs** or if there is no user stanza at all, the maximum jobs that can be simultaneously run by the user is defined in the default stanza. The default is -1, which means no limit is placed on the number of jobs that can simultaneously run for the user. Regardless of this limit, there is no limit to the number of jobs a user can submit.

For more information, see “Controlling the Mix of Idle and Running Jobs” on page 314 .

maxqueued = *number*
where *number* is the maximum number of jobs allowed in the queue for this user. This is the maximum number of jobs which can be either running or being considered to be dispatched by the negotiator for that user. Jobs above this maximum are placed in the NotQueued state. This prevents individual users from dominating the number of jobs that are either running or are being considered to run. If no **maxqueued** is specified in the user stanza, or if there is no user stanza, the maximum number of jobs that can simultaneously be in the queue is defined in the default stanza. The default is -1, which means that no limit is placed on the number of jobs that can simultaneously be in the job queue for that user. Regardless of this limit, there is no limit to the number of jobs a user can submit.

For more information, see “Controlling the Mix of Idle and Running Jobs” on page 314.

max_node = *number*
where *number* specifies the maximum number of nodes this user can request for a parallel job in a job command file using the **node** keyword. The default is -1, which means there is no limit. The max_node keyword will not affect the use of the min_processors and max_processors keywords in the job command file.

max_processors = *number*
where *number* specifies the maximum number of processors this user can request for a parallel job in a job command file using the **min_processors** and **max_processors** keywords. The default is -1, which means there is no limit.

priority = *number*
where *number* is a integer that specifies the priority for jobs submitted by the user. The default is 0. The number specified for priority is referenced as **UserSysprio** in the configuration file. **UserSysprio** can be used in the assignment of job priorities. If the variable **UserSysprio** does not appear in the SYSPRIO expression in the configuration file, the priority numbers for users specified here in the administration file have no effect. See “Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105 for more information about the **UserSysprio** keyword.

total_tasks = *number*
where *number* specifies the maximum number of tasks this user can request for a parallel job in a job command file using the **total_tasks** keyword. The default is -1, which means there is no limit.

Examples of User Stanzas

Example 1: In this example, user `fred` is being provided with a user stanza. His jobs will have a user priority of 100. If he does not specify a job class in his job command file, the default job class **class_a** will be used. In addition, he can have a maximum of 15 jobs running at the same time.

```
# Define user stanzas
fred:  type = user
priority = 100
default_class = class_a
maxjobs = 15
```

Example 2: This example explains how a default interactive class for a parallel job is set by presenting a series of user stanzas and class stanzas. This example assumes that users do not specify the LOADL_INTERACTIVE_CLASS environment variable.

```
default: type =user
         default_interactive_class = red
         default_class = blue

carol:   type = user
         default_class = single double
         default_interactive_class = ijobs

steve:   type = user
         default_class = single double

ijobs:   type = class
         wall_clock_limit = 08:00:00

red:     type = class
         wall_clock_limit = 30:00
```

If the user Carol submits an interactive job, the job is assigned to the default interactive class called **ijobs**. The job is assigned a wall clock limit of 8 hours. If the user Steve submits an interactive job, the job is assigned to the **red** class from the default user stanza. The job is assigned a wall clock limit of 30 minues.

Example 3: In this example, Jane's jobs have a user priority of 50, and if she does not specify a job class in her job command file the default job class **small_jobs** is used. This user stanza does not specify the maximum number of jobs that Jane can run at the same time so this value defaults to the value defined in the default stanza. Also, suppose Jane is a member of the primary UNIX group "staff." Jobs submitted by Jane will use the default LoadLeveler group "staff." Lastly, Jane can use three different account numbers.

```
# Define user stanzas
jane:  type = user
priority = 50
default_class = small_jobs
default_group = Unix_Group
account = dept10 user3 user4
```

Step 3: Specify Class Stanzas

The information in a class stanza defines characteristics for that class. Class stanzas are optional. Class stanzas take the following format. Default values for keywords appear in bold.

```
label: type = class
admin= list
class_comment = "string"
default_resources = name(count) name(count)...name(count)
exclude_groups = list
exclude_users = list
include_groups = list
include_users = list
master_node_requirement = true | false
maxjobs = number
max_node = number
max_processors = number
nice = value
NQS_class = true | false
NQS_submit = name
NQS_query = queue names
priority = number
total_tasks = number
core_limit = hardlimit,softlimit
cpu_limit = hardlimit,softlimit
data_limit = hardlimit,softlimit
file_limit = hardlimit,softlimit
job_cpu_limit = hardlimit,softlimit
rss_limit = hardlimit,softlimit
stack_limit = hardlimit,softlimit
wall_clock_limit = hardlimit,softlimit
```

Figure 27. Format of a Class Stanza

You can specify the following keywords in a class stanza:

admin = *list*
where *list* is a blank-delimited list of administrators for this class. These administrators can hold, release, and cancel jobs in this class.

class_comment = *"string"*
where *string* is text characterizing the class. This information appears when the user is building a job command file using the GUI and requests Choice information on the classes to which he or she is authorized to submit jobs. The length of the string cannot exceed 1024 characters.

default_resources = *name(count) name(count)...name(count)*

Specifies the default amount of resources consumed by a task of a job step, provided that no **resources** keyword is coded for the step in the job command file. If a resources keyword is coded for a job step, then it overrides any default_resources associated with the associated job class. The syntax is:

```
resources=name(count) name(count) ... name(count)
```

where *name(count)* could also be **ConsumableMemory***(count units)* or **ConsumableVirtualMemory***(count units)*. **ConsumableMemory** and **ConsumableVirtualMemory** are the only two consumable resources that can be specified with both a count and units. The count for each specified resource must be an integer greater than or equal to zero, with three exceptions: **ConsumableCpus**, and **ConsumableMemory** must be specified with a value which is greater than zero, and **ConsumableVirtualMemory** must be specified with a value greater than 0, and greater than or equal to the **image_size**. If the count is not valid, then LoadLeveler will issue an error message, and will not submit the job. The allowable units are those normally used with LoadLeveler data limits:

```
b bytes
w words
kb kilobytes (2** 10 bytes)
kw kilowords (2** 10 words)
mb megabytes (2** 20 bytes)
mw megawords (2**20 words)
gb gigabytes (2** 30 bytes)
gw gigawords (2** 30 words)
```

ConsumableMemory and **ConsumableVirtualMemory** values are stored in mb (megabytes) and rounded up. Therefore, the smallest amount of **ConsumableMemory** or **ConsumableVirtualMemory** which you can request is one megabyte. If no units are specified, then megabytes are assumed. However, **image size** units are in kilobytes. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not effect the scheduling of the job. If the **resources** keyword is not specified in the job step, then the **default_resources** (if any) defined in the administration file for the class will be used for each task of the job step.

exclude_groups = *list*
where *list* is a blank-delimited list of groups who are *not* allowed to submit jobs of that *class name*. Do not specify both a list of included groups and a list of excluded groups. Only one of these may be used for any class. The default is that no groups are excluded.

exclude_users = *list*
where *list* is a blank-delimited list of users who are *not* permitted to submit jobs of that *class name*. Do not specify both a list of included users and a list of excluded users. Only one of these may be used for any class. The default is that no users are excluded.

include_groups = *list*
where *list* is a blank-delimited list of groups who are allowed to submit jobs of that *class name*. If provided, this list limits groups of that class to those on the list. Do not specify both a list of included groups and a list of excluded groups. Only one of these may be used for any class. The default is to include all groups.

include_users = *list*
where *list* is a blank-delimited list of users who are permitted to submit jobs of that *class name*. If provided, this list limits users of that class to those on the list. Do not specify both a list of included users and a list of excluded users. Only one of these may be used for any class. The default is to include all users.

master_node_requirement = true**|false**
where **true** specifies that parallel jobs in this class require the master node feature. For these jobs, LoadLeveler allocates the first node (called the "master") on a machine having the **master_node_exclusive = true** setting in its machine stanza. If most or all of your parallel jobs require this feature, you should consider placing the statement **master_node_requirement = true** in your default class stanza. Then, for classes that do not require this feature, you can use the statement **master_node_requirement = false** in their class stanzas to override the default setting. One machine per class should have the **true** setting; if more than one machine has this setting, normal scheduling selection is performed.

maxjobs = *number*
where *number* is the maximum number of jobs that can run in this class. If the class stanza does not specify **maxjobs**, or if there is no class stanza at all, the

maximum jobs that can be simultaneously run in this class is defined in the default stanza. The default is -1, which means that no limit is placed on the number of jobs a user can submit.

max_processors = *number*
where *number* specifies the maximum number of processors a user submitting jobs to this class can request for a parallel job in a job command file using the **min_processors** and **max_processors** keywords. The default is -1 which means that there is no limit.

max_node = *number*
where *number* specifies the maximum number of nodes a user submitting jobs in this class can request for a parallel job in a job command file using the **node** keyword. The default is -1, which means there is no limit. The **max_node** keyword will not affect the use of the **min_processors** and **max_processors** keywords in the job command file.

nice = *value*
where *value* is the amount by which the current UNIX *nice* value is incremented. The *nice* value is one factor in a job's run priority. The lower the number, the higher the run priority. If two jobs are running on a machine, the *nice* value determines the percentage of the CPU allocated to each job.

This value ranges from -20 to 20. Values out of this range are placed at the top (or bottom) of the range. For example, if your current *nice* value is 15, and you specify `nice = 10`, the resulting value is 20 (the upper limit) rather than 25. The default is 0.

For more information, consult the appropriate UNIX documentaion.

NQS_class = true**|false**
When **true**, any job submitted to this class will be routed to an NQS machine.

NQS_submit = *name*
where *name* is the name of the NQS pipe queue to which the job will be routed. When the job is dispatched to LoadLeveler, LoadLeveler will invoke the **qsub** command using the name of this queue. There is no default.

NQS_query = *queue names*
where *queue names* is a blank-delimited list of queue names (including host names if necessary) to be used with the **qstat** command to monitor the job and with the **qdel** command to cancel the job. There is no default.

For more information on routing jobs to machines running NQS, refer to Figure 31 on page 159

priority = *number*
where *number* is an integer that specifies the priority for jobs in this class. The default is 0. The number specified for priority is referenced as **ClassSysprio** in the configuration file. You can use **ClassSysprio** when assigning job priorities. If the variable **ClassSysprio** does not appear in the SYSPRIO expression, then the priority specified here in the administration file is ignored. See "Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105 for more information about the **ClassSysprio** keyword.

total_tasks = *number*
where *number* specifies the maximum number of tasks a user submitting jobs in this class can request for a parallel job in a job command file using the **total_tasks** keyword. The default is -1, which means there is no limit.

Limit Keywords

The class stanza includes the following **limit** keywords, which allow you to control the amount of resources used by a job step or a job process.

Table 10. Types of Limit Keywords

Limit	How It Is Enforced
core_limit	Per process
cpu_limit	Per process
data_limit	Per process
file_limit	Per process
job_cpu_limit	Per job step
rss_limit	Per process
stack_limit	Per process
wall_clock_limit	Per job step

Individual keywords are described in “Specifying Limits in the Class Stanza” on page 90. The following section gives you a general overview of limits.

Overview of Limits: A limit is the amount of a resource that a job step or a process is allowed to use. (A process is a dispatchable unit of work.) A job step may be made up of several processes.

Limits include both a **hard limit** and a **soft limit**. When a hard limit is exceeded, the job is usually terminated. When a soft limit is exceeded, the job is usually given a chance to perform some recovery actions. For more information, see “Exceeding Limits”.

Limits are enforced either per process or per job step, depending on the type of limit. For parallel jobs steps, which consist of multiple tasks running on multiple machines, limits are enforced on a per task basis.

For example, a common limit is the **cpu_limit**, which limits the amount of CPU time a single process can use. If you set **cpu_limit** to five hours and you have a job step that forks five processes, each process can use up to five hours of CPU time, for a total of 25 CPU hours. Another limit that controls the amount of CPU used is **job_cpu_limit**. This is the total amount of CPU that the entire serial job step can use. If you impose a **job_cpu_limit** of five hours, the entire job step (made up of all five processes) cannot consume more than five CPU hours.

You can specify limits in either the class stanza of the administration file or in the job command file. The lowest of these two limits will be used to run the job. If the class limit is used the job will be started regardless of the users system limit.

Exceeding Limits: Process limits are enforced by the operating system. Job step limits are enforced by LoadLeveler.

Exceeding Job Step Limits: When a hard limit is exceeded LoadLeveler sends a *non-trappable* signal to the process (except in the case of a parallel job). When a soft limit is exceeded, LoadLeveler sends a *trappable* signal to the process. The following chart summarizes the actions that occur when a job step limit is exceeded:

Table 11. Exceeding Job Step Limits

Type of Job	When a Soft Limit is Exceeded	When a Hard Limit is Exceeded
Serial	SIGXCPU or SIGKILL issued	SIGKILL issued
Parallel (non-PVM)	SIGXCPU issued to both the user program and to the parallel daemon	SIGTERM issued
PVM	SIGXCPU issued to the user prgram	**pvm_halt** invoked to shut down PVM

On systems that do not support SIGXCPU, LoadLeveler does not distinguish between hard and soft limits. When a soft limit is reached on these platforms, LoadLeveler issues a SIGKILL.

Exceeding Per Process Limits: For per process limits, what happens when your job reaches and exceeds either the soft limit or the hard limit depends on the operating system you are using.

Note that when a job forks a process which exceeds a per process limit, such as the CPU limit, the operating system (and not LoadLeveler) terminates the process by issuing a SIGXCPU. As a result, you will not see an entry in the LoadLeveler logs indicating that the process exceeded the limit. The job will complete with a 0 return code. LoadLeveler can only report the status of any processes it has started.

If you need more specific information, refer to your operating system documentation.

Syntax: The syntax for setting a limit is

```
limit_type = hardlimit,softlimit
```

For example:

```
core_limit = 120kb,100kb
```

To specify only a hard limit, you can enter, for example:

```
core_limit = 120kb
```

To specify only a soft limit, you can enter, for example:

```
core_limit = ,100kb
```

In a keyword statement, you cannot have any blanks between the numerical value (100 in the above example) and the units (kb). Also, you cannot have any blanks to the left or right of the comma when you define a limit in a job command file.

For limit keywords that refer to a data limit — such as **data_limit**, **core_limit**, **file_limit**, **stack_limit**, and **rss_limit** — the hard limit and the soft limit are expressed as:

```
integer[.fraction][units]
```

where *integer* and *fraction* represent numerical strings of up to eight characters. *units* can be:

b bytes
w words
kb kilobytes (2^{10} bytes)
kw kilowords (2^{10} words)

mb megabytes (2^{20} bytes)
mw megawords (2^{20} words)
gb gigabytes (2^{30} bytes)
gw gigawords (2^{30} words)

If no units are specified for data limits, then bytes are assumed.

For limit keywords that refer to a time limit — such as **cpu_limit**, **job_cpu_limit**, and **wall_clock_limit** — the hard limit and the soft limit are expressed as:

[[hours:]minutes:]seconds[.fraction]

Fractions are rounded to seconds.

You can use the following character strings with all limit keywords except the **copy** keyword for **wall_clock_limit**:

rlim_infinity
Represents the largest positive number.

unlimited
Has same effect as **rlim_infinity**.

copy Uses the limit currently active when the job is submitted.

See Table 12 for more information on specifying limits.

Table 12. Setting limits

If the hard limit:	Then the:
Is set in both the class stanza and the job command file	Smaller of the two limits is taken into consideration. If the smaller limit is the job limit, the job limit is then compared with the user limit set on the machine that runs the job. The smaller of these two values is used. If the limit used is the class limit, the class limit is used without being compared to the machine limit.
Is not set in either the class stanza or the job command file	User per process limit set on the machine that runs the job is used.
Is set in the job command file and is less than its respective job soft limit	The job is not submitted.
Is set in the class stanza and is less than its respective class stanza soft limit	Soft limit is adjusted downward to equal the hard limit.
Is specified in the job command file	Hard limit must be greater than or equal to the specified soft limit and less than or equal to the limit set by the administrator in the class stanza of the administration file. Note: If the per process limit is not defined in the administration file and the hard limit defined by the user in the job command file is greater than the limit on the executing machine, then the hard limit is set to the machine limit.

Specifying Limits in the Class Stanza: You can specify the following limit keywords:

core_limit = *hardlimit,softlimit*
Specifies the hard limit and/or soft limit for the size of a core file.

Examples:

```
core_limit = unlimited
core_limit = 30mb
```

For more information, see “Overview of Limits” on page 88

cpu_limit = *hardlimit,softlimit*
Specifies hard limit and/or soft limit for the CPU time to be used by each individual process of a job step. For example, if you impose a **cpu_limit** of five hours and you have a job step composed of five processes, each process can consume five CPU hours; the entire job step can therefore consume 25 total hours of CPU.

Examples:

```
cpu_limit = 12:56:21       # hardlimit = 12 hours 56 minutes 21 seconds
cpu_limit = 56:00,50:00    # hardlimit = 56 minutes 0 seconds
# softlimit = 50 minutes 0 seconds
cpu_limit = 1:03           # hardlimit = 1 minute 3 seconds
cpu_limit = unlimited      # hardlimit = 2,147,483,647 seconds
# (X'7FFFFFFF')
cpu_limit = rlim_infinity  # hardlimit = 2,147,483,647 seconds
# (X'7FFFFFFF')
cpu_limit = copy           # current CPU hardlimit value on the
# submitting machine.
```

For more information, see “Overview of Limits” on page 88.

data_limit = *hardlimit,softlimit*
Specifies hard limit and/or soft limit for the data segment to be used by each process of the submitted job.

Examples:

```
data_limit = 125621         # hardlimit = 125621 bytes
data_limit = 5621kb         # hardlimit = 5621 kilobytes
data_limit = 2mb            # hardlimit = 2 megabytes
data_limit = 2.5mw          # hardlimit = 2.5 megawords
data_limit = unlimited      # hardlimit = 2,147,483,647 bytes
#             (X'7FFFFFF')
data_limit = rlim_infinity  # hardlimit = 2,147,483,647 bytes
#             (X'7FFFFFF')
data_limit = copy           # copy data hardlimit value from submitting
                            # machine.
```

For more information, see “Overview of Limits” on page 88.

file_limit = *hardlimit,softlimit*
Specifies the hard limit and/or soft limit for the size of a file. For more information, see “Overview of Limits” on page 88.

job_cpu_limit = *hardlimit,softlimit*
Specifies the maximum total CPU time to be used by all processes of a job step. That is, if a job step forks to produce multiple processes, the sum total of CPU consumed by all of the processes is added and controlled by this limit.

For example:

```
job_cpu_limit = 10000
```

For more information on this keyword, see the **JOB_LIMIT_POLICY** keyword in “Chapter 7. Gathering Job Accounting Data” on page 153. For more general information on limits, see “Overview of Limits” on page 88.

rss_limit = *hardlimit,softlimit*
Specifies the hard limit and/or soft limit for the resident size. For more information, see “Overview of Limits” on page 88.

stack_limit = *hardlimit,softlimit*
Specifies the hard limit and/or soft limit for the size of a stack. For more information, see “Overview of Limits” on page 88.

wall_clock_limit = *hardlimit,softlimit*
Specifies the hard limit and/or soft limit for the elapsed time for which a job can run. Note that LoadLeveler uses the time the negotiator daemon dispatches the job as the start time of the job. When a job is checkpointed, vacated, and then restarted, the **wall_clock_limit** is not adjusted to account for the amount of time that elapsed before the checkpoint occured. This keyword is not supported for NQS jobs. Also, if the startd daemon terminates abnormally with running jobs, any wall clock limits are not supported when the daemon is restarted.

If you are running the Backfill scheduler, you must set a wall clock limit either in the job command file or in a class stanza (for the class associated with the job you submit). LoadLeveler administrators should consider setting a default wall clock limit in a default class stanza. For more information on setting a wall clock limit when using the Backfill scheduler, see “Choosing a Scheduler” on page 100.

For more general information on limits, see “Overview of Limits” on page 88.

Examples of Class Stanzas

Example 1: Creating a Class that Excludes Certain Users:

```
class_a: type=class                # class that excludes users
priority=10                        # ClassSysprio
exclude_users=green judy           # Excluded users
```

Example 2: Creating a Class for Small-Size Jobs:

```
small:  type=class                 # class for small jobs
priority=80                        # ClassSysprio (max=100)
cpu_limit=00:02:00                 # 2 minute limit
data_limit=30mb                    # max 30 MB data segment
default_resources=ConsumbableVirtualMemory(10mb)  # resources consumed by each
ConsumableCpus(1) resA(3) floatinglicenseX(1)     # task of a small job step if
                                   # resources are not explicitly
                                   # specified in the job command file
core_limit=10mb                    # max 10 MB core file
file_limit=50mb                    # max file size 50 MB
stack_limit=10mb                   # max stack size 10 MB
rss_limit=35mb                     # max resident set size 35 MB
include_users = bob sally          # authorized users
```

Example 3: Creating a Class for Medium-Size Jobs:

```
medium: type=class             # class for medium jobs
priority=70                    # ClassSysprio
cpu_limit=00:10:00             # 10 minute run time limit
data_limit=80mb,60mb           # max 80 MB data segment
                               # soft limit 60 MB data segment
core_limit=30mb                # max 30 MB core file
file_limit=80mb                # max file size 80 MB
stack_limit=30mb               # max stack size 30 MB
rss_limit=100mb                # max resident set size 100 MB
job_cpu_limit=1800,1200        # hard limit is 30 minutes,
                               # soft limit is 20 minutes
```

Example 4: Creating a Class for Large-Size Jobs:

```
large:  type=class             # class for large jobs
priority=60                    # ClassSysprio
cpu_limit=00:10:00             # 10 minute run time limit
```

```
data_limit=120mb               # max 120 MB data segment
default_resources=ConsumableVirtualMemory(40mb)         # resources consumed by each
ConsumableCpus(2) resA(8) floatinglicenseX(1) resB(1)   # task of a large job step if
                               # resources are not explicitly
                               # specified in the job command file
core_limit=30mb                # max 30 MB core file
file_limit=120mb               # max file size 120 MB
stack_limit=unlimited          # unlimited stack size
rss_limit=150mb                # max resident set size 150 MB
job_cpu_limit = 3600,2700      # hard limit 60 minutes
                               # soft limit 45 minutes
wall_clock_limit=12:00:00,11:59:55 # hard limit is 12 hours
```

Example 5: Creating a Class to Route Jobs to NQS Machines:

```
nqs:   type=class               # class for NQS jobs
NQS_class=true
NQS_submit=pipe_queue           # NQS pipe queue name
NQS_query=one two three         # list of queue names
```

You can use the class names in control expressions in both the global and local configuration file.

Example 6: Creating a Class for PVM Jobs:

```
PVM3:  type=class             # class for PVM jobs
priority=60                   # ClassSysprio (max=100)
max_processors=15             # maximum number of processors
```

Example 7: Creating a Class for Master Node Machines:

```
sp-6hr-sp:  type=class               # class for master node machines
priority=50              # ClassSysprio (max=100)
cpu_limit = 06:00:00     # 6 hour limit
job_cpu_limit = 06:00:00 # hard limit is 6 hours
core_limit = lmb         # max 1MB core file
master_node_requirement = true # master node definition
```

Step 4: Specify Group Stanzas

LoadLeveler groups are another way of granting control to the system administrator. Although a LoadLeveler group is independent from a UNIX group, you can configure a LoadLeveler group to have the same users as a UNIX group by using the **include_users** keyword, which is explained in this section.

The information specified in a group stanza defines the characteristics of that group. Group stanzas are optional and take the following format:
You can specify the following keywords in a group stanza:

label: **type = group**
admin = *list*
exclude_users = *list*
include_users = *list*
maxidle = *number*
maxjobs = *number*
maxqueued = *number*
max_node = *number*
max_processors = *number*
priority = *number*
total_tasks = *number*

Figure 28. Format of a Group Stanza

admin = *list*
where *list* is a blank-delimited list of administrators for this group. These administrators can hold, release, and cancel jobs submitted by users in the group.

exclude_users =*list*
where *list* is a blank-delimited list of users that do not belong to the group. Do not specify both a list of included users and a list of excluded users. Only one of these may be used for any group. The default is that no users will be excluded.

include_users =*list*
where *list* is a blank-delimited list of users that belong to the group. If provided, this list limits users of that group to those on the list. Do not specify both a list of included users and a list of excluded users. Only one of these can be used for any group. The default is that all users are included.

maxidle = *number*
where *number* is the maximum number of idle jobs this group can have in queue. That is, *number* is the maximum number of jobs which the negotiator will consider for dispatch for this group. Jobs above this maximum are placed in the NotQueued state. This prevents groups from flooding the job queue. If the group stanza does not specify **maxidle** or if there is no group stanza at all, the maximum number of jobs that can be simultaneously in queue for the group is defined in the default stanza. The default is -1, which means that no limit is placed on the number of jobs that can be simultaneously idle for the group.

For more information, see “Controlling the Mix of Idle and Running Jobs” on page 314 .

maxjobs = *number*
where *number* is a maximum number of jobs this group can run at any time. If the group stanza does not specify the **maxjobs** or if there is no group stanza at all, the maximum number of jobs that can be simultaneously run the group is defined in the default stanza. The default is -1, which means that no limit is placed on the number of jobs that can be simultaneously run for the group. Regardless of the limit set to running jobs, there is no limit to the number of jobs that a group can submit.

For more information, see “Controlling the Mix of Idle and Running Jobs” on page 314.

maxqueued = *number*
where *number* is the maximum number of jobs allowed in the queue for this group. This prevents groups from flooding the job queue. Jobs above this maximum are placed in the NotQueued state. If no **maxqueued** is specified in the group stanza, or if there is no group stanza, the maximum number of jobs that can simultaneously be in the queue is defined in the default stanza. The default is -1, which means that no limit is placed on the number of jobs that can simultaneously be in the job queue for that group. Regardless of the limit set to the number of jobs queued, there is no limit to the number of jobs a group can submit.

For more information, see “Controlling the Mix of Idle and Running Jobs” on page 314.

max_node = *number*
where *number* specifies the maximum number of nodes a user can request for a parallel job in a job command file using the **node** keyword. The default is -1,

which means there is no limit. The **max_node** keyword will not affect the use of the **min_processors** and **max_processors** keywords in the job command file.

max_processors = *number*
where *number* specifies the maximum number of processors a user can request for a parallel job in a job command file using the **min_processors** and **max_processors** keywords. The default is -1, which means there is no limit.

priority = *number*
where *number* is an integer that specifies the job priority for jobs associated with this group. The higher priority numbers result in a better job dispatch order. If the group stanza does not specify a priority or if there is no priority at all, the priority is defined in the default group stanza. The default priority is 0. The number specified for priority is referenced as **GroupSysprio** in the configuration file. **GroupSysprio** can be used in the assignment of job priorities. If the variable **GroupSysprio** does not appear in the SYSPRIO expression in the configuration file, the priority numbers for group specified in the administration file have no effect. See "Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105 for more information about the **GroupSysprio** keyword.

total_tasks = *number*
where *number* specifies the maximum number of tasks a user specifying this group can request for a parallel job in a job command file using the **total_tasks** keyword. The default is -1, which means there is no limit.

Examples of Group Stanzas

Example 1: In this example, the group name is **department_a**. The jobs issued by users belonging to this group will have a priority of 80. There are three members in this group.

```
# Define group stanzas
department_a:  type = group
priority = 80
include_users = susann holly fran
```

Example 2: In this example, the group called **great_lakes** has five members and these user's jobs have a priority of 100:

```
# Define group stanzas
great_lakes:  type = group
priority = 100
include_users = huron ontario michigan erie superior
```

Step 5: Specify Adapter Stanzas

An adapter stanza identifies network adapters that are available on the machines in the LoadLeveler cluster. Adapter stanzas are optional, but you need to specify them when you want LoadLeveler jobs to be able to request a specific adapter. You do not need to specify an adapter stanza when you want LoadLeveler jobs to access a shared, default adapter via TCP/IP.

Note the following when using an adapter stanza:

- An adapter stanza is required for each adapter stanza name you specify on the **adapter_stanzas** keyword of the machine stanza.
- The **adapter_name**, **interface_address**, and **interface_name** keywords are required. For an SP switch adapter, the **switch_node_number** keyword is also required.

For information on creating adapter stanzas for an SP system, see “llextSDR - Extract adapter information from the SDR” on page 182.

An adapter stanza has the following format:
You can specify the following keywords in an adapter stanza:

```
label: type = adapter
adapter_name = name
css_type = type
interface_address = IP_address
interface_name = name
network_type = type
switch_node_number = integer
```

Figure 29. Format of an Adapter Stanza

adapter_name = *string*
Where *string* is the name used to refer to a particular interface card installed on the node. Some examples are en0, tk1, and css0. This keyword defines the adapters a user can specify in a job command file using the **network** keyword. This keyword is required.

css_type = *type*
Where *type* is the designation for the type of switch adapter to be used. The allowable choices are: SP_Switch_Adapter, SP_Switch_MX_Adapter, SP_Switch_MX2_Adapter, RS/6000_SP_System_Attachment_Adapter, and SP_Switch2_Adapter. This keyword must be specified in combination with a switch adapter (″css . . .″), otherwise it will be ignored. The css_type attribute for the available adapters are defined in the SDR. Execute the command **SDRGetObjects Adapter css_type** to obtain a list of css_types, or use **llextSDR** to obtain all of the adapter information from the SDR.

interface_address = *string*
Where *string* is the IP address by which the adapter is known to other nodes in the network. For example: 7.14.21.28. This keyword is required.

interface_name = *string*
Where *string* is the name by which the adapter is known by other nodes in the network. This keyword is required.

network_type = *string*
Where *string* specifies the type of network that the adapter supports (for example, Ethernet). This is an administrator defined name. This keyword defines the types of networks a user can specify in a job command file using the **network** keyword.

switch_node_number = *integer*
Where *integer* specifies the node on which the SP switch adapter is installed. This keyword is required for SP switch adapters. Its value is defined in the switch_node_number field in the Node class in the SDR. This value must match the value in the **/spdata/sys1/st/switch_node_number** file of the Parallel System Support Programs (PSSP).

Example of an Adapter Stanza

Example 1: Specifying an SP Switch Adapter: In the following example, the adapter stanza called “sp01sw.ibm.com” specifies an SP switch adapter. Note that sp01sw.ibm.com is also specified on the **adapter_stanzas** keyword of the machine stanza for the “yugo” machine.

```
          yugo:  type=machine
                 adapter_stanzas = sp01sw.ibm.com
                 ...

sp01sw.ibm.com:  type = adapter
                 adapter_name = css0
                 interface_address = 12.148.44.218
                 interface_name = sp01sw.ibm.com
                 network_type = switch
                 switch_node_number = 7
                 css_type = SP_Switch_MX2_Adapter
```

Configuring LoadLeveler

One of your main tasks as system administrator is to configure LoadLeveler. To configure LoadLeveler, you need to know what the configuration information is and where it is located. Configuration information includes the following:

- The LoadLeveler user ID and group ID
- The configuration directory
- The global configuration file

LoadLeveler sets up the following default values for the configuration information:

- **loadl** is the LoadLeveler user ID and the LoadLeveler group ID. LoadLeveler daemons run under this user ID in order to perform file I/O, and many LoadLeveler files are owned by this user ID.
- The home directory of **loadl** is the configuration directory.
- **LoadL_config** is the name of the configuration file.

You can run your installation with these default values, or you can change any or all of them. To override the defaults, you must update the following keywords in the **/etc/LoadL.cfg** file:

LoadLUserid
: Specifies the LoadLeveler user ID.

LoadLGroupid
: Specifies the LoadLeveler group ID.

LoadLConfig
: Specifies the full path name of the configuration file.

Note that if you change the LoadLeveler user ID to something other than **loadl**, you will have to make sure your configuration files are owned by this ID.

You can also override the **/etc/LoadL.cfg** file. For an example of when you might want to do this, see “Querying Multiple LoadLeveler Clusters” on page 27.

The Configuration Files

By taking a look at the configuration files that come with LoadLeveler, you will find that there are many parameters that you can set. In most cases, you will only have to modify a few of these parameters. In some cases, though, depending upon the LoadLeveler nodes, network connection, and hardware availability, you may need to modify additional parameters. This chapter describes these configuration files and the parameters you can set.

Configuring LoadLeveler involves modifying the configuration files that specify the terms under which LoadLeveler can use machines. There are two types of configuration files:

- *Global Configuration File:* This file by default is called the **LoadL_config** file and it contains configuration information common to all nodes in the LoadLeveler cluster.
- *Local Configuration File:* This file is generally called **LoadL_config.local** (although it is possible for you to rename it). This file contains specific configuration information for an individual node. The **LoadL_config.local** file is in the same format as **LoadL_config** and the information in this file overrides any information specified in **LoadL_config**. It is an optional file that you use to modify information on a local machine. Its full pathname is specified in the **LoadL_config** file by using the **LOCAL_CONFIG** keyword. See “Step 11: Specify Where Files and Directories are Located” on page 112 for more information. “Customizing the Global and Local Configuration Files” on page 99 describes how to tailor this file to suit your needs.

Configuration File Structure and Syntax

The information in both the **LoadL_config** and the **LoadL_config.local** files is in the form of a statement. These statements are made up of *keywords* and *values*. There are three types of configuration file keywords:

- Keywords, described in “Customizing the Global and Local Configuration Files” on page 99 and in “Step 17: Specify Additional Configuration File Keywords” on page 129
- User-defined variables, described in “User-Defined Variables” on page 132
- LoadLeveler variables, described in “LoadLeveler Variables” on page 132

Configuration file statements take one of the following formats:

```
keyword=value
keyword:value
```

Statements in the form *keyword=value* are used primarily to customize an environment. Statements in the form *keyword:value* are used by LoadLeveler to characterize the machine and are known as part of the machine description. Every machine in LoadLeveler has its own machine description which is read by the central manager when LoadLeveler is started.

To continue configuration file statements, use the back-slash character (\).

In the configuration file, comments must be on a separate line from keyword statements.

You can use the following types of constants and operators in the configuration file.

Numerical and Alphabetical Constants

Constants may be represented as:

- Boolean expressions
- Signed integers
- Floating point values
- Strings enclosed in double quotes (″ ″).

Mathematical Operators

You can use the following C operators. The operators are listed in order of precedence. All of these operators are evaluated from left to right:

```
!
* /
- +
< <= > >=
== !=
&&
```

||

Customizing the Global and Local Configuration Files

This section presents a step-by-step approach to configuring LoadLeveler. You do not have to perform the steps in the order that they appear here. Other keywords which are not specifically mentioned in any of these steps are discussed in “Step 17: Specify Additional Configuration File Keywords” on page 129.

Step 1: Define LoadLeveler Administrators

Specify the following keyword:

LOADL_ADMIN = *list of user names* **(required)**

where *list of user names* is a blank-delimited list of those individuals who will have administrative authority. These users are able to invoke the administrator-only commands such as **llctl**, **llfavorjob**, and **llfavoruser**. These administrators can also invoke the administrator-only GUI functions. For more information, see “Administrative Uses for the Graphical User Interface” on page 244.

LoadLeveler administrators on this list also receive mail describing problems that are encountered by the master daemon. When DCE is enabled, the LOADL_ADMIN list is used only as a mailing list. For more information, see “Step 16: Configuring LoadLeveler to use DCE Security Services” on page 123.

An administrator on a machine is granted administrative privileges on that machine. It does not grant him administrative privileges on other machines. To be an administrator on all machines in the LoadLeveler cluster either specify your user ID in the global configuration file with no entries in the local configuration file or specify your userid in every local configuration file that exists in the LoadLeveler cluster.

For example, to grant administrative authority to users `bob` and `mary`, enter the following in the configuration file:

```
LOADL_ADMIN = bob mary
```

Step 2: Define LoadLeveler Cluster Characteristics

You can use the following keywords to define the characteristics of the LoadLeveler cluster:

CUSTOM_METRIC = *number*

Specifies a machine’s relative priority to run jobs. This is an an arbitrary number which you can use in the MACHPRIO expression. If you specify neither **CUSTOM_METRIC** nor **CUSTOM_METRIC_COMMAND**, **CUSTOM_METRIC = 1** is assumed. For more information, see “Step 7: Prioritize the Order of Executing Machines Maintained by the Negotiator” on page 106.

CUSTOM_METRIC_COMMAND = *command*

Specifies an executable and any required arguments. The exit code of this command is assigned to **CUSTOM_METRIC**. If this command does not exit normally, **CUSTOM_METRIC** is assigned a value of 1. This command is forked every (**POLLING_FREQUENCY** * **POLLS_PER_UPDATE**) period.

MACHINE_AUTHENTICATE = true**|false**

Specifies whether machine validation is performed. When set to **true**, LoadLeveler only accepts connections from machines specified in the administration file. When set to **false**, LoadLeveler accepts connections from any machine.

When set to **true**, every communication between LoadLeveler processes will verify that the sending process is running on a machine which is identified via a machine stanza in the administration file. The validation is done by capturing the address of the sending machine when the **accept** function call is issued to accept a connection. The **gethostbyaddr** function is called to translate the address to a name, and the name is matched with the list derived from the administration file.

Choosing a Scheduler: This section discusses the types of schedulers that are available under LoadLeveler, and the keywords you use to define these schedulers.

- *The default LoadLeveler scheduler.* This scheduler runs both serial and parallel jobs, but is primarily meant for serial jobs. It efficiently uses CPU time by scheduling jobs on what otherwise would be idle nodes (and workstations). It does not require that users set a wall clock limit. Also, this scheduler starts, suspends, and resumes jobs based on workload. The default scheduler uses a reservation method to schedule parallel jobs. A possible drawback to the reservation method occurs when LoadLeveler tries to schedule a job requiring a large number of nodes. As LoadLeveler reserves nodes for the job, the reserved nodes will be idle for a period of time. Also, if the job cannot accumulate all the nodes it needs to run, the job may not get dispatched.

  See “Keyword Considerations for Parallel Jobs” on page 59 for information on which keywords associated with parallel jobs are supported by the default scheduler.

- *The Backfill scheduler.* This scheduler runs both serial and parallel jobs, but is primarily meant for parallel jobs. Backfilling is the capability to schedule a job that is short in duration, or which requires a small number of nodes, before a higher priority job. Any idle resources available between the current time and the earliest projected start time of the highest priority job can be used to run other waiting jobs. Jobs will only be backfilled if they will not delay the start of the higher priority job. The scheduler makes this determination by comparing the projected start time of the highest priority job with the **wall_clock_limit** of the potential backfilled job. If the backfilled job will end before the higher priority job’s start time, then it is eligible to run.

  For example: on a rack with 10 nodes, 8 of the nodes are being used by Job A. Job B has the highest priority in the queue, and requires 10 nodes. Job C has the next highest priority in the queue, and requires only two nodes. Job B has to wait for Job A to finish so that it can use the freed nodes. Because Job A is only using 8 of the 10 nodes, the Backfill scheduler can schedule Job C (which only needs the two available nodes) to run as long as it finishes before Job A finishes (and Job B starts). To determine whether or not Job C has time to run, the Backfill scheduler uses Job C’s **wall_clock_limit** value to determine whether or not it will finish before Job A ends. If Job C has a **wall_clock_limit** of **unlimited**, it may not finish before Job B’s start time, and it won’t be dispatched.

  The Backfill scheduler supports:

  – The scheduling of multiple tasks per node.
  – The scheduling of multiple user space tasks per adapter.

  The above functions are not supported by the default LoadLeveler scheduler.

  Note the following when using the Backfill scheduler:

  – To use this scheduler, either users must set a wall clock limit in their job command file or the administrator must define a wall clock limit value for the class to which a job is assigned. Jobs with the **wall_clock_limit** of **unlimited** cannot be used to backfill because they may not finish in time.

- You should use only the default settings for the **START** expression and the other job control functions described in “Step 8: Manage a Job’s Status Using Control Expressions” on page 109. If you do not use these default settings, jobs will still run but the scheduler will not be as efficient. For example, the scheduler will not be able to guarantee a time at which the highest priority job will run.
- You should configure any multiprocessor (SMP) nodes such that the number of jobs that can run on a node (determined by the **MAX_STARTERS** keyword) is always less than or equal to the number of processors on the node.
- Due to the characteristics of the Backfill algorithm, in some cases this scheduler may not honor the **MACHPRIO** statement. For more information on **MACHPRIO**, see “Step 7: Prioritize the Order of Executing Machines Maintained by the Negotiator” on page 106.

See “Keyword Considerations for Parallel Jobs” on page 59 for information on which keywords associated with parallel jobs are supported by the Backfill scheduler.

- *The Workload Management API.* This API allows you to enable an external scheduler, such as the Extensible Argonne Scheduling sYstem (EASY). The API is intended for installations that want to create a scheduling algorithm for parallel jobs based on site-specific requirements. This API provides a time-based (rather than an event-based) interface. That is, your application must use the API to poll LoadLeveler at specific times for machine and job information. Also, some LoadLeveler functions are not available when you use this API. For more information, see “Workload Management API” on page 283.

Use the following keywords to define your scheduler:

SCHEDULER_API = YES**|NO**
where **YES** disables the default LoadLeveler scheduling algorithm. Specifying **YES** implies you will use the job control API to communicate to LoadLeveler scheduling decisions made by an external scheduler. For more information, see “Workload Management API” on page 283. Note that if you change the scheduler from **SCHEDULER=BACKFILL** to **SCHEDULER_API=YES**, you must stop and restart LoadLeveler using **llctl**.

Specify **NO** to run the default LoadLeveler scheduler.

SCHEDULER_TYPE = BACKFILL
where **BACKFILL** specifies the LoadLeveler Backfill scheduler. Note that when you specify this keyword:

- You override the **SCHEDULER_API** keyword (if it is used).
- You should use only the default settings for the **START** expression and the other job control expressions described in “Step 8: Manage a Job’s Status Using Control Expressions” on page 109.

Step 3: Define LoadLeveler Machine Characteristics

You can use the following keywords to define the characteristics of machines in the LoadLeveler cluster:

ARCH = *string* **(required)**
Indicates the standard architecture of the system. The architecture you specify here must be specified in the same format in the **requirements** and **preferences** statements in job command files. The administrator defines the character string for each architecture.

For example, to define a machine as a RISC System/6000, the keyword would look like:

```
ARCH = R6000
```

CLASS = { "*class1***" "***class2***" ... } | { "No_Class" }**

where "*class1*" "*class2*" *...* is a blank delimited list of class names. This keyword determines whether a machine will accept jobs of a certain job class. For parallel jobs, you must define a class for each task you want to run on a node.

You can specify a **default_class** in the default user stanza of the administration file to set a default class. If you don't, jobs will be assigned the class called **No_Class**.

In order for a LoadLeveler job to run on a machine, the machine must have a vacancy for the class of that job. If the machine is configured for only one **No_Class** job and a LoadLeveler job is already running there, then no further LoadLeveler jobs are started on that machine until the current job completes.

You can have a maximum of 1024 characters in the class statement. You cannot use **allclasses** as a class name, since this is a reserved LoadLeveler keyword.

You can assign multiple classes to the same machine by specifying the classes in the LoadLeveler configuration file (called **LoadL_config**) or in the local configuration file (called **LoadL_config.local**). The classes, themselves, should be defined in the administration file. See "Setting Up a Single Machine To Have Multiple Job Classes" on page 315 and "Step 3: Specify Class Stanzas" on page 84 for more information on classes.

Defining Classes – Examples:

Example 1: This example defines the default class:

```
Class = { "No_Class" }
```

This is the default. The machine will only run one LoadLeveler job at a time that has either defaulted to, or explicitly requested class **No_Class**. A LoadLeveler job with class **CPU_bound**, for example, would not be eligible to run here. Only one LoadLeveler job at a time will run on the machine.

Example 2: This example specifies multiple classes:

```
Class = { "No_Class" "No_Class" }
```

The machine will only run jobs that have either defaulted to or explicitly requested class **No_Class**. A maximum of two LoadLeveler jobs are permitted to run simultaneously on the machine if the **MAX_STARTERS** keyword is not specified. See "Step 5: Specify How Many Jobs a Machine Can Run" on page 104 for more information on **MAX_STARTERS**.

Example 3: This example specifies multiple classes:

```
Class = { "No_Class" "Small" "Medium" "Large" }
```

The machine will only run a maximum of four LoadLeveler jobs that have either defaulted to, or explicitly requested **No_Class**, **Small**, **Medium**, or **Large** class. A LoadLeveler job with class **IO_bound**, for example, would not be eligible to run here.

Example 4: This example specifies multiple classes:

```
Class = { "B" "B" "D" }
```

The machine will run only LoadLeveler jobs that have explicitly requested class **B** or **D**. Up to three LoadLeveler jobs may run simultaneously: two of class **B** and one of class **D**. A LoadLeveler job with class **No_Class**, for example, would not be eligible to run here.

Feature = *{"string" ...}*
where *string* is the (optional) characteristic to use to match jobs with machines.

You can specify unique characteristics for any machine using this keyword. When evaluating job submissions, LoadLeveler compares any required features specified in the job command file to those specified using this keyword. You can have a maximum of 1024 characters in the feature statement.

For example, if a machine has licenses for installed products ABC and XYZ, in the local configuration file you can enter the following:

```
Feature = {"abc" "xyz"}
```

When submitting a job that requires both of these products, you should enter the following in your job command file:

```
requirements = (Feature == "abc") && (Feature == "xyz")
```

START_DAEMONS = true|false
Specifies whether to start the LoadLeveler daemons on the node. When **true**, the daemons are started.

In most cases, you will probably want to set this keyword to **true**. An example of why this keyword would be set to **false** is if you want to run the daemons on most of the machines in the cluster but some individual users with their own local configuration files do not want their machines to run the daemons. The individual users would modify their local configuration files and set this keyword to **false**. Because the global configuration file has the keyword set to **true**, their individual machines would still be able to participate in the LoadLeveler cluster.

Also, to define the machine as strictly a submit-only machine, set this keyword to **false**. For more information, see “the submit-only keyword” on page 80.

SCHEDD_RUNS_HERE = true|false
Specifies whether the schedd daemon runs on the host. If you do not want to run the schedd daemon, specify **false**.

To define the machine as an executing machine only, set this keyword to **false**. For more information, see “the submit-only keyword” on page 80.

SCHEDD_SUBMIT_AFFINITY = true|false
Specifies that the **llsubmit** command submits a job to the machine where the command was invoked, provided that the schedd daemon is running on that machine (this is called schedd affinity). Installations with a large number of nodes should consider setting this keyword to **false**. For more information, see “Scaling Considerations” on page 312.

STARTD_RUNS_HERE = true|false
Specifies whether the startd daemon runs on the host. If you do not want to run the startd daemon, specify **false**.

X_RUNS_HERE = true|false
Set **X_RUNS_HERE** to **true** if you want to start the keyboard daemon.

Step 4: Define Consumable Resources

The LoadLeveler scheduler can schedule jobs based on the availability of consumable resources. You can use the following keywords to use Consumable Resources:

SCHEDULE_BY_RESOURCES = *name name ... name*
specifies which consumable resources are considered by the LoadLeveler schedulers. Each consumable resource name may be an administrator-defined alphanumeric string, or may be one of the following predefined resources: **ConsumableCpus**, **ConsumableMemory**, or **ConsumableVirtualMemory**. Each string may only appear in the list once. These resources are either floating resources, or machine resources. If any resource is specified incorrectly with the **SCHEDULE_BY_RESOURCES** keyword, then all scheduling resources will be ignored.

FLOATING_RESOURCES = *name(count) name(count) ... name(count)*
specifies which consumable resources are available collectively on all of the machines in the LoadLeveler cluster. The count for each resource must be an integer greater than or equal to zero, and each resource can only be specified once in the list. Any resource specified for this keyword that is not already listed in the **SCHEDULE_BY_RESOURCES** keyword will not affect job scheduling. If any resource is specified incorrectly with the **FLOATING_RESOURCES** keyword, then all floating resources will be ignored. **ConsumableCpus**, **ConsumableMemory**, and **ConsumableVirtualMemory** may not be specified as floating resources.

Step 5: Specify How Many Jobs a Machine Can Run

To specify how many jobs a machine can run, you need to take into consideration both the **MAX_STARTERS** keyword, which is described in this section, and the **Class** statement, which is mentioned here and described in more detail in “Step 3: Define LoadLeveler Machine Characteristics” on page 101

The syntax for **MAX_STARTERS** is:

MAX_STARTERS = *number*
Where *number* specifies the maximum number of tasks that can run simultaneously on a machine. In this case, a task can be a serial job step, a parallel task, or an instance of the PVM daemon (PVMD). If not specified, the default is the number of elements in the **Class** statement. **MAX_STARTERS** defines the number of initiators on the machine (the number of tasks that can be initiated from a **startd**).

For example, if the configuration file contains these statements:

```
Class = { "A" "B" "B" "C"}
MAX_STARTERS = 2
```

the machine can run a maximum of two LoadLeveler jobs simultaneously. The possible combinations of LoadLeveler jobs are:

- A and B
- A and C
- B and B
- B and C
- Only A, or only B, or only C

If this keyword is specified in conjunction with a **Class** statement, the maximum number of jobs that can be run is equal to the lower of the two numbers. For example, if:

```
MAX_STARTERS = 2
Class = { "class_a" }
```

then the maximum number of job steps that can be run is one (the **Class** statement above defines one class).

If you specify **MAX_STARTERS** keyword without specifying a **Class** statement, by default one class still exists (called **No_Class**). Therefore, the maximum number of jobs that can be run when you do not specify a **Class** statement is one.

If this keyword is not defined in either the global configuration file or the local configuration file, the maximum number of jobs that the machine can run is equal to the number of classes in the **Class** statement.

Step 6: Prioritize the Queue Maintained by the Negotiator

Each job submitted to LoadLeveler is assigned a system priority number, based on the evaluation of the **SYSPRIO** keyword expression in the configuration file of the central manager. The LoadLeveler system priority number is assigned when the central manager adds the new job to the queue of jobs eligible for dispatch. Once assigned, the system priority number for a job is never changed (unless jobs for a user swap their **SYSPRIO**, or **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** is not zero). Jobs assigned higher **SYSPRIO** numbers are considered for dispatch before jobs with lower numbers. See “How Does a Job’s Priority Affect Dispatching Order?” on page 28 for more information on job priorities.

You can use the following LoadLeveler variables to define the **SYSPRIO** expression:

ClassSysprio
: The priority for the class of the job step, defined in the class stanza in the administration file. The default is 0.

GroupQueuedJobs
: The number of job steps associated with a LoadLeveler group which are either running or queued. (That is, job steps which are in one of these states: Running, Starting, Pending, or Idle.)

GroupRunningJobs
: The number of job steps for the LoadLeveler group which are in one of these states: Running, Starting, or Pending.

GroupSysprio
: The priority for the group of the job step, defined in the group stanza in the administration file. The default is 0.

GroupTotalJobs
: The total number of job steps associated with this LoadLeveler group. Total job steps are all job steps reported by the **llq** command.

QDate The difference in the UNIX date when the job step enters the queue and the UNIX date when the negotiator starts up.

UserPrio
: The user-defined priority of the job step, specified in the job command file with the **user_priority** keyword. The default is 50.

UserQueuedJobs
The number of job steps either running or queued for the user. (That is, job steps which are in one of these states: Running, Starting, Pending, or Idle.)

UserRunningJobs
The number of job step steps for the user which are in one of these states: Running, Starting, or Pending.

UserSysprio
The priority of the user who submitted the job step, defined in the user stanza in the administration file. The default is 0.

UserTotalJobs
The total number of job steps associated with this user. Total job steps are all job steps reported by the **llq** command.

Usage Notes for the SYSPRIO Keyword:

- The **SYSPRIO** keyword is valid only on the machine where the central manager is running. Using this keyword in a local configuration file has no effect.
- It is recommended that you do not use **UserPrio** in the **SYSPRIO** expression, since user jobs are already ordered by **UserPrio**.
- You can use the **UserRunningJobs**, **GroupRunningJobs**, **UserQueuedJobs**, **GroupQueuedJobs**, **UserQueuedJobs**, **GroupQueuedJobs UserTotalJobs**, and **GroupTotalJobs** parameters to prioritize the queue based on current usage. You should also set **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** so that the priorities are adjusted according to current usage rather than usage only at submission time.

Using the SYSPRIO Keyword – Examples:

Example 1: This example creates a FIFO job queue based on submission time:

```
SYSPRIO : 0 - (QDate)
```

Example 2: This example accounts for Class, User, and Group system priorities:

```
SYSPRIO : (ClassSysprio * 100) + (UserSysprio * 10) + (GroupSysprio * 1) - (QDate)
```

Example 3: This example orders the queue based on the number of jobs a user is currently running. The user who has the fewest jobs running is first in the queue. You should set **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** in conjunction with this **SYSPRIO** expression.

```
SYSPRIO : 0 - UserRunningJobs
```

Step 7: Prioritize the Order of Executing Machines Maintained by the Negotiator

Each executing machine is assigned a machine priority number, based on the evaluation of the **MACHPRIO** keyword expression in the configuration file of the central manager. The LoadLeveler machine priority number is updated every time the central manager updates its machine data. Machines assigned higher **MACHPRIO** numbers are considered to run jobs before machines with lower numbers. For example, a machine with a **MACHPRIO** of 10 is considered to run a job before a machine with a **MACHPRIO** of 5. Similarly, a machine with a **MACHPRIO** of -2 would be considered to run a job before a machine with a **MACHPRIO** of -3.

Note that the **MACHPRIO** keyword is valid only on the machine where the central manager is running. Using this keyword in a local configuration file has no effect.

When you use a **MACHPRIO** expression that is based on load average, the machine may be temporarily ordered later in the list immediately after a job is scheduled to that machine. This is because the negotiator adds a compensating factor to the startd machine's load average every time the negotiator assigns a job. For more information, see "the NEGOTIATOR_INTERVAL keyword" on page 130.

You can use the following LoadLeveler variables in the **MACHPRIO** expression:

LoadAvg
The Berkeley one-minute load average of the machine, reported by startd.

Cpus The number of processors of the machine, reported by startd.

Speed The relative speed of the machine, defined in a machine stanza in the administration file. The default is 1.

Memory
The size of real memory in megabytes of the machine, reported by startd.

VirtualMemory
The size of available swap space in kilobytes of the machine, reported by startd.

Disk The size of free disk space in kilobytes on the filesystem where the executables reside.

CustomMetric
Allows you to set a relative priority number for one or more machines, based on the value of the **CUSTOM_METRIC** keyword. (See "Example 4" for more information.)

MasterMachPriority
A value that is equal to 1 for nodes which are master nodes (those with **master_node_exclusive = true**); this value is equal to 0 for nodes which are not master nodes. Assigning a high priority to master nodes may help job scheduling performance for parallel jobs which require master node features.

ConsumableCpus
If **ConsumableCpus** is specified in the **SCHEDULE_BY_RESOURCES** keyword, then this is the number of **ConsumableCpus** available on the machine. If **ConsumableCpus** is not specified in the **SCHEDULE_BY_RESOURCES** keyword, then this is the same as **Cpus**.

ConsumableMemory
This is the number of megabytes of **ConsumableMemory** available on the machine, provided that **ConsumableMemory** is specified in the **SCHEDULE_BY_RESOURCES** keyword. If **ConsumableMemory** is not specified in the **SCHEDULE_BY_RESOURCES** keyword, then this is the same as **Memory**.

ConsumableVirtualMemory
This is the number of megabytes of **ConsumableVirtualMemory** available on the machine, provided that **ConsumableVirtualMemory** is specified in the **SCHEDULE_BY_RESOURCES** keyword. If ConsumableVirtualMemory is not specified in the SCHEDULE_BY_RESOURCES keyword, then this is the same as **VirtualMemory**.

PagesFreed
The number of pages freed per second by the page replacement algorithm of the virtual memory manager.

PagesScanned
The number of pages scanned per second by the page replacement algorithm of the virtual memory manager.

FreeRealMemory
The amount of free real mrmory in megabytes on the machine.

Using the MACHPRIO Keyword – Examples:

Example 1: This example orders machines by the Berkeley one-minute load average.

```
MACHPRIO : 0 - (LoadAvg)
```

Therefore, if **LoadAvg** equals .7, this example would read:

```
MACHPRIO : 0 - (.7)
```

The **MACHPRIO** would evaluate to -.7.

Example 2: This example orders machines by the Berkeley one-minute load average normalized for machine speed:

```
MACHPRIO : 0 - (1000 * (LoadAvg / (Cpus * Speed)))
```

Therefore, if **LoadAvg** equals .7, **Cpus** equals 1, and **Speed** equals 2, this example would read:

```
MACHPRIO : 0 - (1000 * (.7 / (1 * 2)))
```

This example further evaluates to:

```
MACHPRIO : 0 - (350)
```

The **MACHPRIO** would evaluate to -350.

Notice that if the speed of the machine were increased to 3, the equation would read:

```
MACHPRIO : 0 - (1000 * (.7 / (1 * 3)))
```

The **MACHPRIO** would evaluate to approximately -233. Therefore, as the speed of the machine increases, the **MACHPRIO** also increases.

Example 3: This example orders machines accounting for real memory and available swap space (remembering that Memory is in Mbytes and VirtualMemory is in Kbytes):

```
MACHPRIO : 0 - (10000 * (LoadAvg / (Cpus * Speed))) +
(10 * Memory) + (VirtualMemory / 1000)
```

Example 4: This example sets a relative machine priority based on the value of the **CUSTOM_METRIC** keyword.

```
MACHPRIO : CustomMetric
```

To do this, you must specify a value for the **CUSTOM_METRIC** keyword or the **CUSTOM_METRIC_COMMAND** keyword in either the **LoadL_config.local** file of a machine or in the global **LoadL_config** file. To assign the same relative priority to all machines, specify the **CUSTOM_METRIC** keyword in the global configuration file. For example:

```
CUSTOM_METRIC = 5
```

You can override this value for an individual machine by specifying a different value in that machine's **LoadL_config.local** file.

Example 5: This example gives master nodes the highest priority:

```
MACHPRIO : (MasterMachPriority * 10000)
```

Step 8: Manage a Job's Status Using Control Expressions

You can control running jobs by using five control functions as Boolean expressions in the configuration file. These functions are useful primarily for serial jobs. You define the expressions, using normal C conventions, with the following functions:

START
SUSPEND
CONTINUE
VACATE
KILL

The expressions are evaluated for each job running on a machine using both the job and machine attributes. Some jobs running on a machine may be suspended while others are allowed to continue.

The START expression is evaluated twice; once to see if the machine can accept jobs to run and second to see if the specific job can be run on the machine. The other expressions are evaluated after the jobs have been dispatched and in some cases, already running.

When evaluating the START expression to determine if the machine can accept jobs, **Class != { "Z" }** evaluates to true only if `Z` is not in the class definition. This means that if two different classes are defined on a machine, **Class != { "Z" }** (where `Z` is one of the defined classes) always evaluates to false when specified in the START expression and, therefore, the machine will not be considered to start jobs.

START: *expression that evaluates to T or F (true or false)*
Determines whether a machine can run a LoadLeveler job. When the expression evaluates to **T**, LoadLeveler considers dispatching a job to the machine.

When you use a START expression that is based on the CPU load average, the negotiator may evaluate the expression as **F** even though the load average indicates the machine is Idle. This is because the negotiator adds a compensating factor to the startd machine's load average every time the negotiator assigns a job. For more information, see "the NEGOTIATOR_INTERVAL keyword" on page 130.

SUSPEND: *expression that evaluates to T or F (true or false)*
Determines whether running jobs should be suspended. When **T**, LoadLeveler temporarily suspends jobs currently running on the machine. Suspended LoadLeveler jobs will either be continued or vacated. This keyword is not supported for parallel jobs.

CONTINUE: *expression that evaluates to T or F (true or false)*
Determines whether suspended jobs should continue execution. When **T**, suspended LoadLeveler jobs resume execution on the machine.

VACATE: *expression that evaluates to T or F (true or false)*
Determines whether suspended jobs should be vacated. When **T**, suspended LoadLeveler jobs are removed from the machine and placed back into the

queue (provided you specify **restart=yes** in the job command file). If a checkpoint was taken, the job restarts from the checkpoint. Otherwise, the job restarts from the beginning.

KILL: *expression that evaluates to T or F (true or false)*
Determines whether or not vacated jobs should be killed and replaced in the queue. It is used to remove a job that is taking too long to vacate. When **T**, vacated LoadLeveler jobs are removed from the machine with no attempt to take checkpoints.

Typically, machine load average, keyboard activity, time intervals, and job class are used within these various expressions to dynamically control job execution.

How Control Expressions Affect Jobs: After LoadLeveler selects a job for execution, the job can be in any of several states. Figure 30 shows how the control expressions can affect the state a job is in. The rectangles represent job or daemon states, and the diamonds represent the control expressions.

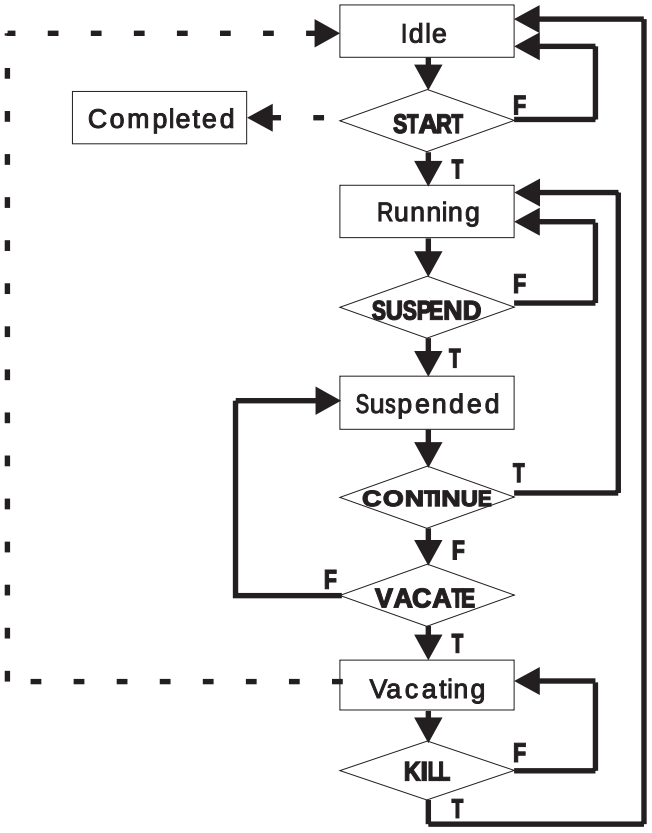

Figure 30. How Control Expressions Affect Jobs

Criteria used to determine when a LoadLeveler job will enter Start, Suspend, Continue, Vacate, and Kill states are defined in the LoadLeveler configuration files and may be different for each machine in the cluster. They may be modified to meet local requirements.

Step 9: Define Job Accounting

LoadLeveler provides accounting information on completed LoadLeveler jobs. For detailed information on this function, refer to “Chapter 7. Gathering Job Accounting Data” on page 153.

The following keywords allow you to control accounting functions:

ACCT = *flag*
The available flags are:

A_ON Turns accounting data recording on. If specified without the **A_DETAIL** flag, the following is recorded:
- The total amount of CPU time consumed by the entire job
- The maximum memory consumption of all tasks (or nodes).

A_OFF
Turns accounting data recording off. This is the default.

A_VALIDATE
Turns account validation on.

A_DETAIL
Enables extended accounting. Using this flag causes LoadLeveler to record detail resource consumption by machine and by events for each job step. This flag also enables the **-x** flag of the **llq** command, permitting users to view resource consumption for active jobs.

For example:

```
ACCT = A_ON A_DETAIL
```

This example specifies that accounting should be turned on and that extended accounting data should be collected and that the -x flag of the **llq** command be enabled.

ACCT_VALIDATION = *$(BIN/llacctval* **(optional)**
Keyword used to identify the executable that is called to perform account validation. You can replace the **llacctval** executable with your own validation program by specifying your program in this keyword.

GLOBAL_HISTORY = *$(SPOOL)* **(optional)**
Keyword used to identify the directory that will contain the global history files produced by **llacctmrg** command when no directory is specified as a command argument.

For example, the following section of the configuration file specifies that the accounting function is turned on. It also identifies the module used to perform account validation and the directory containing the global history files:

```
ACCT               = A_ON A_VALIDATE
ACCT_VALIDATION    = $(BIN)/llacctval
GLOBAL_HISTORY     = $(SPOOL)
```

Step 10: Specify Alternate Central Managers

In one of your machine stanzas specified in the administration file, you specified that the machine would serve as the central manager. It is possible for some problem to cause this central manager to become unusable such as network communication or software or hardware failures. In such cases, the other machines in the LoadLeveler cluster believe that the central manager machine is no longer operating. To remedy this situation, you can assign one or more alternate central managers in the machine stanza to take control.

The following machine stanza example defines the machine `deep_blue` as an alternate central manager:

```
#
deep_blue:  type=machine
central_manager = alt
```

If the primary central manager fails, the alternate central manager then becomes the central manager. The alternate central manager is chosen based upon the order in which its respective machine stanza appears in the administration file.

When an alternate becomes the central manager, jobs will not be lost, but it may take a few minutes for all of the machines in the cluster to check in with the new central manager. As a result, job status queries may be incorrect for a short time.

When you define alternate central managers, you should set the following keywords in the configuration file:

CENTRAL_MANAGER_HEARTBEAT_INTERVAL = *number*
where *number* is the amount of time in seconds that defines how frequently primary and alternate central managers communicate with each other. The default is 300 seconds or 5 minutes.

CENTRAL_MANAGER_TIMEOUT = *number*
where *number* is the number of heartbeat intervals that an alternate central manager will wait without hearing from the primary central manager before declaring that the primary central manager is not operating. The default is 6.

In the following example, the alternate central manager will wait for 30 intervals, where each interval is 45 seconds:

```
# Set a 45 second interval
CENTRAL_MANAGER_HEARTBEAT_INTERVAL = 45
# Set the number of intervals to wait
CENTRAL_MANAGER_TIMEOUT = 30
```

For more information on central manager backup, refer to “What Happens if the Central Manager Isn’t Operating?” on page 309.

Step 11: Specify Where Files and Directories are Located

The configuration file provided with LoadLeveler specifies default locations for all of the files and directories. You can modify their locations using the following keywords. Keep in mind that the LoadLeveler installation process installs files in these directories and these files may be periodically cleaned up. Therefore, you should not keep any files that do not belong to LoadLeveler in these directories.

To specify the location of the:	Specify these keywords:
Administration File	**ADMIN_FILE =** *pathname* **(required)** points to the administration file containing user, class, group, machine, and adapter stanzas. For example, `ADMIN_FILE = $(tilde)/admin_file`

To specify the location of the:	Specify these keywords:

Local Configuration File	**LOCAL_CONFIG =** *pathname*
	defines the pathname of the optional local configuration file containing information specific to a node in the LoadLeveler network. If you are using a distributed file system like NFS, some examples are:
	`LOCAL_CONFIG = $(tilde)/$(host).LoadL_config.local`
	`LOCAL_CONFIG = $(tilde)/LoadL_config.$(host).$(domain)`
	`LOCAL_CONFIG = $(tilde)/LoadL_config.local.$(hostname)`
	If you are using a local file system, an example is:
	`LOCAL_CONFIG = /var/LoadL/LoadL_config.local`
	See “LoadLeveler Variables” on page 132 for information about the **tilde**, **host**, and **domain** variables.

Local Directory	The following subdirectories reside in the local directory. It is possible that the local directory and LoadLeveler’s home directory are the same. **EXECUTE =** *local directory***/execute (required)** defines the local directory to store the executables of jobs submitted by other machines. **LOG =** *local directory***/log (required)** defines the local directory to store log files. It is not necessary to keep all the log files created by the various LoadLeveler daemons and programs in one directory but you will probably find it convenient. **SPOOL =** *local directory***/spool (required)** Defines the local directory where LoadLeveler keeps the local job queue and checkpoint files, as well as: **HISTORY = $(SPOOL)/history (required)** defines the pathname where a file containing the history of local LoadLeveler jobs is kept.
Release Directory	**RELEASEDIR =** *release directory* **(required)** defines the directory where all the LoadLeveler software resides. The following subdirectories are created during installation and they reside in the release directory. You can change their locations. **BIN = $(RELEASEDIR)/bin (required)** defines the directory where LoadLeveler binaries are kept. **LIB = $(RELEASEDIR)/lib (required)** defines the directory where LoadLeveler libraries are kept. **NQS_DIR =** *NQS directory (optional)* defines the directory where NQS commands **qsub**, **qstat**, and **qdel** reside. The default is **/usr/bin**.

Step 12: Record and Control Log Files

The LoadLeveler daemons and processes keep log files according to the specifications in the configuration file. A number of keywords are used to describe where LoadLeveler maintains the logs and how much information is recorded in each log. These keywords, shown in Table 13 on page 114, are repeated in similar form to specify the pathname of the log file, its maximum length, and the debug flags to be used.

"Controlling Debugging Output" describes the events that can be reported through logging controls.

Table 13. Log Control Statements

Daemon/ Process	Log File *(required)* (See note 1)	Max Length *(required)* (See note 2)	Debug Control *(required)* (See note 4)
Master	**MASTER_LOG =** *path*	**MAX_MASTER_LOG =** *bytes*	**MASTER_DEBUG =** *flags*
Schedd	**SCHEDD_LOG =** *path*	**MAX_SCHEDD_LOG =** *bytes*	**SCHEDD_DEBUG =** *flags*
Startd	**STARTD_LOG =** *path*	**MAX_STARTD_LOG =** *bytes*	**STARTD_DEBUG =** *flags*
Starter	**STARTER_LOG** = *path*	**MAX_STARTER_LOG =** *bytes*	**STARTER_DEBUG =** *flags*
Negotiator	**NEGOTIATOR_LOG =** *path*	**MAX_NEGOTIATOR_LOG =** *bytes*	**NEGOTIATOR_DEBUG =** *flags*
Kbdd	**KBDD_LOG =** *path*	**MAX_KBDD_LOG =** *bytes*	**KBDD_DEBUG =** *flags*
GSmonitor	**GSMONITOR_LOG = path**	**MAX_GSMONITOR_LOG = bytes**	**GSMONITOR_DEBUG = flags**

Notes:

1. When coding the *path* for the log files, it is not necessary that all LoadLeveler daemons keep their log files in the same directory, however, you will probably find it a convenient arrangement.
2. There is a maximum length, in bytes, beyond which the various log files cannot grow. Each file is allowed to grow to the specified length and is then saved to an **.old** file. The **.old** files are overwritten each time the log is saved, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file. The default length is 64KB. To obtain records over a longer period of time, that don't get overwritten, you can use the SAVELOGS keyword in the local or global configuration files. See"Saving Log Files" on page 116 for more information on extended capturing of LoadLeveler logs.

   You can also specify that the log file be started anew with every invocation of the daemon by setting the **TRUNC** statement to **true** as follows:

   TRUNC_MASTER_LOG_ON_OPEN = true|**false**
   TRUNC_STARTD_LOG_ON_OPEN = true|**false**
   TRUNC_SCHEDD_LOG_ON_OPEN = true|**false**
   TRUNC_KBDD_LOG_ON_OPEN = true|**false**
   TRUNC_STARTER_LOG_ON_OPEN = true|**false**
   TRUNC_NEGOTIATOR_LOG_ON_OPEN = true|**false**
   TRUNC_GSMONITOR_LOG_ON_OPEN = true|**false**
3. LoadLeveler creates temporary log files used by the **starter** daemon. These files are used for synchronization purposes. When a job starts, a **StarterLog.***pid* file is created. When the job ends, this file is appended to the **StarterLog** file.
4. Normally, only those who are installing or debugging LoadLeveler will need to use the debug flags, described in "Controlling Debugging Output" The default error logging, obtained by leaving the right side of the debug control statement null, will be sufficient for most installations.

Controlling Debugging Output: You can control the level of debugging output logged by LoadLeveler programs. The following flags are presented here for your information, though they are used primarily by IBM personnel for debugging purposes:

D_ACCOUNT
Logs accounting information about processes. If used, it may slow down the network.

D_AFS
Logs information related to AFS credentials.

D_DAEMON
Logs information regarding basic daemon set up and operation, including information on the communication between daemons.

D_DBX
Bypasses certain signal settings to permit debugging of the processes as they execute in certain critical regions.

D_DCE
Logs information related to DCE credentials.

D_EXPR
Logs steps in parsing and evaluating control expressions.

D_FULLDEBUG
Logs details about most actions performed by each daemon but doesn't log as much activity as setting all the flags.

D_JOB
Logs job requirements and preferences when making decisions regarding whether a particular job should run on a particular machine.

D_KERNEL
Activates diagnostics for errors involving the process tracking kernel extension.

D_LOAD
Displays the load average on the startd machine.

D_LOCKING
Logs requests to acquire and release locks.

D_MACHINE
Logs machine control functions and variables when making decisions regarding starting, suspending, resuming, and aborting remote jobs.

D_NEGOTIATE
Displays the process of looking for a job to run in the negotiator. It only pertains to this daemon.

D_NQS
Provides more information regarding the processing of NQS files.

D_PROC
Logs information about jobs being started remotely such as the number of bytes fetched and stored for each job.

D_QUEUE
Logs changes to the job queue.

D_STANZAS
Displays internal information about the parsing of the administration file.

D_SCHEDD
Displays how the schedd works internally.

D_STARTD
Displays how the startd works internally.

D_STARTER
Displays how the starter works internally.

D_THREAD
Displays the ID of the thread producing the log message. The thread ID is displayed immediately following the date and time. This flag is useful for debugging threaded daemons.

D_XDR
Logs information regarding External Data Representation (XDR) communication protocols.

For example,

```
SCHEDD_DEBUG = D_CKPT  D_XDR
```

causes the scheduler to log information about checkpointing user jobs and exchange xdr messages with other LoadLeveler daemons. These flags will primarily be of interest to LoadLeveler implementers and debuggers.

Saving Log Files: By default, LoadLeveler stores only the two most recent iterations of a daemon's log file (<*daemon name*>_Log, and <*daemon name*>_Log.old). Occasionally, for problem diagnosing, users will need to capture LoadLeveler logs over an extended period. Users can specify that all log files be saved to a particular directory by using the **SAVELOGS** keyword in a local or global configuration file. Be aware that LoadLeveler does not provide any way to manage and clean out all of those log files, so users must be sure to specify a directory in a file system with enough space to accomodate them. This file system should be separate from the one used for the LoadLeveler log, spool, and execute directories. The syntax is:

SAVELOGS = <*directory*>

where <*directory*> is the directory in which log files will be archived.

Each log file is represented by the name of the daemon that generated it, the exact time the file was generated, and the name of the machine on which the daemon is running. When you list the contents of the **SAVELOGS** directory, the list of log file names looks like this:

```
NegotiatorLogNov02.16:10:39c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:42c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:46c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:48c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:51c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:53c163n10.ppd.pok.ibm.com
StarterLogNov02.16:09:19c163n10.ppd.pok.ibm.com
StarterLogNov02.16:09:51c163n10.ppd.pok.ibm.com
StarterLogNov02.16:10:30c163n10.ppd.pok.ibm.com
SchedLogNov02.16:09:05c163n10.ppd.pok.ibm.com
SchedLogNov02.16:09:26c163n10.ppd.pok.ibm.com
SchedLogNov02.16:09:47c163n10.ppd.pok.ibm.com
SchedLogNov02.16:10:12c163n10.ppd.pok.ibm.com
SchedLogNov02.16:10:37c163n10.ppd.pok.ibm.com
StartLogNov02.16:09:05c163n10.ppd.pok.ibm.com
StartLogNov02.16:09:26c163n10.ppd.pok.ibm.com
StartLogNov02.16:09:47c163n10.ppd.pok.ibm.com
StartLogNov02.16:10:12c163n10.ppd.pok.ibm.com
StartLogNov02.16:10:37c163n10.ppd.pok.ibm.com
```

Step 13: Define Network Characteristics

A **port number** is an integer that specifies the port number to use to connect to the specified daemon. You can define these port numbers in the configuration file or the **/etc/services** file or you can accept the defaults. LoadLeveler first looks in the configuration file for these port numbers. If the port number is in the configuration file and is valid, this value is used. If it is an invalid value, the default value is used.

If LoadLeveler does not find the value in the configuration file, it looks in the **/etc/services** file. If the value is not found in this file, the default is used.

The configuration file keywords associated with port numbers are the following:

CLIENT_TIMEOUT = *number*
where *number* specifies the maximum time, in seconds, that a LoadLeveler

daemon waits for a response over TCP/IP from a process. If the waiting time exceeds the specified amount, the daemon tries again to communicate with the process. The default is 30 seconds. In general, you should use this default setting unless you are experiencing delays due to an excessively loaded network. If so, you should try increasing this value. **CLIENT_TIMEOUT** is used by all LoadLeveler daemons.

CM_COLLECTOR_PORT = *port number*
The default is 9612.

MASTER_STREAM_PORT = *port number*
The default is 9616.

NEGOTIATOR_STREAM_PORT = *port number*
The default is 9614.

SCHEDD_STATUS_PORT = *port number*
The default is 9606.

SCHEDD_STREAM_PORT = *port number*
The default is 9605.

STARTD_STREAM_PORT = *port number*
The default is 9611.

STARTD_DGRAM_PORT = *port number*
The default is 9615.

MASTER_DGRAM_PORT = *port number*
The default is 9617.

As stated earlier, if LoadLeveler does not find the value in the configuration file, it looks in the **/etc/services** file. If the value is not found in this file, the default is used. The first field on each line in the example that follows represents the name of a ″service″. In most cases, these services are also the names of daemons because few daemons need more than one udp and one tcp connection. There are two exceptions: LoadL_negotiator_collector is the service name for a second stream port that is used by the LoadL_negotiator daemon; LoadL_schedd_status is the service name for a second stream port used by the LoadL_schedd daemon.

```
LoadL_master               9616/tcp   # Master port number for stream port
LoadL_negotiator           9614/tcp   # Negotiator port number
LoadL_negotiator_collector 9612/tcp   # Second negotiator stream port
LoadL_schedd               9605/tcp   # Schedd port number for stream port
LoadL_schedd_status        9606/tcp   # Schedd stream port for job status data
LoadL_startd               9611/tcp   # Startd port number for stream port
LoadL_master               9617/udp   # Master port number for dgram port
LoadL_startd               9615/udp   # Startd port number for dgram port
```

Step 14: Enable Checkpointing

This section tells you how to set up checkpointing for jobs. For more information on the job command file keywords mentioned here, see “Job Command File Keywords” on page 36. To enable checkpointing for parallel jobs, you must use the APIs provided with the Parallel Environment (PE) program. For information on parallel checkpointing, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

Checkpointing is a method of periodically saving the state of a job so that if the job does not complete it can be restarted from the saved state. You can checkpoint both serial and parallel jobs.

You can specify the following types of checkpointing:

user initiated
The user's application program determines when the checkpoint is taken. This type of checkpointing is available to both serial and parallel jobs.

system initiated
The checkpoint is taken at administrator-defined intervals. This type of checkpointing is available only to serial jobs.

At checkpoint time, a checkpoint file is created, by default, on the executing machine and stored on the scheduling machine. You can control where the file is created and stored by using the CHKPT_FILE and CHKPT_DIR environment variables, which are described in "Set the Appropriate Environment Variables". The checkpoint file contains the program's data segment, stack, heap, register contents, signal state and the states of the open files at the time of the checkpoint. The checkpoint file is often much larger in size than the executable.

When a job is vacated, the most recent checkpoint file taken before the job was vacated is used to restart the job when it is scheduled to run on a new machine. Note that a vacating job may be killed by LoadLeveler if the job takes too long to write its checkpoint file. This occurs only when a job is vacated by the executing machine after the job's VACATE expression evaluates to TRUE. See "Step 8: Manage a Job's Status Using Control Expressions" on page 109 for more information on the VACATE and KILL expressions.

If the executing machine fails, then when the machine restarts LoadLeveler reschedules the job, which restores its state from the most recent checkpoint file. LoadLeveler waits for the original executing machine to restart before scheduling the job to run on another machine in order to ensure that only one copy of the job will run.

Planning Considerations for Checkpointing Jobs

Review the following guidelines before you submit a checkpointing job:

Set the Appropriate Environment Variables: This section discusses the CHKPT_STATE, CHKPT_FILE, and CHKPT_DIR environment variables.

The CHKPT_STATE environment variable allows you to enable and disable checkpointing. CHKPT_STATE can be set to the following:

enable
Enables checkpointing.

restart
Restarts the executable from an existing checkpoint file.

If you set **checkpoint=no** in your job command file, no checkpoints are taken, regardless of the value of the CHKPT_STATE environment variable. See "checkpoint" on page 37 for more information.

The CHKPT_FILE and CHKPT_DIR environment variables help you manage your checkpoint files. For parallel jobs, you must specify at least one of these variables in order to designate the location of the checkpoint file. For serial jobs, if you do not specify either of these variables, LoadLeveler manages your checkpoint files. LoadLeveler stores the checkpoint file in its working directories and deletes the file as soon as the job terminates (that is, when the job exits the LoadLeveler system.) If your job terminates abnormally, there is no checkpoint file from which LoadLeveler can restart the job. When you resubmit the job, it will start running from the beginning.

To avoid this problem, use CHKPT_FILE and CHKPT_DIR to control where your checkpoint file is stored. CHKPT_DIR specifies the directory where it is stored, and CHKPT_FILE specifies the checkpoint file name. (You can use just CHKPT_FILE provided you specify a full path name. Also, you can use just CHKPT_DIR; in this case the checkpoint file is copied to the directory you specify with a file name of *executable*.**chkpt**.) You can use these variables to have your checkpoint file written to a the file system of your choice. This allows you to resubmit your job and have it restart from the last checkpoint file, since the file will not be erased if your job is terminated. If your job completes normally, the checkpoint library deletes all checkpoint files associated with the job.

Note that two or more job steps running at the same time cannot both write to the same checkpoint file, since the file will be corrupted.

See "How to Checkpoint a Job" on page 121 for more information.

Plan for Jobs that You Will Migrate: If you plan to migrate jobs (restart jobs on a different node or set of nodes), you should understand the difference between writing checkpoint files to a local file system (such as JFS) versus a global file system (such as AFS or GPFS). The CHKPT_DIR and CHKPT_FILE environment variables allow you to write to either type of file system. If you are using a local file system, you must first move the checkpoint file(s) to the target node(s) before resubmitting the job. Then you must ensure that the job runs on those specific nodes. If you are using a global file system, the checkpointing may take longer, but there is no additional work required to migrate the job.

Reserve Adequate Disk Space in the Execute Directory: A checkpoint file requires a significant amount of disk space. Your job may fail if the directory where the checkpoint file is written does not have adequate space. For serial jobs, the directory must be able to contain two checkpoint files. For parallel jobs, the directory must be able to contain **2*n** checkpoint files, where **n** is the number of tasks. You can make an accurate size estimate only after you've run your job and noticed the size of the checkpoint file that is created. LoadLeveler attempts to reserve enough disk space for the checkpoint file when the job is started. However, only you can ensure that enough space is available.

Set your Checkpoint File Size to the Maximum: To make sure that your job is not prevented from writing a checkpoint file due to system limits, assign your job to a job class that has its file creation limit set to the maximum (unlimited). In the administration file, set up a class stanza for checkpointing jobs with the following entry:

```
file_limit = unlimited,unlimited
```

This statement specifies that there is no limit on the maximum size of a file that your program can create.

Checkpoint Programs Whose States are Simple to Checkpoint and Recreate: For some processes, it is impossible to obtain or recreate the state of the process. For this reason, you should only checkpoint programs whose states are simple to checkpoint and recreate. A program that is long-running, computation-intensive, and does not fork any processes is an example of a job well suited for checkpointing.

Avoid Using Certain System Services in Checkpointed Jobs: In order to prevent unpredictable results from occurring, checkpointing jobs should not use the following system services:

- Threads

- Shared libraries
- Dynamic loading
- Shared memory (such as pfork and shmget)
- IPC (sockets, pipes, semaphores, and message queues)
- Memory-mapped files
- Fork and exec system calls
- Device I/O
- File locks
- Set/get user or group IDs and process IDs
- Open system calls from inside a signal handler
- Time and timer services
- Administrative calls (for example, DCE security, audit, and swapqry)
- 64 bit addressing

Another limitation of checkpointing jobs is file I/O. Since individual write calls are not traced, the file recovery scheme requires that all I/O operations, when repeated, must yield the same result. A job that opens all files as read only can be checkpointed. A job that writes to a file and then reads the data back may also be checkpointed. An example of I/O that could cause unpredictable results is reading, writing, and then reading again the same area of a file.

Ensure Jobs are Restarted on an Appropriate Machine: A checkpointed serial job must be restarted on a machine with the same processor and the same operating system level, including service fixes, as the machine on which the checkpoint was taken.

A checkpointed parallel job must be restarted on a machine with the same processor, the same operating system level, including service fixes, and the same SP switch adapter(s) as the machine on which the checkpoint was taken.

Choose a Supported Compiler: Compile your program with one of the following supported compilers:

- For FORTRAN: xlf 5.1.1 or later releases
- For C and C++: xlC 3.6.x, or Visual Age C, C++ (VAC++) 4.1

Ensure all User's Jobs are Linked to Checkpointing Libraries: All serial checkpointing programs must be linked with the LoadLeveler libraries **libchkrst.a** and **chkrst_wrap.o**. To ensure your checkpointing jobs are linked correctly, compile your programs using the compile scripts found in the **bin** subdirectory of the LoadLeveler release directory. These compile scripts are as follows:

crxlc (for use with C)
crxlC (for use with C++)
crxlf (for use with FORTRAN)

In all these scripts, be sure to substitute all occurrences of "RELEASEDIR" with the location of the LoadLeveler release directory.

C Syntax

crxlc *executable* [*args*] *source_file*

Where:

executable
Is your checkpointable binary.

args Is one or more arguments you supply to the compiler (**xlc -c**).

source_file
Is your C source code.

Some examples are:

```
crxlc myprog myprog.c
crxlc myprog -qlanglvl=extended myprog.c
```

C++ Syntax

crxlC *executable* [*args*] *source_file*

Where:

executable
Is your checkpointable binary.

args Is one or more arguments you supply to the compiler (**xlC -c**).

source_file
Is your C++ source code.

Some examples are:

```
crxlC myprog myprog.C
crxlC myprog -qlanglvl=extended myprog.C
```

FORTRAN Syntax

crxlf *executable* [*args*] *source_file*

Where:

executable
Is your checkpointable binary.

args Is one or more arguments you supply to the compiler (**xlf -c**).

source_file
Is your FORTRAN source code.

Some examples are:

```
crxlf myprog myprog.f
crxlf myprog -qintlog -qfullpath myprog.f
```

How to Checkpoint a Job

There are several ways to checkpoint a job. To determine which type of checkpointing is appropriate for your situation, refer to the following table:

To specify that:	Do this:
Your serial job determines when the checkpoint occurs	Add the following option to your job command file: **checkpoint = user_initiated** You can also select this option on the Build a Job window of the GUI. User initiated checkpointing is available to FORTRAN, C, and C++ programs which call the **ckpt** serial checkpointing API. See “Serial Checkpointing API” on page 253 for more information.

To specify that:	Do this:
LoadLeveler automatically checkpoints your serial job.	Add the following option to your job command file: **checkpoint = system_initiated** You can also select this option on the Build a Job window of the GUI. For this type of checkpointing to work, system administrators must set two keywords in the configuration file to specify how often LoadLeveler would take a checkpoint of the job. These two keywords are: **MIN_CKPT_INTERVAL =** *number* **MAX_CKPT_INTERVAL =** *number* where *number* specifies a period, in seconds, between checkpoints taken for running jobs. The time between checkpoints will be increased after each checkpoint within these limits as follows: • The first checkpoint is taken after a period of time equal to the **MIN_CKPT_INTERVAL** has passed. • The second checkpoint is taken after LoadLeveler waits *twice as long* (**MIN_CKPT_INTERVAL** X 2) • The third checkpoint is taken after LoadLeveler waits twice as long again (**MIN_CKPT_INTERVAL** X 4) before taking the third checkpoint. LoadLeveler continues to double this period until the value of **MAX_CKPT_INTERVAL** has been reached, where it stays for the remainder of the job. A minimum value of 900 (15 minutes) and a maximum value of 7200 (2 hours) are the defaults. You can set these keyword values globally in the global configuration file so that all machines in the cluster have the same value, or you can specify a different value for each machine by modifying the local configuration files. To enable both user initiated and system initiated checkpointing for a job, specify **checkpoint=system_initiated** in your job command file, and code the **ckpt** API call in your program. System initiated checkpointing is not available to parallel jobs.
LoadLeveler restarts your executable from an existing checkpoint file when you submit the job.	Pass the CHKPT_STATE environment variable using the LoadLeveler **environment** keyword in your job command file. For more information, see “environment” on page 41. You must also set the CHKPT_DIR and/or CHKPT_FILE environment variables.
Your job not be checkpointed	Add the following option to your job command file: **checkpoint = no** You can also select this option on the Build a Job window of the GUI. This option is the default.

Step 15: Specify Process Tracking

When a job terminates, it’s orphaned processes may continue to consume or hold resources, thereby degrading system performance, or causing jobs to hang or fail. Process tracking allows LoadLeveler to cancel any processes (throughout the entire cluster), left behind when a job terminates. Using process tracking is optional. There are two keywords used in specifying process tracking:

PROCESS_TRACKING

To activate process tracking, set **PROCESS_TRACKING=TRUE** in the LoadLeveler global configuration file. By default, **PROCESS_TRACKING** is set to **FALSE**.

PROCESS_TRACKING_EXTENSION
This keyword is used to specify the path to the kernel extension binary **LoadL_pt_ke** in the local or global configuration file. If the **PROCESS_TRACKING_EXTENSION** keyword is not supplied, then LoadLeveler will search the default directory **$HOME/bin**.

Step 16: Configuring LoadLeveler to use DCE Security Services

When LoadLeveler is configured to exploit DCE security, it uses PSSP and DCE security services to:

- Authenticate the identity of users and programs interacting with LoadLeveler.
- Authorize users and programs to use LoadLeveler services. It will prevent unauthorized users and programs from misusing resources or disrupting services.
- Delegate the user credentials at submit time to the Starter process to give the user's job the same DCE permissions at run time.

You can skip this section if you do not plan to use these security features or if you plan to continue to use only the limited support for DCE available in LoadLeveler 2.1. Please consult "Usage Notes" on page 128 for additional information.

When LoadLeveler is configured to exploit DCE security, most of its interactions with DCE are through the PSSP security services API. For this reason, it is important that you configure PSSP security services before you configure LoadLeveler for DCE. For more information on PSSP security services, please refer to: *RS/6000 SP Planning Volume 2, Control Workstation and Software Environment* (GA22-7281-05), *Parallel System Support Programs for AIX Installation and Migration Guide Version 3 Release 2* (GA22-7347-02), and *Parallel System Support Programs for AIX Administration Guide Version 3 Release 2* (SA22-7348-02).

DCE maintains a registry of all DCE principals which have been authorized to login to the DCE cell. In order for LoadLeveler daemons to login to DCE, DCE accounts must be set up, and DCE key files must be created for these daemons. In LoadLeveler 2.2 each LoadLeveler daemon on each node is associated with a different DCE principal. The DCE principal of the Schedd daemon running on node A is distinct from the DCE principal of the Schedd daemon running on node B. Since it is possible for up to seven LoadLeveler daemons to run on any particular node (Master, Negotiator, Schedd, Startd, Kbdd, Starter, and GSmonitor), the number of DCE principal accounts and key files that must be created could reach as high as 7x(number of nodes). Since it is not always possible to know in advance on which node a particular daemon will run, a conservative approach would be to create accounts and key files for all seven daemons on all nodes in a given LoadLeveler cluster. However, it is only necessary to create accounts and keyfiles for DCE principals which will actually be instantiated and run in the cluster.

These are the steps used for configuring LoadLeveler for DCE. We recommend that you use SMIT and the lldcegrpmaint command to perform this task. The manual steps are also described in "Manual Configuration" on page 125, and may be useful should you need to create a highly customized LoadLeveler environment. Some of the names used in this section are the default names as defined in the file /usr/lpp/ssp/config/spsec_defaults and can be overridden with appropriate specifications in the file /spdata/sys1/spsec/spsec_overrides. Also, the term ″LoadLeveler node″ is used to refer to a node on an SP system that will be part of a LoadLeveler cluster.

Using SMIT and the lldcegrpmaint command:

1. Login to the SP control workstation as **root**, then login to DCE as **cell_admin**.

2. Start the SMIT program. From SMIT's main menu, select the **RS/6000 SP System Management** option, then select the **RS/6000 SP Security** option in the next menu.
3. Perform the appropriate steps associated with this menu to configure the security features of this SP system. From LoadLeveler's perspective, the important actions are:
   - **Create dcehostnames**
   - **Configure SP Trusted Services to use DCE Authentication**

   Before continuing to step 4, ensure that:
   - DCE hostnames for LoadLeveler nodes are defined.
   - A DCE group named **spsec-services** and a DCE organization named **spsec-services** are created.
   - The DCE principals of the LoadLeveler daemons on LoadLeveler nodes are created.
   - The DCE principals of the LoadLeveler daemons on LoadLeveler nodes are added to the **spsec-services** group and the **spsec-services** organization.
   - A DCE account is created for each DCE principal associated with the LoadLeveler daemons on the SP system.
   - A DCE key file is created for each LoadLeveler daemon on the LoadLeveler nodes.
4. If the LoadLeveler cluster consists of nodes spanning several SP systems, then you should repeat step 1 on page 123 through step 3 for each SP system.
5. PSSP security services use certain fields in the SDR (System Data Repository) to determine the current software configuration. Use the command "**splstdata -p**" to verify that the field **ts_auth_methods** is set to either **dce** or **dce:compat**. If **ts_auth_methods** is set to **dce:compat** then either DCE or non-DCE authentication is allowed. For some PSSP applications, this setting also implies that if DCE authentication is activated but, DCE authentication cannot be performed, then non-DCE authentication will be used. However, LoadLeveler can not change authentication methods dynamically, and the **dce:compat** setting simply indicates that LoadLeveler can be brought up in either DCE or non-DCE authentication modes using the **DCE_ENABLEMENT** keyword.
6. Add these statements to the LoadLeveler global configuration file:

   ```
   DCE_ENABLEMENT = TRUE
   DCE_ADMIN_GROUP = LoadL-admin
   DCE_SERVICES_GROUP = LoadL-services
   ```

   DCE_ENABLEMENT must be set to **TRUE** to activate the DCE security features of LoadLeveler version 2.2. The *LoadL-admin* group should be populated with DCE principals of users who are to be given LoadLeveler administrative priviledges. For more information on populating the *LoadL-admin* group, see 9 on page 125. The *LoadL-services* group should be populated with the DCE principals of all the LoadLeveler daemons that will be running in the current cluster. You can use the **lldcegrpmaint** command to automate this process. For more information on populating the *LoadL-services* group, see step 8 on page 125. Note that these daemons are already members of the **spsec-services** group. If there is more than one DCE-enabled LoadLeveler cluster within the same DCE cell, then it is important that the name assigned to **DCE_SERVICES_GROUP** for each cluster be distinct; this will avoid any potential operational conflict.

7. Add DCE hostnames to the machine stanzas of the LoadLeveler administration file. The machine stanza of each node defined in the LoadLeveler administration file must contain a statement with this format:

   dce_host_name = *DCE hostname*

   Execute either ″**SDRGetObjects Node dcehostname,**″ or ″**llextSDR**″ to obtain a listing of DCE hostnames of nodes on an SP system.

8. Execute the command:

   ```
   lldcegrpmaint config_pathname admin_pathname
   ```

   where *config_pathname* is the pathname of the LoadLeveler global configuration file and *admin_pathname* is the pathname of the LoadLeveler administration file. The **lldcegrpmaint** command will:

   - Create the *LoadL-services* and *LoadL-admin* DCE groups (if they do not already exist).
   - Add the DCE principals of all the LoadLeveler daemons in the LoadLeveler cluster defined by the *admin_pathname* file to the *LoadL-services* group.

   For more information about the lldcegrpmaint command, see “lldcegrpmaint - LoadLeveler DCE group Maintenance Utility” on page 180.

9. Add the DCE principals of users who will have LoadLeveler administrative authority for the cluster to the *LoadL-admin* group. For example, this command adds **loadl** to the **LoadL-admin** group:

   ```
   dcecp -c group add LoadL-admin -member loadl
   ```

Manual Configuration: Here is an example of the steps you must take to configure LoadLeveler for DCE.

In this example, the LoadLeveler cluster consists of 3 nodes of an SP system which belong to the same DCE cell. Their hostnames and DCE hostnames are the same: c163n01.pok.ibm.com, c163n02.pok.ibm.com, and c163n03.pok.ibm.com. Assume that the basic PSSP security setup steps have been performed, and that the DCE group **spsec-services** and the DCE organization **spsec-services** have been created.

1. Login to any node in the DCE cell as **root** and login to DCE as **cell_admin**.
2. Create LoadLeveler’s product directory if it does not already exist. First, see if the directory has already been created:

   ```
   dcecp -c cdsli /.:/subsys
   ```

   This command lists the contents of the **/.:/subsys** directory in DCE. LoadLeveler’s product name within DCE is **LoadL**, so its product directory is **/.:/subsys/LoadL**. If this directory already exists, then continue to the next step. If it does not exist, issue to following command to create it:

   ```
   dcecp -c directory create /.:/subsys/LoadL
   ```

3. Create the DCE principal names for all of the LoadLeveler daemons in the LoadLeveler cluster. PSSP security services expect the DCE principal name of a LoadLeveler daemon to have the format:

   product_name/dce_host_name/dce_daemon_name

   where:

   product_name
   is the product name and should always be set to **LoadL**.

dce_host_name
is the DCE hostname of the node on which the daemon will run.

dce_daemon_name
is the DCE name of the daemon and is defined in the file **/usr/lpp/ssp/config/spsec_defaults**. Go to the LoadLeveler section of this file. You will find a **SERVICE** record similar to this for all the seven daemons:

```
SERVICE:LoadL/Master:kw:root:system
```

The relevant portion of this record is **Master**; this is the DCE daemon name of **LoadL_master**. The DCE daemon names of other daemons can be identified in a similar manner.

For the c163n01.pok.ibm.com node, the following commands will create the desired principal names:

```
dcecp -c principal create LoadL/c163n01.pok.ibm.com/Master
dcecp -c principal create LoadL/c163n01.pok.ibm.com/Negotiator
dcecp -c principal create LoadL/c163n01.pok.ibm.com/Schedd
dcecp -c principal create LoadL/c163n01.pok.ibm.com/Kbdd
dcecp -c principal create LoadL/c163n01.pok.ibm.com/Startd
dcecp -c principal create LoadL/c163n01.pok.ibm.com/Starter
dcecp -c principal create LoadL/c163n01.pok.ibm.com/GSmonitor
```

These commands must then be repeated for each node in the LoadLeveler cluster, replacing the *dce_host_name* with the DCE hostname of each respective node.

4. Add the principals defined in step 3 on page 125 to the PSSP security services' services group. This group is named **spsec-services**. PSSP security services require that any daemon using their APIs be members of this group. This command will add the DCE principal of the Master daemon on node c163n01 to the spsec-services group.

   ```
   dcecp -c group add spsec-services -member LoadL/c163n01.pok.ibm.com/Master
   ```

   This operation must be repeated for all of the other LoadLeveler daemons on c163n01, and the complete set of operations must be repeated for all of the nodes in the LoadLeveler cluster.

5. Add the principals defined in step 3 on page 125 to the **spsec-services** organization. The following command will add the DCE principal of the Master daemon on node c163n01 to the **spsec-services** organization.

   ```
   dcecp -c organization add spsec-services -member LoadL/c163n01.pok.ibm.com/Master
   ```

   This operation must be repeated for all of the other LoadLeveler daemons on c163n01, and the complete set of operations must be repeated for all of the nodes in the LoadLeveler cluster.

6. Create a DCE account for each of the principals defined in step 3 on page 125. This series of commands will create a DCE account for the Master daemon on node c163n01:

   ```
   dcecp  <Enter>
   dcecp> account create LoadL/c163n01.pok.ibm.com/Master \
          -group spsec-services -organization spsec-services \
          -password  service-password -mypwd  cell_admin's-password
   dcecp> quit
   ```

   The *service-password* passed to DCE in this command can be any valid DCE password. Please take note of it since you will need it when you create the key file for this daemon in step 8 on page 127. The continuation character "\" is not

supported by **dcecp**, but appears in the example merely for clarity. This operation must be repeated for the other LoadLeveler daemons on c163n01, and the complete set of operations must be repeated for all of the nodes in the LoadLeveler cluster.

7. Create directories to contain the key files for the principals defined in step 3 on page 125.

   ```
   mkdir -p /spdata/sys1/keyfiles/LoadL/dce_host_name
   ```

   You must login to the appropriate node to perform this operation. This operation must be repeated for every node in the LoadLeveler cluster.

   NOTE: The directory **/spdata/sys1/keyfiles** should already exist on each node in the cluster which has been installed with a level of PSSP software that supports DCE Security exploitation. If this directory does not exist, then the node cannot support DCE Security and LoadLeveler 2.2 in DCE mode will not run on it. If this configuration seems to be in error, contact your system administrator to determine which nodes in the cluster should support DCE Security.

8. Create a key file for each LoadLeveler daemon on the node on which it will run. The key file contains security-related information specific to each daemon. Use this series of commands:

   ```
   dcecp  <Enter>
   dcecp> keytab create LoadL/c163n01.pok.ibm.com/Master \
          -storage /spdata/sys1/keyfiles/LoadL/c163n01.pok.ibm.com/Master \
          -data { LoadL/c163n01.pok.ibm.com/Master plain 1 service-password }
   dcecp> quit
   ```

   You must login to node c163n01 to perform this operation. DCE must be able to locate the key file locally, otherwise the daemon's login to DCE on startup will fail. The principal name passed to DCE in the preceeding example is the same principal name defined in step 3 on page 125. The AIX path passed with the ″-storage″ flag should point to the same directory created in step 7. The principal name passed with the ″-data″ flag should match the principal name used at the beginning of the command. The password used in the *service-password* field must be the same as the service password defined when this principal's account was created in step 6 on page 126.

   This operation must be repeated for all of the other LoadLeveler daemons on node c163n01, and the complete set of operations must be repeated for all of the nodes in the LoadLeveler cluster.

9. Perform steps 5 on page 124, 6 on page 124, and 7 on page 125 of "Using SMIT and the lldcegrpmaint command" on page 123.

10. Create the DCE groups *LoadL-admin*, and *LoadL-services*. This command creates the DCE group **LoadL-admin**:

    ```
    dcecp -c group create LoadL-admin
    ```

11. Add the DCE principals of users who will have LoadLeveler administrative authority for the cluster to the *LoadL-admin* group. This command adds **loadl** to the **LoadL-admin** group:

    ```
    dcecp -c group add LoadL-admin -member loadl
    ```

12. Add the principals defined in step 3 on page 125 to the *LoadL-services* group. This command will add the DCE principal of the Master daemon on node c163n01.pok.ibm.com to **LoadL-services**:

    ```
    dcecp -c group add LoadL-services -member LoadL/c163n01.pok.ibm.com/Master
    ```

This operation must be repeated for all of the other LoadLeveler daemons on node c163n01, and the complete set of operations must be repeated for all of the nodes in the LoadLeveler cluster.

Usage Notes:

1. Limited support for DCE security was available in a previous version of LoadLeveler. In version 2.1, the configuration keyword ″**DCE_AUTHENTICATION_PAIR** = *program1*, *program2*″ was used to activate LoadLeveler support for DCE security and to specify to LoadLeveler which programs should be used to authenticate DCE security credentials. *program1* obtains a handle (an opaque credentials object), at the time the job is submitted to LoadLeveler, which is used to authenticate to DCE. *program2* uses the handle obtained by *program1* to authenticate to DCE before starting the job on the executing machine(s). These programs could be the default LoadLeveler binaries **llgetdce** and **llsetdce**, or a pair of installation defined binaries. See pages 129, and 295 for more information on the **DCE_AUTHENTICATION_PAIR** keyword.

   In LoadLeveler 2.2, this limited form of support for DCE is still available. If the **DCE_ENABLEMENT** keyword is not defined, then the **DCE_AUTHENTICATION_PAIR** keyword can still be used to activate this legacy feature. If this level of DCE support meets your requirements, then you can ignore the setup steps in this section. However, setting the **DCE_ENABLEMENT** configuration keyword to **TRUE** activates a more comprehensive level of support for DCE . In this case, LoadLeveler will use the PSSP security services API to perform mutual authentication of all appropriate transactions in addition to using **llgetdce** and **llsetdce** (or the pair of programs specified by **DCE_AUTHENTICATION_PAIR**) to obtain the opaque credentials object and to authenticate to DCE before starting the job. Unless you want to specify a pair of programs other than the default **llgetdce** and **llsetdce** binaries, the use of the **DCE_AUTHENTICATION_PAIR** keyword in the configuration file is optional when ″**DCE_ENABLEMENT = TRUE**″.

2. When **DCE_ENABLEMENT** is set to **TRUE**, LoadLeveler uses a different set of criteria to determine who owns job steps, and who has administrator privileges.
   - LoadLeveler considers you to be the owner of a job step if your DCE principal matches the DCE principal associated with that job step.
   - LoadLeveler administrators are usually defined to LoadLeveler through a list of names associated with the **LOADL_ADMIN** keyword. However, when **DCE_ENABLEMENT** is **TRUE**, this list is no longer used for this purpose. Instead, users and processes whose DCE principals are members of the *LoadL-admin* DCE group are given LoadLeveler administrative privileges.

   Note: The **LOADL_ADMIN** keyword is also used to provide LoadLeveler with a list of users who are to receive mail notification of problems encountered by the **LoadL_master** daemon. This function is not affected by the **DCE_ENABLEMENT** keyword.

3. If **DCE_ENABLEMENT** is set to **TRUE**, you must login to DCE with the **dce_login** command before attempting to execute any LoadLeveler command. Also, if an AIX user's user name is different from the user's DCE principal name, then the AIX user must have a .k5login file in the home directory specifying which DCE principal may execute using the AIX account. For example, if your DCE principal in the cell **local_dce_cell** is **user1_dce**, and your AIX user name is **user1**, then you will have to add an entry such as ″user1_dce@local_dce_cell″ to the .k5login file in your home directory.

Step 17: Specify Additional Configuration File Keywords

This section describes keywords that were not mentioned in the previous configuration steps. Unless your installation has special requirements for any of these keywords, you can use them with their default settings.

Note: For the keywords listed below which have a *number* as the value on the right side of the equal sign, that *number* must be a numerical value and cannot be an arithmetic expression.

ACTION_ON_MAX_REJECT = HOLD | SYSHOLD **|** CANCEL

Specifies the state in which jobs are placed when their rejection count has reached the value of the **MAX_JOB_REJECT** keyword. HOLD specifies that jobs are placed in User Hold status; SYSHOLD specifies that jobs are placed in System Hold status; CANCEL specifies that jobs are canceled. The default is HOLD. When a job is rejected, LoadLeveler sends a mail message stating why the job was rejected.

AFS_GETNEWTOKEN = *myprog*

where *myprog* is an administrator supplied program that, for example, can be used to refresh an AFS token. The default is to not run a program.

For more information, see “Handling an AFS Token” on page 295.

DCE_AUTHENTICATION_PAIR = *program1, program2*

where program1 and program2 are LoadLeveler or installation supplied programs that are used to authenticate DCE security credentials. program1 obtains a handle (an opaque credentials object), at the time the job is submitted, which is used to authenticate to DCE. program2 is the path name of a LoadLeveler or installation supplied program that uses the handle obtained by program1 to authenticate to DCE before starting the job on the executing machine(s).

You must specify this keyword in order to enable DCE authentication. To use LoadLeveler’s default DCE authentication method, specify:

```
DCE_AUTHENTICATION_PAIR = $(BIN)/llgetdce, $(BIN)/llsetdce
```

To use your own DCE authentication method, substitute your own programs into the keyword definition. For more information on DCE security credentials, see “Handling DCE Security Credentials” on page 294.

DRAIN_ON_SWITCH_TABLE_ERROR = true | false

When **DRAIN_ON_SWITCH_TABLE_ERROR** is set to true, the **startd** will be drained when the switch table fails to unload. This will flag the administator that intervention may be required to unload the switch table. The default is **false**.

MACHINE_UPDATE_INTERVAL = *number*

where *number* specifies the time period, in seconds, during which machines must report to the central manager. Machines that do not report in this number of seconds are considered *down*. The default is 300 seconds.

MAX_JOB_REJECT = *number*

where *number* specifies the number of times a job can be rejected before it is removed (cancelled) or put in User Hold or System Hold status. That is, a rejected job is redispatched until the **MAX_JOB_REJECT** value is reached. The default is -1, meaning a job is redispatched an unlimited number of times. A job that cannot run for various reasons (such as a **uid** mismatch, unavailable resources, or wrong permissions) on one machine will be rejected on that machine, and LoadLeveler will attempt to run the job on another machine. A

value of 0 means that if the job is rejected, it is immediately removed. (For related information, see the **NEGOTIATOR_REJECT_DEFER** keyword in this section.)

NEGOTIATOR_INTERVAL = *number*
where *number* specifies the interval, in seconds, at which the negotiator daemon performs a "negotiation loop" during which it attempts to assign available machines to waiting jobs. A negotiation loop also occurs whenever job states or machine states change. The default is 30 seconds.

NEGOTIATOR_CYCLE_DELAY = *number*
where *number* specifies the time, in seconds, the negotiator delays between periods when it attempts to schedule jobs. This time is used by the negotiator daemon to respond to queries, reorder job queues, collect information about changes in the states of jobs, etc. Delaying the scheduling of jobs might improve the overall performance of the negotiator by preventing it from spending excessive time attempting to schedule jobs. The **NEGOTIATOR_CYCLE_DELAY** must be less than the **NEGOTIATOR_INTERVAL**. The default is 0 seconds.

NEGOTIATOR_LOADAVG_INCREMENT = *number*
where *number* specifies the value the negotiator adds to the startd machine's load average whenever a job in the Pending state is queued on that machine. This value is used to compensate for the increased load caused by starting another job. The default value is .5.

NEGOTIATOR_PARALLEL_DEFER = *number*
where *number* specifies the amount of time in seconds that defines how long a job stays out of the queue after it fails to get the correct number of processors. This keyword applies only to the default LoadLeveler scheduler. This keyword must be greater than the **NEGOTIATOR_INTERVAL**. value; if it is not, the default is used. The default, set internally by LoadLeveler, is **NEGOTIATOR_INTERVAL** multiplied by 5.

NEGOTIATOR_PARALLEL_HOLD = *number*
where *number* specifies the amount of time in seconds that defines how long a job is given to accumulate processors. This keyword applies only to the default LoadLeveler scheduler. This keyword must be greater than the **NEGOTIATOR_INTERVAL** value; if it is not, the default is used. The default, set internally by LoadLeveler, is **NEGOTIATOR_INTERVAL** multiplied by 5.

NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL = *number*
where *number* specifies the amount of time in seconds between calculation of the **SYSPRIO** values for waiting jobs. The default is 120 seconds. Recalculating the priority can be CPU-intensive; specifying low values for the **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** keyword may lead to a heavy CPU load on the **negotiator** if a large number of jobs are running or waiting for resources. A value of 0 means the **SYSPRIO** values are not recalculated.

You can use this keyword to base the order in which jobs are run on the current number of running, queued, or total jobs for a user or a group. For more information, see "Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105.

NEGOTIATOR_REJECT_DEFER = *number*
where *number* specifies the amount of time in seconds the negotiator waits before it considers scheduling a job to a machine that recently rejected the job.

The default is 120 seconds. (For related information, see the **MAX_JOB_REJECT** keyword in this section.)

NEGOTIATOR_REMOVE_COMPLETED = *number*
where *number* is the amount of time in seconds that you want the negotiator to keep information regarding completed and removed jobs so that you can query this information using the **llq** command. The default is 0 seconds.

NEGOTIATOR_RESCAN_QUEUE = *number*
where *number* specifies the amount of time in seconds that defines how long the negotiator waits to rescan the job queue for machines which have bypassed jobs which could not run due to conditions which may change over time. This keyword must be greater than the **NEGOTIATOR_INTERVAL** value; if it is not, the default is used. The default is 900 seconds.

OBITUARY_LOG_LENGTH = *number*
where *number* specifies the number of lines from the end of the file that are appended to the mail message. The master daemon mails this log to the LoadLeveler administrators when one of the daemons dies. The default is 25.

POLLING_FREQUENCY = *number*
where *number* specifies the interval, in seconds, with which the startd daemon evaluates the load on the local machine and decides whether to suspend, resume, or abort jobs. This is also the minimum interval at which the kbdd daemon reports keyboard or mouse activity to the startd daemon. A value of 5 is the default.

POLLS_PER_UPDATE = *number*
where *number* specifies how often, in **POLLING_FREQUENCY** intervals, startd daemon updates the central manager. Due to the communication overhead, it is impractical to do this with the frequency defined by the **POLLING_FREQUENCY** keyword. Therefore, the startd daemon only updates the central manager every *n*th (where *n* is the number specified for **POLLS_PER_UPDATE**) local update. Change **POLLS_PER_UPDATE** when changing the **POLLING_FREQUENCY**. The default is 6.

PUBLISH_OBITUARIES = true|false
where **true** specifies that the master daemon sends mail to the administrator(s), identified by **LOADL_ADMIN** keyword, when any of the daemons it manages dies abnormally.

RESTARTS_PER_HOUR = *number*
where *number* specifies how many times the master daemon attempts to restart a daemon that dies abnormally. Because one or more of the daemons may be unable to run due to a permanent error, the master only attempts **$(RESTARTS_PER_HOUR)** restarts within a 60 minute period. Failing that, it sends mail to the administrator(s) identified by the **LOADL_ADMIN** keyword and exits. The default is 12.

SCHEDD_INTERVAL = *number*
where *number* specifies the interval, in seconds, at which the schedd daemon checks the local job queue and updates the negotiator daemon. The default is 60 seconds.

WALLCLOCK_ENFORCE = true|false
Where **true** specifies that the **wall_clock_limit** on the job will be enforced. The **WALLCLOCK_ENFORCE** keyword is only valid when the External Scheduler is enabled.

User-Defined Variables

This type of variable, which is generally created and defined by the user, can be named using any combination of letters and numbers. A user-defined variable is set equal to values, where the *value* defines conditions, names files, or sets numeric values. For example, you can create a variable named **MY_MACHINE** and set it equal to the name of your machine named *iron* as follows:

```
MY_MACHINE = iron.ore.met.com
```

You can then identify the keyword using a dollar sign ($) and parentheses. For example, the literal **$(MY_MACHINE)** following the definition in the previous example results in the automatic substitution of **iron.ore.met.com** in place of **$(MY_MACHINE)**.

User-defined definitions may contain references, enclosed in parentheses, to previously defined keywords. Therefore:

```
A = xxx
C = $(A)
```

is a valid expression and the resulting value of **C** is *xxx*. Note that **C** is actually bound to **A**, not to its value, so that

```
A = xxx
C = $(A)
A = yyy
```

is also legal and the resulting value of **C** is *yyy*.

The sample configuration file shipped with the product defines and uses some “user-defined” variables.

LoadLeveler Variables

The LoadLeveler product includes variables that you can use in the configuration file. LoadLeveler variables are evaluated by the LoadLeveler daemons at various stages. They do not require you to use any special characters (such as a parenthesis or a dollar sign) to identify them.

LoadLeveler provides the following variables that you can use in your configuration file statements.

Arch
: indicates the system architecture. Note that **Arch** is a special case of a LoadLeveler variable called a machine variable. You specify a machine variable using the the following format:

  variable : $*(value)*

ConsumableCpus
: the number of **ConsumableCpus** currently available on the machine, if **ConsumableCpus** is defined in the **SCHEDULE_BY_RESOURCES**. If it is not defined in the **SCHEDULE_BY_RESOURCES**, then it is equivalent to **Cpus**.

ConsumableMemory
: the amount of **ConsumableMemory** currently available on the machine, if **ConsumableMemory** is defined in the **SCHEDULE_BY_RESOURCES**. If it is not defined in the **SCHEDULE_BY_RESOURCES**, then it is equivalent to **Memory**.

ConsumableVirtualMemory
: the amount of **ConsumableVirtualMemory** currently available on the machine, if **ConsumableVirtualMemory** is defined in the

SCHEDULE_BY_RESOURCES. If it is not defined in the **SCHEDULE_BY_RESOURCES**, then it is equivalent to **VirtualMemory**.

Cpus
the number of CPU's installed.

CurrentTime
the **UNIX date**; the current system time, in seconds, since January 1, 1970, as returned by the time() function.

CustomMetric
sets a relative machine priority.

Disk
the free disk space in kilobytes on the file system where the executables for the LoadLeveler jobs assigned to this machine are stored. This refers to the file system that is defined by the execute keyword.

domain or **domainname**
dynamically indicates the official name of the domain of the current host machine where the program is running. Whenever a machine name can be specified or one is assumed, a domain name is assigned if none is present.

EnteredCurrentState
the value of **CurrentTime** when the current state (START, SUSPEND, etc) was entered.

host or **hostname**
dynamically indicates the official name of the host machine where the program is running. **host** returns the machine name without the domain name; **hostname** returns the machine and the domain.

KeyboardIdle
the number of seconds since the keyboard or mouse was last used. It also includes any telnet or interactive activity from any remote machine.

LoadAvg
The Berkely one-minute load average, a measure of the CPU load on the system. The load average is the average of the number of processes ready to run or waiting for disk I/O to complete. The load average does not map to CPU time.

Machine
indicates the name of the current machine. Note that **Machine** is a special case of a LoadLeveler variable called a machine variable. See the description of the **Arch** variable for more information.

Memory
the physical memory installed on the machine in megabytes.

MasterMachPriority
a value that is equal to 1 for nodes which are master nodes, and is equal to 0 otherwise.

OpSys
indicates the operating system on the host where the program is running. This value is automatically determined and need not be defined in the configuration file. Note that **OpSys** is a special case of a LoadLeveler variable called a machine variable. See the description of the **Arch** variable for more information.

QDate
the difference in seconds between when LoadLeveler (specifically the negotiator daemon) comes up and when the job is submitted using llsubmit.

Speed
the relative speed of a machine.

State
the state of the startd daemon.

tilde
the home directory for the LoadLeveler userid.

UserPrio
the user defined priority of the job. The priority ranges from 0 to 100, with higher numbers corresponding to greater priority.

VirtualMemory
the size of available swap space on the machine in kilobytes.

Time: You can use the following time variables in the START, SUSPEND, CONTINUE, VACATE, and KILL expressions. If you use these variables in the START expression and you are operating across multiple time zones, unexpected results may occur. This is because the negotiator daemon evaluates the START expressions and this evaluation is done in the time zone in which the negotiator resides. Your executing machine also evaluates the START expression and if your executing machine is in a different time zone, the results you may receive may be inconsistent. To prevent this inconsistency from occurring, ensure that both your negotiator daemon and your executing machine are in the same time zone.

tm_hour
the number of hours since midnight (0-23).

tm_min
number of minutes after the hour (0-59).

tm_sec
number of seconds after the minute (0-59).

tm_isdst
Daylight Savings Time flag: positive when in effect, zero when not in effect, negative when information is unavailable. For example, to start jobs between 5PM and 8AM during the month of October, factoring in an adjustment for Daylight Savings Time, you can issue:

```
START: (tm_mon == 9) && (tm_hour < 8) && (tm_hour > 17) && (tm_isdst = 1)
```

Date:

tm_mday
the number of the day of the month (1-31).

tm_wday
number of days since Sunday (0-6).

tm_yday
number of days since January 1 (0-365).

tm_mon
number of months since January (0-11).

tm_year
the number of years since 1900 (0-9999). For example:

```
tm_year == 100
```

denotes the year 2000.

tm4_year
The integer representation of the current year. For example:

```
tm4_year == 2010
```

denotes the year 2010.

Keyword Summary

This section contains summaries keywords you can use in the administration file and those you can use in the configuration file.

Administration File Keywords

The following table contains a brief description of the keywords you can use in the administration file. For more information on a specific keyword, see the section and page number referenced in the “For Details” column.

Admin. File Keyword	Stanza(s)	Brief Description	For Details
account	User, Group	A list of account numbers available to a user submitting jobs.	“Step 2: Specify User Stanzas” on page 81
adapter_name	Adapter	Specifies the name the operating system uses to refer to an interface card installed on a node (such as en0).	“Step 5: Specify Adapter Stanzas” on page 95
adapter_stanzas	Machine	A list of adapter stanza names that define the adapters on a machine which can be requested.	“Step 1: Specify Machine Stanzas” on page 75
admin	Group, Class	A list of administrators for a group or class.	“Step 3: Specify Class Stanzas” on page 84
alias	Machine	Lists one or more alias names to associate with the machine name.	“Step 1: Specify Machine Stanzas” on page 75
central_manager	Machine	When **true**, this designates the machine as the LoadLeveler central manager.	“Step 1: Specify Machine Stanzas” on page 75
class_comment	Class	Text characterizing the class	“Step 3: Specify Class Stanzas” on page 84
core_limit	Class	Specifies the hard limit and/or soft limit for the size of a core file a job can create.	“Limit Keywords” on page 88
cpu_limit	Class	Specifies the hard limit and/or soft limit for the CPU time a job can use.	“Limit Keywords” on page 88
cpu_speed_scale	Machine	Determines whether CPU time is normalized according to machine speed.	“Step 1: Specify Machine Stanzas” on page 75
data_limit	Class	Specifies the hard limit and/or soft limit for the size of a data segment a job can use.	“Limit Keywords” on page 88
default_class	User	A class name that is the default value assigned to jobs submitted by users for which no class statement appears.	“Step 2: Specify User Stanzas” on page 81
default_group	User	A group name to which the user belongs.	“Step 2: Specify User Stanzas” on page 81

Admin. File Keyword	Stanza(s)	Brief Description	For Details
default_interactive_class	User	A class to which interactive jobs are assigned for jobs submitted by users who do not specify a class using LOADL_INTERACTIVE_CLASS.	"Step 2: Specify User Stanzas" on page 81
default_resources	Class	Specifies the default amount of resources consumed by a task of a job step, provided that no **resources** keyword is coded for the step in the job command file.	"Step 3: Specify Class Stanzas" on page 84
exclude_groups	Class	A list of groups names identifying those who cannot submit jobs of a particular class.	"Step 3: Specify Class Stanzas" on page 84
exclude_users	Class, Group	A list of user names identifying those who cannot submit jobs of a particular class or who are not members of the group.	"Step 3: Specify Class Stanzas" on page 84
feature	Machine	A string specifying unique characteristics of a machine.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
file_limit	Class	Specifies the hard limit and/or soft limit for the size of a file that a job can create.	"Limit Keywords" on page 88
include_groups	Class	A list of groups names identifying those who can submit jobs of a particular class.	"Step 3: Specify Class Stanzas" on page 84
include_users	Class, Group	A list of user names identifying those who can submit jobs of a particular class or who do belong to the group.	"Step 3: Specify Class Stanzas" on page 84
interface_address	Adapter	Specifies the IP address by which the adapter is known to other nodes in the network.	"Step 5: Specify Adapter Stanzas" on page 95
interface_name	Adapter	Specifies the name by which the adapter is known to other nodes in the network.	"Step 5: Specify Adapter Stanzas" on page 95
job_cpu_limit	Class	Specifies the hard limit and/or soft limit for the amount of CPU time an individual job step can use per processor.	"Limit Keywords" on page 88
machine_mode	Machine	Specifies the type of jobs this machine can run (batch, interactive, or both).	"Step 1: Specify Machine Stanzas" on page 75
master_node_exclusive	Machine	When **true**, this machine is used only as a master node for parallel jobs.	"Step 1: Specify Machine Stanzas" on page 75
master_node_requirement	Class	When **true**, jobs in this class have the requirement that they run on a master node having the **master_node_exclusive** setting.	"Step 3: Specify Class Stanzas" on page 84
max_adapter_windows	Machine	Specifies how many of a machine's available adapter windows LoadLeveler can use.	"Step 1: Specify Machine Stanzas" on page 75

Admin. File Keyword	Stanza(s)	Brief Description	For Details
maxidle	User, Group	Maximum number of idle jobs this user or group can have simultaneously.	“Step 2: Specify User Stanzas” on page 81
maxjobs	User, Class, Group	Maximum number of jobs this user, class, or group can have running simultaneously.	“Step 2: Specify User Stanzas” on page 81
max_jobs_scheduled	Machine	The maximum number of jobs that this machine can run.	“Step 1: Specify Machine Stanzas” on page 75
max_node	User, Class, Group	The maximum number of nodes a user can request for a parallel job.	“Step 2: Specify User Stanzas” on page 81
max_processors	User, Class, Group	The maximum number of machines a user can request for a parallel job.	“Step 2: Specify User Stanzas” on page 81
maxqueued	Group, User	The maximum number of jobs a single group or user can have queued at the same time.	“Step 2: Specify User Stanzas” on page 81
name_server	Machine	A list of nameservers used for a machine.	“Step 1: Specify Machine Stanzas” on page 75
network_type	Adapter	The type of network the adapter supports (for example, Ethernet). This is an administrator defined name.	“Step 5: Specify Adapter Stanzas” on page 95
nice	Class	Increments the *nice* value of a job.	“Step 3: Specify Class Stanzas” on page 84
NQS_class	Class	When **true**, any job submitted to this class is routed to an NQS machine.	“Step 3: Specify Class Stanzas” on page 84
NQS_query	Class	A list of queue names to use to monitor and cancel jobs.	“Step 3: Specify Class Stanzas” on page 84
NQS_submit	Class	A name that identifies the name of the NQS pipe queue to which the job will be routed.	“Step 3: Specify Class Stanzas” on page 84
pool_list	Machine	Specifies a list of pool numbers to which the machine belongs. Do not use negative numbers in a machine pool_list.	“Step 1: Specify Machine Stanzas” on page 75
priority	User, Class, Group	A number that identifies the priority of the appropriate user, class, or group.	“Step 2: Specify User Stanzas” on page 81
pvm_root	Machine	A directory in which PVM 3.3 is installed.	“Step 1: Specify Machine Stanzas” on page 75
resources	Machine	Specifies quantities of the consumable resources initially available on the machine.	“Step 1: Specify Machine Stanzas” on page 75
rss_limit	Class	Specifies the hard limit and/or soft limit for the resident set size for a job.	“Limit Keywords” on page 88
schedd_fenced	Machine	When **true**, the central manager ignores connections from this schedd machine.	“Step 1: Specify Machine Stanzas” on page 75

Admin. File Keyword	Stanza(s)	Brief Description	For Details
schedd_host	Machine	When **true**, this machine is used to help submit-only machines access LoadLeveler hosts that run LoadLeveler jobs.	"Step 1: Specify Machine Stanzas" on page 75
spacct_excluse_enable	Machine	Specifies whether the SP accounting function is informed whenever this machine is being used exclusively by a particular job.	"Step 1: Specify Machine Stanzas" on page 75
speed	Machine	The weight associated with the machine.	"Step 1: Specify Machine Stanzas" on page 75
stack_limit	Class	Specifies the hard limit and/or soft limit for the size of a stack.	"Limit Keywords" on page 88
submit_only	Machine	When **true**, designates this as a submit-only machine.	"Step 1: Specify Machine Stanzas" on page 75
switch_node_number	Adapter	The node on which the SP switch adapter is installed.	"Step 5: Specify Adapter Stanzas" on page 95
total_tasks	User, Class, Group	The maximum number of tasks a user can request for a parallel job.	"Step 2: Specify User Stanzas" on page 81
type	All	The type of stanza.	"Administering LoadLeveler" on page 74
wall_clock_limit	Class	Specifies the hard limit and/or soft limit for the amount of elapsed time for which a job can run.	"Limit Keywords" on page 88

Configuration File Keywords and LoadLeveler Variables

The following tables contain a brief description of the keywords you can use in the configuration file. The term *configuration file keywords* refers to keywords, user-defined variables, and LoadLeveler variables. A summary table is provided for each of the three types of configuration file keywords.

Keywords

The following table serves only as a reference. For more information on a specific keyword, see the section and page number referenced in the "For Details" column.

Configuration File Keyword	Brief Description	For Details
ACCT	Turns the accounting function on (or off).	"Step 9: Define Job Accounting" on page 110
ACCT_VALIDATION	The module called to perform account validation.	"Step 9: Define Job Accounting" on page 110
ACTION_ON_MAX_REJECT	Specifies whether a job is cancelled or put in User Hold or System Hold status when the job exceeds the **MAX_JOB_REJECT** value.	"Step 17: Specify Additional Configuration File Keywords" on page 129
ADMIN_FILE	Points to the administration file containing user, class, and machine list stanzas.	"Step 11: Specify Where Files and Directories are Located" on page 112
AFS_GETNEWTOKEN	A filter which can be used to renew an AFS token.	"Step 17: Specify Additional Configuration File Keywords" on page 129

Configuration File Keyword	Brief Description	For Details
ARCH	The standard architecture of the system.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
BIN	The directory where LoadLeveler binaries are kept.	"Step 11: Specify Where Files and Directories are Located" on page 112
CENTRAL_MANAGER_ HEARTBEAT_INTERVAL	The amount of time in seconds that defines how frequently primary and alternate central manager communicate with each other.	"Step 10: Specify Alternate Central Managers" on page 111
CENTRAL_MANAGER_TIMEOUT	The number of heartbeat intervals that an alternate central manager will wait before declaring that the primary central manager is not operating.	"Step 10: Specify Alternate Central Managers" on page 111
CLASS	The class of jobs that can run on the machine.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
CLIENT_TIMEOUT	The maximum time, in seconds, that a daemon waits for a response over TCP/IP from a process .	"Step 13: Define Network Characteristics" on page 116
COLLECTOR_DGRAM_PORT	The port number used when connecting to a daemon.	"Step 13: Define Network Characteristics" on page 116
CONTINUE	Continue expression. Determines if a job should continue.	"Step 8: Manage a Job's Status Using Control Expressions" on page 109
CUSTOM_METRIC	A machine's relative priority to run jobs.	"Step 2: Define LoadLeveler Cluster Characteristics" on page 99
CUSTOM_METRIC_COMMAND	An executable whose exit code is value is assigned to **CUSTOM_METRIC**.	"Step 2: Define LoadLeveler Cluster Characteristics" on page 99
DCE_ADMIN_GROUP	Specifies the DCE group containing the DCE ids of those users who will have administrator authority for the current cluster.	"Step 16: Configuring LoadLeveler to use DCE Security Services" on page 123
DCE_AUTHENTICATION_PAIR	A pair of installation supplied programs that are used to authenticate DCE security credentials.	"Step 17: Specify Additional Configuration File Keywords" on page 129
DCE_ENABLEMENT	Activates the exploitation of DCE security.	"Step 16: Configuring LoadLeveler to use DCE Security Services" on page 123
DCE_SERVICES_GROUP	Specifies the DCE group containing all of the principal names of the LoadLeveler daemons that are authorized to run in the current cluster.	"Step 16: Configuring LoadLeveler to use DCE Security Services" on page 123

Configuration File Keyword	Brief Description	For Details
DRAIN_ON_SWITCH_TABLE_ERROR	Specifies that the **startd** should be drained when the switch table fails to unload.	"Step 17: Specify Additional Configuration File Keywords" on page 129
EXECUTE	The local directory to store the executable checkpoints of jobs submitted by other machines.	"Step 11: Specify Where Files and Directories are Located" on page 112
FLOATING_RESOURCES	Specifies which consumable resources are available collectively on all of the machines in the LoadLeveler cluster.	"Step 4: Define Consumable Resources" on page 104
GLOBAL_HISTORY	The directory containing the global history files.	"Step 9: Define Job Accounting" on page 110
GSMONITOR	Location of the gsmonitor executable (LoadL_gsmonitor).	"The gsmonitor Daemon" on page 18
GSMONITOR_RUNS_HERE	When true, specifies that you want to start the gsmonitor daemon (you must have PSSP Groups Service).	"The gsmonitor Daemon" on page 18
HISTORY	The pathname of the history file for local LoadLeveler jobs.	"Step 11: Specify Where Files and Directories are Located" on page 112
JOB_ACCT_Q_POLICY	The amount of time in seconds that determines how often the startd daemon updates the schedd daemon with accounting data of running jobs.	"Chapter 7. Gathering Job Accounting Data" on page 153
JOB_EPILOG	Pathname of the epilog program.	"Writing Prolog and Epilog Programs" on page 297
JOB_LIMIT_POLICY	The amount of time in seconds that LoadLeveler checks to see if **job_cpu_limit** has been exceeded.	"Chapter 7. Gathering Job Accounting Data" on page 153
JOB_PROLOG	Pathname of the prolog program.	"Writing Prolog and Epilog Programs" on page 297
JOB_USER_EPILOG	Pathname of the user epilog program.	"Writing Prolog and Epilog Programs" on page 297
JOB_USER_PROLOG	Pathname of the user prolog program.	"Writing Prolog and Epilog Programs" on page 297
KBDD	KBDD expression. Location of kbdd executable (Loadl_kbdd).	"LoadLeveler Daemons" on page 6
KILL	Kill expression. Determines if vacated jobs should be killed.	"Step 8: Manage a Job's Status Using Control Expressions" on page 109
LIB	The directory where LoadLeveler libraries are kept.	"Step 11: Specify Where Files and Directories are Located" on page 112
LOADL_ADMIN	List of LoadLeveler administrators.	"Step 1: Define LoadLeveler Administrators" on page 99

Configuration File Keyword	Brief Description	For Details
LOCAL_CONFIG	Pathname of the optional local configuration file containing information specific to a node in the LoadLeveler network.	"Step 11: Specify Where Files and Directories are Located" on page 112
LOG	Local directory for storing log files.	"Step 11: Specify Where Files and Directories are Located" on page 112
MACHINE_AUTHENTICATE	Specifies whether machine validation is performed.	"Step 2: Define LoadLeveler Cluster Characteristics" on page 99
MACHINE_UPDATE_INTERVAL	The time, in seconds, during which machines must report to the central manager.	"Step 17: Specify Additional Configuration File Keywords" on page 129
MACHPRIO	Machine priority expression	"Step 7: Prioritize the Order of Executing Machines Maintained by the Negotiator" on page 106
MAIL	Name of a local mail program used to override default mail notification.	"Using Your Own Mail Program" on page 297
MASTER	Location of the master executable (LoadL_master).	"LoadLeveler Daemons" on page 6
MASTER_DGRAM_PORT	The port number used when connecting to the daemon.	"Step 13: Define Network Characteristics" on page 116
MASTER_STREAM_PORT	The port number to used when connecting to the daemon.	"Step 13: Define Network Characteristics" on page 116
MAX_CKPT_INTERVAL	The maximum number of seconds between checkpoints for running jobs.	"Step 14: Enable Checkpointing" on page 117
MAX_JOB_REJECT	The number of times a job is rejected before it is cancelled or put in User Hold or System Hold status.	"Step 17: Specify Additional Configuration File Keywords" on page 129
MAX_STARTERS	The maximum number of jobs that can run simultaneously.	"Step 5: Specify How Many Jobs a Machine Can Run" on page 104
MIN_CKPT_INTERVAL	The minimum number of seconds between checkpoints for running jobs.	"Step 14: Enable Checkpointing" on page 117
NEGOTIATOR	Location of the negotiator executable (LoadL_negotiator).	"LoadLeveler Daemons" on page 6
NEGOTIATOR_INTERVAL	The time interval, in seconds, at which the negotiator daemon updates the status of jobs in the LoadLeveler cluster and negotiates with machines that are available to run jobs.	"Step 17: Specify Additional Configuration File Keywords" on page 129

Configuration File Keyword	Brief Description	For Details
NEGOTIATOR_CYCLE_DELAY	The time, in seconds, the negotiator delays between periods when it attempts to schedule jobs. This time is used by the negotiator daemon to respond to queries, reorder job queues, collect information about changes in the states of jobs, etc. Delaying the scheduling of jobs might improve the overall performance of the negotiator by preventing it from spending excessive time attempting to schedule jobs.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_LOADAVG_ INCREMENT	The factor added to the startd machine's load average to compenstate for the increased load caused by starting another machine.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_PARALLEL_ DEFER	The length of time that a job is given to accumulate processors.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_PARALLEL_ HOLD	The length of time a job attempts to collect machines before releasing them.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_RECALCULATE_ SYSPRIO_INTERVAL	The amount of time in seconds between calculation of the **SYSPRIO** values for waiting jobs.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_REJECT_DEFER	The amount of time in seconds the negotiator waits before it considers scheduling a job to a machine that recently rejected the job.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_REMOVE_COMPLETED	The amount of time the negotiator keeps information on completed and removed jobs.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_RESCAN_QUEUE	The amont of time the negotiator waits to rescan the job queue for machines that temporarily have non-runnable jobs.	"Step 17: Specify Additional Configuration File Keywords" on page 129
NEGOTIATOR_STREAM_PORT	The port number used when connecting to the daemon.	"Step 13: Define Network Characteristics" on page 116
NQS_DIR	The directory where NQS commands reside.	"Step 11: Specify Where Files and Directories are Located" on page 112
OBITUARY_LOG_LENGTH	The number of lines from the ned of the file that are appended to the Master_Log.	"Step 17: Specify Additional Configuration File Keywords" on page 129

Configuration File Keyword	Brief Description	For Details
POLLING_FREQUENCY	The frequency in seconds the startd daemon uses to evaluate the load on the local machine and to decide whether to suspend, resume, or abort jobs.	"Step 17: Specify Additional Configuration File Keywords" on page 129
POLLS_PER_UPDATE	The frequency, in POLLING_FREQUENCY intervals, with which the startd daemon updates the central manager.	"Step 17: Specify Additional Configuration File Keywords" on page 129
PROCESS_TRACKING	When **true** ensures that when a job is terminated, no processes created by the job will continue running.	"Step 15: Specify Process Tracking" on page 122
PROCESS_TRACKING_EXTENSION	The directory containing the kernel extension binary **LoadL_pt_ke**.	"Step 15: Specify Process Tracking" on page 122
PUBLISH_OBITUARIES	When **true**, specifies that the master daemon sends mail to the administrator(s) when any daemon it manages dies abnormally.	"Step 17: Specify Additional Configuration File Keywords" on page 129
RELEASEDIR	The directory where all the LoadLeveler software resides.	"Step 11: Specify Where Files and Directories are Located" on page 112
RESOURCES	Specifies quantities of the consumable resources ″consumed″ by each task of a job step.	"Step 4: Define Consumable Resources" on page 104
RESTARTS_PER_HOUR	The number of times the master daemon attempts to restart a daemon that dies abnormally.	"Step 17: Specify Additional Configuration File Keywords" on page 129
SCHEDD	Location of the schedd executable (LoadL_schedd).	"LoadLeveler Daemons" on page 6
SCHEDD_INTERVAL	Specifies the interval, in seconds, at which the schedd daemon checks the local job queue.	"Step 17: Specify Additional Configuration File Keywords" on page 129
SCHEDD_RUNS_HERE	Specifies whether this daemon will run on the host.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
SCHEDD_SUBMIT_AFFINITY	Specifies whether the **llsubmit** command submits a job to the machine where the command was invoked provided the schedd daemon is running on the machine.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
SCHEDD_STREAM_PORT	The port number used when connecting to the daemon.	"Step 13: Define Network Characteristics" on page 116
SCHEDULE_BY_RESOURCES	Specifies which consumable resources are considered by the LoadLeveler schedulers.	"Step 4: Define Consumable Resources" on page 104

Configuration File Keyword	Brief Description	For Details
SCHEDULER_API	When **YES**, disables the native LoadLeveler scheduling algorithm.	"Step 2: Define LoadLeveler Cluster Characteristics" on page 99
SCHEDULER_TYPE	Specifies the LoadLeveler Backfill scheduling algorithm.	"Step 2: Define LoadLeveler Cluster Characteristics" on page 99
SPOOL	The local directory where LoadLeveler keeps the local job queue and checkpoint files.	"Step 11: Specify Where Files and Directories are Located" on page 112
START	Start expression. Determines if a machine can run a job.	"Step 8: Manage a Job's Status Using Control Expressions" on page 109
STARTD	Location of the startd executable (LoadL_startd).	"LoadLeveler Daemons" on page 6
STARTER	Location of the starter executable (LoadL_starter).	"LoadLeveler Daemons" on page 6
STARTD_RUNS_HERE	Specifies whether this daemon will run on the host.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
START_DAEMONS	Specifies whether to start the daemons on the machine.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101
STARTD_DGRAM_PORT	The port number used when connecting to the daemon.	"Step 13: Define Network Characteristics" on page 116
STARTD_STREAM_PORT	The port number used when connecting to the daemon.	"Step 13: Define Network Characteristics" on page 116
SUBMIT_FILTER	The program you want to run to filter a job script when the job is submitted.	"Filtering a Job Script" on page 296
SUSPEND	Suspend expresson. Determines if a job should be suspended.	"Step 8: Manage a Job's Status Using Control Expressions" on page 109
SYSPRIO	System priority expression.	"Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105
TRUNC_GSMONITOR_LOG_ON_OPEN	When **true**, specifies that the log file is restarted with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113
TRUNC_KBDD_LOG_ON_OPEN	When **true**, specifies the log file is restarted with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113
TRUNC_MASTER_LOG_ON_OPEN	When **true**, specifies the log file is re started with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113

Configuration File Keyword	Brief Description	For Details
TRUNC_NEGOTIATOR_LOG_ON_OPEN	When **true**, specifies the log file is restarted with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113
TRUNC_SCHEDD_LOG_ON_OPEN	When **true**, specifies the log file is restarted with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113
TRUNC_STARTD_LOG_ON_OPEN	When **true**, specifies the log file is restarted with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113
TRUNC_STARTER_LOG_ON_OPEN	When **true**, specifies the log file is restarted with every invocation of the daemon.	"Step 12: Record and Control Log Files" on page 113
VACATE	The vacate expression. Determines whether suspended jobs should be vacated.	"Step 8: Manage a Job's Status Using Control Expressions" on page 109
WALLCLOCK_ENFORCE	When **true**, specifies that the **wall_clock_limit** on the job will be enforced. The **WALLCLOCK_ENFORCE** keyword is only valid when the External Scheduler is enabled.	131
X_RUNS_HERE	When **true**, specifies that you want to start the keyboard daemon.	"Step 3: Define LoadLeveler Machine Characteristics" on page 101

User-Defined Keywords

The following table serves only as a reference. These keywords are described in more detail in "User-Defined Variables" on page 132.

Keyword	Brief Description
BackgroundLoad	Defines the variable **BackgroundLoad** and assigns to it a floating point constant. This might be used as a noise factor indicating no activity.
CPU_Busy	Defines the variable **CPU_Busy** and reassigns to it at each evaluation the Boolean value True or False, depending on whether the Berkeley one-minute load average is equal to or greater than the saturation level of 1.5.
CPU_Idle	Defines the variable **CPU_Idle** and reassigns to it at each evaluation the Boolean value True or False, depending on whether the Berkeley one-minute load average is equal or less than 0.7.
HighLoad	Is a keyword that the user can define to use as a saturation level at which no further jobs should be started.
HOUR	Defines the variable **HOUR** and assigns to it a constant integer value.
JobLoad	Defines the variable **JobLoad** which defines the load on the machine caused by running the job.
KeyboardBusy	Defines the variable **KeyboardBusy** and reassigns to it at each evaluation the Boolean value True or False, depending on whether the keyboard and mouse have been idle for fifteen minutes.
LowLoad	Defines the variable **LowLoad** and assigns to it the value of **BackgroundLoad**. This might be used as a restart level at which jobs can be started again and assumes only running 1 job on the machine.

Keyword	Brief Description
mail	Specifies a local program you want to use in place of the LoadLeveler default mail notification method.
MINUTE	Defines the variable **MINUTE** and assigns to it a constant integer value.
StateTimer	Defines the variable **StateTimer** and reassigns to it at each evaluation the number of seconds since the current state was entered.

LoadLeveler Variables

The following table serves only as a reference. For more information on a specific keyword, see the section and page number referenced in the "For Details" column.

Keyword	Brief Description	For Details
Arch	Standard architecture of the system.	"LoadLeveler Variables" on page 132
ClassSysprio	Job priority for the class.	"Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105
Cpus	Number of CPU's installed.	"LoadLeveler Variables" on page 132
ConsumableCpus	Number of ConsumableCpus currently available on the machine, if defined in **SCHEDULE_BY_RESOURCES**. If not, then it is the same as Cpus.	"LoadLeveler Variables" on page 132
ConsumableMemory	Amount of ConsumableMemory currently available on the machine, if defined in **SCHEDULE_BY_RESOURCES**. If not, then it is the same as Memory.	"LoadLeveler Variables" on page 132
ConsumableVirtualMemory	Amount of ConsumableVirtualMemory currently available on the machine, if defined in **SCHEDULE_BY_RESOURCES**. If not, then it is the same as VirtualMemory.	"LoadLeveler Variables" on page 132
CurrentTime	The UNIX date that includes the current system time, in seconds, since January 1, 1970.	"LoadLeveler Variables" on page 132
CustomMetric	The relative machine priority.	"LoadLeveler Variables" on page 132
Disk	Free disk in megabytes on the filesystem where checkpoints are stored.	"LoadLeveler Variables" on page 132
domain or domainname	Dynamically indicates the domain name of the current host machine where the program is running.	"LoadLeveler Variables" on page 132
EnteredCurrentState	Value of CurrentTime when the current state was entered.	"LoadLeveler Variables" on page 132
GroupQueuedJobs	The number of jobs either running or queued for the LoadLeveler group.	"Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105
GroupRunningJobs	The number of jobs currently running for the LoadLeveler group.	"Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105
GroupSysprio	The job priority for the group.	"Step 6: Prioritize the Queue Maintained by the Negotiator" on page 105

Keyword	Brief Description	For Details
GroupTotalJobs	The total number of jobs associated with the LoadLeveler group.	“Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105
host or hostname	Dynamically indicates the name of the host machine where the program is running.	“LoadLeveler Variables” on page 132
KeyboardIdle	Number of seconds since the keyboard or mouse was last used.	“LoadLeveler Variables” on page 132
LoadAvg	Berkeley one-minute load average.	“LoadLeveler Variables” on page 132
Machine	Name of the current machine.	“LoadLeveler Variables” on page 132
MasterMachPrio	A value that is 1 for master nodes and is 0 otherwise.	“LoadLeveler Variables” on page 132
Memory	Physical memory installed on the machine in megabytes.	“LoadLeveler Variables” on page 132
OpSys	Indicates the operating system on the host where the program is running.	“LoadLeveler Variables” on page 132
QDate	Difference in seconds between when the negotiator starts up and when the job is submitted.	“LoadLeveler Variables” on page 132
Speed	The relative machine speed.	“LoadLeveler Variables” on page 132
State	State of the startd. Can be None, Busy, Running, Idle, Suspend, Flush, or Drain.	“LoadLeveler Variables” on page 132
tilde	Dynamically defines the pathname of the LoadLeveler home directory.	“LoadLeveler Variables” on page 132
tm_hour	Number of hours since midnight (0-23).	“LoadLeveler Variables” on page 132
tm_isdst	Daylight Savings Time flag: positive when in effect, zero when not in effect, negative when information is unavailable.	“LoadLeveler Variables” on page 132
tm_mday	Number of the day of the month (1-31).	“LoadLeveler Variables” on page 132
tm_min	Number of minutes after the hour (0-59).	“LoadLeveler Variables” on page 132
tm_mon	Number of months since January (0-11).	“LoadLeveler Variables” on page 132
tm_sec	Number of seconds after the minute (0-59).	“LoadLeveler Variables” on page 132
tm_wday	Number of days since Sunday (0-6).	“LoadLeveler Variables” on page 132
tm_yday	Number of days since January 1 (0-365).	“LoadLeveler Variables” on page 132
tm_year	Number of years since 1900 (0-9999).	“LoadLeveler Variables” on page 132
tm4_year	The integer representation of the current year.	“LoadLeveler Variables” on page 132
UserPrio	User defined priority of a job.	“Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105
UserQueuedJobs	The number of jobs either running or queued for the user.	“Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105
UserRunningJobs	The number of jobs currently running for the user.	“Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105
UserSysprio	The priority of the user who submitted the job.	“Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105

Keyword	Brief Description	For Details
UserTotalJobs	The total number of jobs associated with the this user.	“Step 6: Prioritize the Queue Maintained by the Negotiator” on page 105
VirtualMemory	The size of the available swap space on the machine in kilobytes.	“LoadLeveler Variables” on page 132

Chapter 6. Administration Tasks for Parallel Jobs

This chapter describes administration tasks that apply to parallel jobs. For more general information on administering and configuring LoadLeveler, see “Chapter 5. Administering and Configuring LoadLeveler” on page 71. For information on submitting parallel jobs, see “Chapter 4. Submitting and Managing Parallel Jobs” on page 59.

Scheduling Considerations for Parallel Jobs

For parallel jobs, the LoadLeveler Backfill scheduler makes the most efficient use of your resources. This scheduler runs both serial and parallel jobs, but is meant primarily for installations running parallel jobs.

The Backfill scheduler also supports:

- Multiple tasks per node
- Multiple user space tasks per adapter

You specify the Backfill scheduler using the **SCHEDULER_TYPE** keyword. For more information on this keyword and other schedulers you can run, see “Choosing a Scheduler” on page 100.

Allowing Users to Submit Interactive POE Jobs

Follow the steps in this section to set up your system so that users can submit interactive POE jobs to LoadLeveler.

1. Make sure that you have installed LoadLeveler and defined LoadLeveler administrators. See “Quick Set Up” on page 73 for information on defining LoadLeveler administrators.
2. Run the **llextSDR** command to extract node and adapter information from the SDR. See “llextSDR - Extract adapter information from the SDR” on page 182 for information on using this command.
3. Incorporate the appropriate node and adapter information into your LoadLeveler administration file stanzas.

   For example, the following output represents two adapter stanzas and their corresponding machine stanza:

```
k10n09.ppd.pok.ibm.com: type = adapter
 adapter_name = en0
 network_type = ethernet
 interface_address = 9.114.51.73
 interface_name = k10n09.ppd.pok.ibm.com


k10sn09.ppd.pok.ibm.com: type = adapter
 adapter_name = css0
 css_type = SP_Switch_MX_Adapter
 network_type = switch
 interface_address = 9.114.51.137
 interface_name = k10sn09.ppd.pok.ibm.com
 switch_node_number = 8


k10n09.ppd.pok.ibm.com: type=machine
 adapter_stanzas = k10n09.ppd.pok.ibm.com k10sn09.ppd.pok.ibm.com
 spacct_exclusive_enable = true
```

4. Define a machine to act as the LoadLeveler central manager. See “Quick Set Up” on page 73 for more information.
5. Define your scheduler to be the LoadLeveler Backfill scheduler by specifying **SCHEDULER_TYPE = BACKFILL** in the LoadLeveler configuration file. See “Choosing a Scheduler” on page 100 for more information.
6. Consider setting up a class stanza for your interactive POE jobs. See “Setting Up a Class for Parallel Jobs” on page 151 for more information. Define this class to be your default class for interactive jobs by specifying this class name on the **default_interactive_class** keyword. See “Step 2: Specify User Stanzas” on page 81 for more information.
7. Configure optional functions, including:
   - Setting up pools: you can organize nodes into pools by using the **pool_list** keyword in the machine stanza. See “Step 1: Specify Machine Stanzas” on page 75 for more information.
   - Specifying batch, interactive, or general use for nodes: you can use the **machine_mode** keyword in the machine stanza to specify the type of jobs that can run on a node.
   - Enabling SP exclusive use accounting: you can specify that the accounting function on an SP system be informed that a job step has exclusive use of a machine by specifying **spacct_exclusive_enable = true** in the machine stanza (as shown in the previous example).

     See “Step 1: Specify Machine Stanzas” on page 75 for more information on these keywords.
8. Start LoadLeveler using the **llctl** command. See “Quick Set Up” on page 73 for more information.

Allowing Users to Submit PVM Jobs

If users will be submitting PVM jobs, your installation must first obtain and install PVM. PVM is a public domain package distributed through electronic mail by Oak Ridge National Labs. To obtain information on PVM, issue the following:

```
echo "send index from pvm3" | mail netlib@ornl.gov
```

For RS6K architecture PVM, LoadLeveler expects to find PVM installed in **loadl/pvm3**. You can override this using the **pvm_root** entry in the machine stanza. The value of **pvm_root** is used to set the environment variable **$(PVM_ROOT)** required by PVM . For example:

```
gallifrey:  type = machine
central_manager = true
schedd_host = true
alias = drwho
pvm_root = /home/userid/loadl/2.2.0/aix43/pvm3
```

For PVM 3.3.11+ (that is, SP2MPI architecture), LoadLeveler does not expect to find PVM installed in **loadl/pvm3**. PVM 3.3.11+ must be installed in a directory accessable to, and executable by, all nodes in the LoadLeveler cluster. Administrators must communicate the location of this directory to their users.

Running PVM requires that each user be allowed to run only one instance of PVM per machine. In order to ensure that LoadLeveler does not attempt to start more than one PVM job per machine, you can set up a class for PVM jobs. To do this, you need to add a class stanza to your administration file and a class statement to your configuration file. The following is an example of a PVM class stanza that you can add to your administration file:

```
PVM3:  type = class
max_node = 15  # max of 15 processors per user per job
```

The following is an example of statements that you can add to your configuration file:

```
MAX_STARTERS = 2
Class = {"ClassA" "ClassA" "PVM3" }
```

This combination of the **MAX_STARTERS** keyword and the **Class** keyword allows two jobs of Class A, or one job of Class A and one of class PVM3, to start. Limiting PVM jobs by using a class where **MAX_STARTERS** is greater than 1 is only a policy. The user can still submit a PVM job to Class A. Note also that specifying **MAX_STARTERS=1** would enforce a policy of one job per machine.

See “Common Set Up Problems with Parallel Jobs” on page 307 for more information.

Restrictions and Limitations for PVM Jobs

For PVM 3.3, dynamic allocation and de-allocation of parallel machines are not supported.

Setting Up a Class for Parallel Jobs

To define the characteristics of parallel jobs run by your installation you should set up a class stanza in the administration file and define a class (in the **Class** statement in the configuration file) for each task you want to run on a node.

Suppose your installation plans to submit long-running parallel jobs, and you want to define the following characteristics:

- Only certain users can submit these jobs
- Jobs have a 30 hour run time limit
- A job can request a maximum of 60 nodes and 120 total tasks
- Jobs will have a relatively low run priority

The following is a sample class stanza for long-running parallel jobs which takes into account the above characteristics:

```
long_parallel: type=class
wall_clock_limit = 1800
include_users = jack queen king ace
priority = 50
total_tasks = 120
max_node = 60
maxjobs = 2
```

Note the following about this class stanza:

- The **wall_clock_limit** keyword sets a wall clock limit of 1800 seconds (30 hours) for jobs in this class
- The **include_users** keyword allows four users to submit jobs in this class
- The **priority** keyword sets a relative priority of 50 for jobs in this class
- The **total_tasks** keyword specifies that a user can request up to 120 total tasks for a job in this class
- The **max_node** keyword specifies that a user can request up to 60 nodes for a job in this class

- The **maxjobs** keyword specifies that a maximum of two jobs in this class can run simultaneously

Suppose users need to submit job command files containing the following statements:

```
node = 30
tasks_per_node = 4
```

You must code the **Class** statement such that at least 30 nodes have four or more long_parallel classes defined. That is, the configuration file for each of these nodes must include the following statement:

```
Class = { "long_parallel" "long_parallel" "long_parallel" "long_parallel" }
```

Setting Up a Parallel Master Node

LoadLeveler allows you to define a parallel master node–which LoadLeveler will use as the first node for a job submitted to a particular class. To set up a parallel master node, code the following keywords in the node's class and machine stanzas in the administration file:

```
# MACHINE STANZA: (optional)
mach1:     type = machine
master_node_exclusive = true


# CLASS STANZA: (optional)
pmv3:      type = class
master_node_requirement = true
```

Specifying **master_node_requirement = true** forces all parallel jobs in this class to use–as their first node–a machine with the **master_node_exclusive = true** setting. For more information of these keywords, see "Step 1: Specify Machine Stanzas" on page 75 and "Step 3: Specify Class Stanzas" on page 84.

Chapter 7. Gathering Job Accounting Data

Your organization may have a policy of charging users or groups of users for the amount of resources that their jobs consume. You can do this using LoadLeveler's accounting feature. Using this feature, you can produce accounting reports that contain job resource information for completed serial and parallel jobs. You can also view job resource information on jobs that are continuing to run.

Collecting Job Resource Data on Serial and Parallel Jobs

Information on completed serial and parallel jobs is gathered using the UNIX *wait3* system call. Information on non-completed serial and parallel jobs is gathered in a platform-dependent manner by examining data from the UNIX process.

Accounting information on a completed serial job is determined by accumulating resources consumed by that job on the machine(s) that ran the job. Similarly, accounting information on completed parallel jobs is gathered by accumulating resources used on all of the nodes that ran the job.

You can also view resource consumption information on serial and parallel jobs that are still running by specifying the **-x** option of the **llq** command. In order to enable **llq -x**, you should specify the following keywords in the configuration file:

ACCT = A_ON A_DETAIL
Turns accounting data recording on. For more information on this keyword, see "Step 9: Define Job Accounting" on page 110.

JOB_ACCT_Q_POLICY = *number*
where *number* is the amount of time in seconds that determines how often the startd daemon updates the schedd daemon with accounting data of running jobs. This controls the accuracy of the **llq -x** command. The default is 300 seconds.

JOB_LIMIT_POLICY = *number*
where *number* is an amount of time in seconds. The smaller of **JOB_LIMIT_POLICY** and **JOB_ACCT_Q_POLICY** is used to control how often the **startd** daemon collects resource consumption data on running jobs, and how often the **job_cpu_limit** is checked. The default for **JOB_LIMIT_POLICY** is **POLLING_FREQUENCY** multiplied by **POLLS_PER_UPDATE**.

Collecting Job Resource Data Based on Machines

LoadLeveler can collect job resource usage information for every machine on which a job may run. A job may run on more than one machine because it is a parallel job or because the job is vacated from one machine and rescheduled to another machine.

To enable recording of resources by machine, you need to specify **ACCT = A_ON A_DETAIL** in the configuration file.

The machine's speed is part of the data collected. With this information, an installation can develop a charge back program which can charge more or less for resources consumed by a job on different machines. For more information on a machine's speed, refer to the machine stanza information. See "Step 1: Specify Machine Stanzas" on page 75.

Collecting Job Resource Data Based on Events

In addition to collecting job resource information based upon machines used, you can gather this information based upon an event or time that you specify. For example, you may want to collect accounting information at the end of every work shift or at the end of every week or month. To collect accounting information on all machines in this manner, use the **llctl** command with the **capture** parameter:

llctl -g capture *eventname*

eventname is any string of continuous characters (no white space is allowed) that defines the event about which you are collecting accounting data. For example, if you were collecting accounting data on the *graveyard* work shift, your command could be:

```
llctl -g capture graveyard
```

This command allows you to obtain a snapshot of the resources consumed by active jobs up to and including the moment when you issued the command. If you want to capture this type of information on a regular basis, you can set up a crontab entry to invoke this command regularly. For example:

```
#  sample crontab for accounting
#  shift crontab 94/8/5
#
# Set up three shifts, first, second, and graveyard shift.
#  Crontab entries indicate the end of shift.
#
#M  H d m day command
#
00 08 * *  * /u/loadl/bin/llctl -g capture graveyard
00 16 * *  * /u/loadl/bin/llctl -g capture first
00 00 * *  * /u/loadl/bin/llctl -g capture second
```

For more information on the **llctl** command, refer to “llctl - Control LoadLeveler Daemons” on page 175. For more information on the collection of accounting records, see “llq - Query Job Status” on page 193.

Collecting Job Resource Information Based on User Accounts

If your installation is interested in keeping track of resources used on an account basis, you can require all users to specify an account number in their job command files. They can specify this account number with the **account_no** keyword which is explained in detail in “Job Command File Keywords” on page 36. Interactive POE jobs can specify an account number using the LOADL_ACCOUNT_NO environment variable.

LoadLeveler validates this account number by comparing it against a list of account numbers specified for the user in the user stanza in the administration file.

Account validation is under the control of the **ACCT** keyword in the configuration file. The routine which performs the validation is called **llacctval**. You can supply your own validation routine by specifying the **ACCT_VALIDATION** keyword in the configuration file. The following are passed as character string arguments to the validation routine:

- User name
- User’s login group name
- Account number specified on the Job

- Blank separated list of account numbers obtained from the user's stanza in the administration file.

The account validation routine must exit with a return code of zero if the validation succeeds. If it fails, the return code is a non-zero number.

Collecting the Accounting Information and Storing it into Files

LoadLeveler stores the accounting information that it collects in a file called *history* in the spool directory of the machine that initially scheduled this job, the schedd machine. Data on parallel jobs are also stored in the *history* files.

Resource information collected on the LoadLeveler job is constrained by the capabilities of the wait3 system call. Information for processes which fork child processes will include data for those child processes as long as the parent process waits for the child process to terminate. Complete data may not be collected for jobs which are not composed of simple parent/child processes. For example, if you have a LoadLeveler job which invokes an rsh command to execute a function on another machine, the resources consumed on the other machine will not be collected as part of the LoadLeveler accounting data.

LoadLeveler accounting uses the following types of files:

- The local history file which is local to each schedd machine is where job resource information is first recorded. These files are usually named *history* and are located in the spool directory of each schedd machine, but you may specify an alternate name with the **HISTORY** keyword in either the global or local configuration file. For more information, refer to the "Step 9: Define Job Accounting" on page 110.
- The global history file is a combination of the history files from some or all of the machines in the LoadLeveler cluster merged together. The command **llacctmrg** is used to collect files together into a global file. As the files are collected from each machine, the local history file for that machine is reset to contain no data. The file is named *globalhist.YYYYMMDDHHmm*. You may specify the directory in which to place the file when you invoke the **llacctmrg** command or you can specify the directory with the **GLOBAL_HISTORY** keyword in the configuration file. The default value set up in the sample configuration file is the local spool directory:

GLOBAL_HISTORY = *$(SPOOL) (optional)*

Accounting Reports

You can produce three types of reports using either the local or global history file. These reports are called the *short*, *long*, and *extended* versions. As their names imply, the short version of the report is a brief listing of the resources used by LoadLeveler jobs. The long version provides more comprehensive detail with summarized resource usage and the extended version of the report provides the comprehensive detail with detailed resource usage. If you do not specify a report type, you will receive the default short version.

The short report displays the number of jobs along with the total CPU usage according to user, class, group, and account number. The extended version of the report displays all of the data collected for every job. See the **llsummary** command, "llsummary - Return Job Resource Information for Accounting" on page 214, for examples of the short and extended versions of the report.

For information on the accounting Application Programming Interfaces, refer to “Chapter 11. LoadLeveler APIs” on page 251.

Sample Job Accounting Scenario

The following sample scenario walks you through the process of collecting account data. You can perform all of the steps or just the ones that apply to your situation.

Task 1: Update the Configuration File

Edit the configuration file according to the following table:

Edit this keyword:	To:
GLOBAL_HISTORY	Specify a directory in which to place the global history files.
ACCT	Turn accounting and account validation on and off and specify detailed accounting.
ACCT_VALIDATION	Specify the account validation routine.
Note: See “Step 9: Define Job Accounting” on page 110 for more information on these keywords.	

Task 2: Merge Multiple Files Collected From Each Machine Into One File

You can accomplish this step using either the **llacctmrg** command or the graphical user interface:

- Using **llacctmrg**: See “llacctmrg - Collect machine history files” on page 168 for the syntax of this command.
- Using the graphical user Interface:

  Select A machine from the Machines window

  Select **Admin → Collect Account Data...** from the Machines window.

  ▲ A window appears prompting you to enter a directory name where the file will be placed. If no directory is specified, the directory specified with the **GLOBAL_HISTORY** keyword in the global configuration file is the default directory.

  Press **OK**

  ▲ The window closes and you return to the main window.

Task 3: Report Job Information on all the Jobs in the History File

You can accomplish this step using either the **llsummary** command or the graphical user interface:

- Using **llsummary**: see “llsummary - Return Job Resource Information for Accounting” on page 214 for the syntax of this command.
- Using the graphical user interface:

  Select **Admin → Create Account Report...** from the Machines window.

  Note: If you want to receive an extended accounting report, select the **extended** cascading button.

  ▲ A window appears prompting you to enter the following information:

  – A short, long, or extended version of the output. The short version is the default version.

– Start and end date ranges for the report. If no date is specified, the default is to report all of the data in the report.
– The name of the input data file.
– The name of the output data file.

Press OK

▲ The window closes and you return to the main window. The report appears in the Messages window if no output data file was specified.

Task 4: Using Account Numbers and Setting Up Account Validation

1. Specify the following keyword in the user stanza in the administration file:

   account = *list*
   where *list* is a blank delimited list of account numbers a user may use when submitting jobs.
2. Instruct users to associate an account number with their job:
   - Using the job command file: add the **account_no** keyword to the job command file. See "Job Command File Keywords" on page 36 for details.
   - Using the graphical user interface:

     Select File → Build a Job from the main window.

     ▲ The Build a Job window appears.

     Type the account number in the **account_no** field on the Build a Job window.

     Press OK

     ▲ The window closes and you return to the main window.
3. Specify the **ACCT_VALIDATION** keyword in the configuration file that identifies the module that will be called to perform account validation. The default module is called **llacctval**. You can replace this module with your installation's own accounting routine by specifying a new module with this keyword.

Task 5: Specifying Machines and Their Weights

To specify weights to associate with machines, specify the following keyword in a machine's machine stanza in the administration file:

speed = *number*
where *number* defines the weight associated with a particular machine. The higher numbers correspond with a greater weight. The default weight is 1.0.

Also, if you have in your cluster machines of differing speeds and you want LoadLeveler accounting information to be normalized for these differences, specify **cpu_speed_scale=true** in each machine's respective machine stanza.

For example, suppose you have a cluster of two machines, called A and B, where Machine B is three times as fast as Machine A. Machine A has **speed=1.0**, and Machine B has **speed=3.0**. Suppose a job runs for 12 CPU seconds on Machine A. The same job runs for 4 CPU seconds on Machine B. When you specify **cpu_speed_scale=true**, the accounting information collected on Machine B for that job shows the normalized value of 12 CPU seconds rather than the actual 4 CPU seconds.

Chapter 8. Routing Jobs to NQS Machines

Users can submit NQS scripts to LoadLeveler and have them routed to a machine outside of the LoadLeveler cluster that runs NQS. LoadLeveler supports COSMIC NQS version 2.0 and other versions of NQS that support the same commands and options and produce similar output for those commands.

The following diagram illustrates a typical environment that allows users to have their jobs routed to machines outside of LoadLeveler for processing:

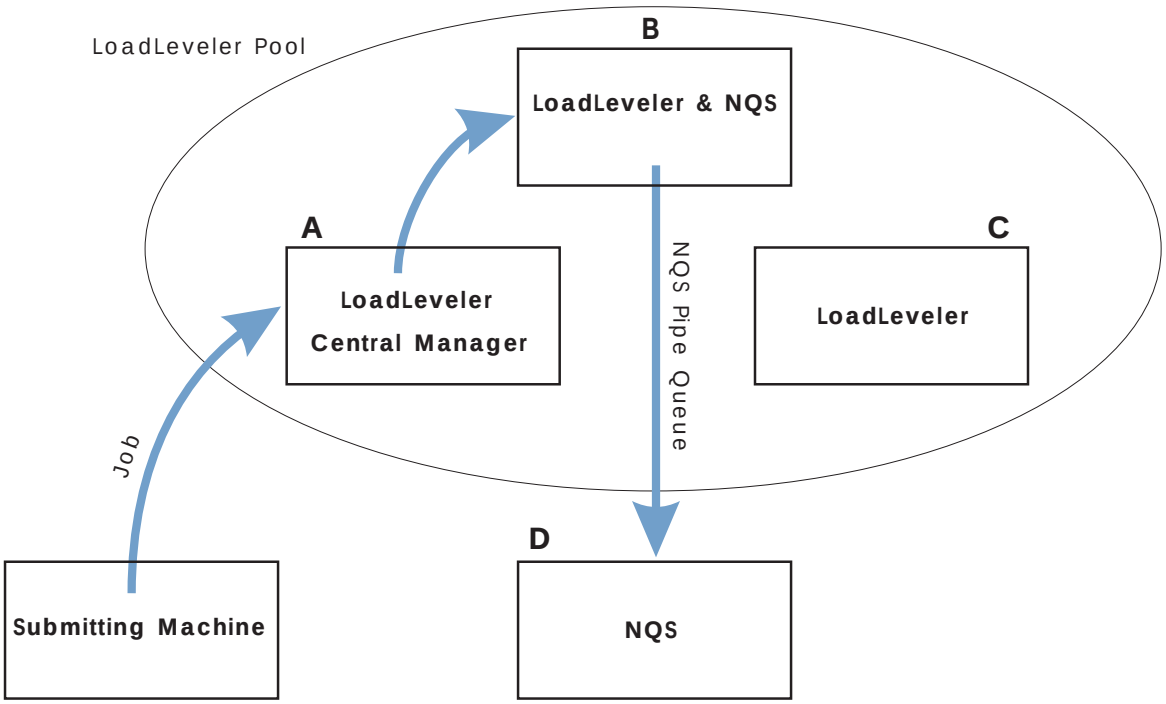

Figure 31. Environment illustrating jobs being routed to NQS machines.

As the diagram illustrates, machines A, B, and C, are members of the LoadLeveler cluster. Machine A has the central manager running on it and machine B has both LoadLeveler and NQS running on it. Machine C is a third member of the cluster. Machine D is outside of the cluster and is running NQS.

When a user submits a job to LoadLeveler, machine A, that runs the central manager, schedules the job to machine B. LoadLeveler running on machine B routes the job to machine D using NQS. Keep this diagram in mind as you continue to read this chapter.

Setting Up the NQS Environment

Setting up the NQS environment involves the following:

- Install NQS on each node that an NQS class is defined. In the previous diagram, this is machine B.
- Create an NQS pipe queue on the LoadLeveler machine whose destination is the NQS batch queue on the machine designated to run the NQS jobs.

  In the previous diagram, you would create the NQS pipe queue on machine B.

- Create an NQS batch queue on the machine designated to run the NQS jobs. In the previous diagram, this is machine D.

Designating Machines to Which Jobs Will be Routed

To designate a machine to which your jobs will be routed, follow these steps:

1. Set up a special class in the **LoadL_admin** file by adding the following class definitions to the file:

   NQS_class = true **| false**
   When this flag is set to **true**, any job submitted to this class will be routed to an NQS machine.

   NQS_submit = *name*
   The name of the NQS pipe queue to which the job will be routed. When the job is dispatched by LoadLeveler, LoadLeveler will invoke the **qsub** command using the name of the this queue.

   NQS_query = *queue names*
   A blank delimited list of queue names (including host names if necessary) to be used with the **qstat** command to monitor the job and **qdel** to cancel the job.

   You can set up multiple classes to access different machines.
2. Modify the local configuration file on the machines that you want to accept this class of jobs.
3. Add the **NQS_DIR** keyword to the **LoadL_config** file:

   NQS_DIR = *NQS directory*
   defines the directory where NQS commands **qsub**, **qstat**, and **qdel** reside. The default is **/usr/bin**.

Sample Routing Jobs to NQS Machines Scenario

The following example walks you through the process of setting up your environment to route jobs to machines that run NQS.

Assume Figure 31 on page 159 depicts your environment. You have three machines in the cluster named A, B, and C. Outside of the cluster, you have machine D running NQS.

Task 1: Modify the Administration File

After setting up your NQS environment, modify the **LoadL_admin** file by defining the class **NQS** including the following stanzas:

```
NQS:
type = class
NQS_class = true
NQS_submit = pipe_a
NQS_query = queue@chevy.kgn.ibm.com
```

Task 2: Modify the Configuration File

Modify the **LoadL_config.local** on the machine(s) that you want to accept this class of jobs. In this example, you would modify machine B's **LoadL_config.local** file. To do this, add a class statement similar to:

```
CLASS = {"NQS" "a" "b" ....}
```

where `NQS` is the name of the class of jobs that will be routed to the machines that run NQS, and `a` and `b` are names of additional classes.

Task 3: Submit the Jobs

After you perform the previous tasks, users can route their jobs to machines running NQS using the **llsubmit** command. The job command file must specify the **class** keyword. For example:

```
class = NQS
```

The job command file must also contain the shell script to be submitted to the NQS node. NQS accepts only shell scripts, binaries are not allowed. All options in the command file pertaining to scheduling the job will be used by LoadLeveler to schedule the job. When the job is dispatched to the node running the specified NQS class, the LoadLeveler options pertaining to the runtime environment are converted to NQS options and the job is submitted to the specified NQS queue.

LoadLeveler command file options are used as follows:

arguments
: error message generated and job not submitted

checkpoint
: error message generated and job not submitted

class used only for LoadLeveler scheduling

core_limit
: converted to **-lc** option

cpu_limit
: converted to **-lt** option

data_limit
: converted to **-ld** option

environment
: if COPY_ALL is specified, the option is converted to **-x**, otherwise error message generated and job not submitted

error converted to **-e**

executable
: error message generated and job not submitted

file_limit
: converted to **-lf** option

hold used only for LoadLeveler scheduling

image_size
: error message generated and job not submitted

initialdir
: error message generated and job not submitted

input error message generated and job not submitted

notification
: If the option specified is

  always
  : converted to **-mb**and **-me** options

  error converted to **-me** option

start converted to **-mb** option

never ignored

complete
converted to **-me** option

notify_user
converted to **-mu** option

output
converted to **-o** option

preferences
used only for LoadLeveler scheduling

queue places one copy of job in the LoadLeveler queue

requirements
used only for LoadLeveler scheduling

restart
If the option specified is

yes ignored

no converted to **-nr** option

rss_limit
converted to **-lw** option

shell converted to **-s** option

stack_limit
converted to **-ls** option

start_date
used only for LoadLeveler scheduling

user_priority
used only for LoadLeveler scheduling

Users can also submit an NQS script. In this case, any NQS options in the script are used to schedule the job and once dispatched by LoadLeveler, the file is sent to NQS unmodified.

LoadLeveler schedules these jobs the same as it schedules other jobs. When the job is dispatched, LoadLeveler determines whether or not it is running in an NQS class. If it is, an NQS command **qsub** is issued.

LoadLeveler monitors the job by periodically invoking a **qstat** command. A **qstat** command is first issued for the pipe queue on the local host. If the request id is not found, a **qstat** is issued for each queue listed in the NQS_query class keyword. If the request id is still not found, starter marks the job as complete.

When a job is sent to an NQS class, **llsubmit** saves the following environment variables:
- HOME
- LOGNAME
- MAIL
- PATH
- SHELL
- TZ

- USER

When LoadLeveler dispatches the job, these environment variables are installed so that they are available to **qsub**. **llsubmit** also saves the name of the current directory (pwd) and the current value of the user file create mask (umask).

Task 4: Obtain Status of NQS Jobs

Users can obtain status of NQS jobs in the same way as they obtain status of LoadLeveler jobs - either by using the **llq** command or by viewing the Jobs window on the graphical user interface. The users can identify the NQS jobs by the class field on the Jobs window.

LoadLeveler monitors the job until **qstat** shows the job is no longer in any specified queue.

NQS does not provide job accounting. Therefore, the only accounting information LoadLeveler will have is the total time for the job.

LoadLeveler will not send mail when the job completes. The LoadLeveler notification option is translated to the appropriate NQS flag (me or mb) and NQS will send the mail.

Task 5: Cancel NQS Jobs

Users can cancel NQS jobs using the LoadLeveler **llcancel** command. All they need to know is the LoadLeveler job id for the NQS job. Once they submit their request to cancel the job, LoadLeveler forwards their request to the appropriate node and a **qdel** will be issued for the job for the queue listed in the the **NQS_submit** and **NQS_query** keywords.

NQS Scripts

Scripts originally written for NQS that contain NQS options are acceptable to LoadLeveler. The options are mapped as closely as possible to the features provided by LoadLeveler, but the exact function is not always available. NQS options map to LoadLeveler as follows:

a startdate
e error
ke ignored
ko ignored
lc core_limit
ld data_limit
lf file_limit
lm rss_limit
lM ignored
ln ignored
ls stack_limit
lt cpu_limit
lT ignored
lv ignored
lw ignored
mb notification (always)
me notification (complete)
mu notify_user
nr restart = no
o output

p user_priority
q class
r ignored
re ignored
ro ignored
s shell
x environment = copyall
z suppresses messages but not mail

Part 4. Command Reference

Chapter 9. LoadLeveler Commands

LoadLeveler provides two types of commands: those that are available to all users of LoadLeveler, and those that are reserved for LoadLeveler administrators. If DCE is not used, then administrators are identified by the LOADL_ADMIN keyword in the configuration file. If DCE is enabled with DCE_ENABLEMENT=TRUE, the members of the DCE group specified by the keyword DCE_ADMIN_GROUP are LoadLeveler administrators.

The administrator commands can operate on the entire LoadLeveler job queue and all machines configured. The user commands mainly affect those jobs submitted by that user. Some commands, such as **llhold**, include options that can only be performed by an administrator.

Summary of LoadLeveler Commands

The following table summarizes the LoadLeveler commands:

Command	Description	Who Can Issue?	For More Information
llacctmrg	Collects all individual machine history files together into a single file.	Administrators	See page 168
llcancel	Cancels a submitted job.	Users and Administrators	See page 170
llclass	Returns information about LoadLeveler classes.	Users and Administrators	See page 172
llctl	Controls daemons on one or more machines in the LoadLeveler cluster.	Administrators	See page 175
lldcegrpmaint	Sets up DCE groups and principal names.	DCE Administrators	See page180
llextSDR	Extracts adapter information from the system data repository (SDR).	Administrators	See page 182
llfavorjob	Raises one or more jobs to the highest priority, or restores original priority.	Administrators	See page 185
llfavoruser	Raises job(s) submitted by one or more users to the highest priority, or restores original priority.	Administrators	See page 186
llhold	Holds or releases a hold on a job.	Users and Administrators	See page 187
llinit	Initializes a new machine as a member of the LoadLeveler cluster.	Administrators	See page 189
llprio	Changes the user priority of a submitted job step.	Users and Administrators	See page 191
llq	Queries the status of LoadLeveler jobs.	Users and Administrators	See page 193
llstatus	Queries the status of LoadLeveler machines.	Users and Administrators	See page 205
llsubmit	Submits a job.	Users and Administrators	See page 213
llsummary	Returns resource information on completed jobs.	Administrators	See page 214

llacctmrg - Collect machine history files

Purpose
Collects individual machine history files together into a single file specified as a parameter.

Syntax
llacctmrg [**-?**] [**-H**] [**-v**] [**-h** *hostlist*] [**-d** *directory*]

Flags
-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-h *hostlist*
Specifies a blank delimited list of machines from which to collect data. The default is all machines in the LoadLeveler cluster.

-d *directory*
Specifies the directory to hold the new global history file. If not specified, the directory specified in the **GLOBAL_HISTORY** keyword in the configuration file is used.

Description
This command by default collects data from all the machines identified in the administration file. To override the default, specify a machine or a list of machines using the **-h** flag.

When the **llacctmrg** command ends, accounting information is stored in a file called **globalhist.***YYYYMMDDHHmm*. Information such as the amount of resources consumed by the job and other job-related data is stored in this file. In this file:
YYYY indicates the year
MM indicates the month
DD indicates the day
HH indicates the hour
mm indicates the minute.

You can use this file as input to the **llsummary** command. For example, if you created the file **globalhist.199808301050**, you can issue **llsummary globalhist.199808301050** to record information on all machines.

Data on processes which fork child processes will be included in the file only if the parent process waits for the child process to end. Therefore, complete data may not be collected for jobs which are not composed of simple parent/child processes. For example, if a LoadLeveler job invokes an **rsh** command to execute some function on another machine, the resources consumed on the other machine will not be collected as part of the accounting data.

Examples
The following example collects data from machines named `mars` and `pluto`:

```
llacctmrg -h mars pluto
```

The following example collects data from the machine named `mars` and places the data in an existing directory called **merge**:

```
llacctmrg -h mars -d merge
```

Results

The following shows a sample system response from the **llacctmrg -h mars -d merge** command.

```
llacctmrg: History transferred successfully from mars (10080 bytes)
```

llcancel - Cancel a Submitted Job

Purpose
Cancels one or more jobs from the LoadLeveler queue.

Syntax
llcancel [**-?**] [**-H**] [**-v**] [**-q**] [**-u** *userlist*] [**-h** *hostlist*] [*joblist*]

Flags
-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

-u *userlist*
Is a blank-delimited list of users. When used with the **-h** option, only the user's jobs monitored on the machines in the *hostlist* are cancelled. When used alone, only the user's jobs monitored by the machine issuing the command are cancelled.

-h *hostlist*
Is a blank-delimited list of machine names. All jobs monitored on machines in this list are cancelled. When issued with the **-u** option, the *userlist* is used to further select jobs for cancellation.

joblist
Is a blank-delimited list of jobs of the form *host.jobid.stepid* where:

- *host* is the name of the machine to which the job was submitted (delimited by dot). The default is the local machine.
- *jobid* is the job ID assigned to the job when it was submitted using the **llsubmit** command. The *jobid* is required.
- *stepid* (delimited by dot) is the step ID assigned to the job when it was submitted using the **llsubmit** command. The default is to include all steps of the job.

The **-u** or **-h** flags override the *host.jobid.stepid* parameters.

When the **-h** flag is specified by a non-administrator, all jobs submitted from the machines in *hostlist* by the user issuing the command are cancelled.

When the **-h** flag is specified by an administrator, all jobs submitted by the administrator are canceled, unless the **-u** is also specified, in which case all jobs both submitted by users in *userlist* and monitored on machines in *hostlist* are cancelled.

Group administrators and class administrators are considered normal users unless they are also LoadLeveler administrators.

Description
When you issue **llcancel**, the command is sent to the negotiator. You should then use the **llq** command to verify your job was cancelled. A job state of RM (Removed) indicates the job was cancelled. A job state of RP (Remove Pending) indicates the job is in the process of being cancelled.

When cancelling a job from a submit-only machine, you must specify the machine name that scheduled the job. For example, if you submitted the job from machine A, a submit-only machine, and machine B, a scheduling machine, scheduled the job to run, you must specify machine B's name in the cancel command. If machine A and B are in different sub-domains, you must specify the fully-qualified name of the job in the cancel command. You can use the **llq -l** command to determine the fully-qualified name of the job.

Examples

This example cancels the job step 3 that is part of the job 18 that is scheduled by the machine named bronze:

```
llcancel bronze.18.3
```

This example cancels all the job steps that are a part of job 8 that are scheduled by the machine named gold.

```
llcancel gold.8
```

Results

The following shows a sample system response for the **llcancel gold.8** command.

```
llcancel: Cancel command has been sent to the central manager.
```

llclass - Query Class Information

Purpose
Returns information about classes.

Syntax
llclass [**-?**] [**-H**] [**-v**] [**-l**] [*classlist*]

Flags
-? Provides a short usage message.

-H Provides entended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-l Specifies that a long listing be generated for each class for which status is requested. If **-l** is *not* specified, then the standard listing is generated.

classlist
Is a blank-delimited list of classes for which you are requesting status. If no *classlist* is specified, all classes are queried.

If you have more than a few classes configured for LoadLeveler, consider redirecting the output to a file when you use the **-l** flag.

Examples
This example generates a long listing for classes named *silver* and *gold*:

```
llclass -l silver gold
```

Results
The Standard Listing: . The standard listing is generated when you do *not* specify **-l** with the **llclass** command. The following is sample output from the **llclass silver** command, where there are five `silver` classes configured in the cluster, with one `silver` class job currently running:

```
Name              MaxJobCPU     MaxProcCPU  Free   Max Description
                 d+hh:mm:ss     d+hh:mm:ss Slots Slots

silver           0+00:30:00     0+00:10:00     4     5 silver grade jobs
```

The standard listing includes the following fields:

MaxJobCPU
The CPU limit for all the processes in a job of this class. For a parallel job, this is the CPU limit for all processes in a task.

MaxProcCPU
The CPU limit for processes in this class.

Free Slots
The number of free slots (available classes) on this machine.

Max Slots
The total number of slots (configured classes) on this cluster.

Description
The description of this class.

The Long Listing: The long listing is generated when you specify the **-l** option on the **llclass** command. The following is sample output from the **llclass -l silver** command, where there are five `silver` classes configured in the cluster, with one `silver` class job currently running:

```
=============== Class silver ===============
                Name: silver
            Priority: 50
       Exclude_Users: user1
      Exclude_Groups: 85ba
               Admin: loadl:brownap:alice
           NQS_class: F
          NQS_submit:
           NQS_query:
      Max_processors: 1
             Maxjobs: 15
Resource_requirement: spice2g6(2)
       Class_comment: silver grade jobs
    Wall_clock_limit:   0+02:00:00,   0+01:00:00
       Job_cpu_limit:   0+00:59:59,   0+00:29:29
           Cpu_limit:   0+00:30:00,   0+00:10:00
          Data_limit: -1, -1
          Core_limit: -1, -1
          File_limit: -1, -1
         Stack_limit: -1, -1
           Rss_limit: -1, -1
                Nice: 15
                Free: 13
             Maximum: 21
```

The long listing includes these fields:

Name The name of the class

Priority
The system priority of this class relative to other classes.

Exclude_Users
Users who are not permitted to submit jobs of this class.

Exclude_Groups
Groups who are not allowed to submit jobs of this class.

Admin
The list of administrators of this class.

NQS_class
Indicates whether this class is a gateway for an NQS system.

NQS_submit
The NQS queue where the job will be submitted.

NQS_query
The NQS queues to query where the job has been dispatched.

Max_processors
The maximum number of processors than can be used for parallel jobs.

Max_jobs
The maximum number of jobs the class can run at any time.

Resource_requirement
Default consumable resource requirements for jobs of this class.

Class_comment
The text supplied by the administrator describing this class.

Wall_clock_limit
The hard and soft wall clock limits (the elapsed time for which the job can run).

Job_cpu_limit
The hard and soft CPU limits for all processes in a job of this class.

Cpu_limit
The hard and soft CPU limits for all processes in this class.

Data_limit
The hard and soft limits for the data area used for processes in this class.

Core_limit
The hard and soft core size limits.

File_limit
The hard and soft file size limits.

Stack_limit
The hard and soft stack size limits.

Rss_limit
The hard and soft rss size limits.

Nice The *nice* value of jobs in this class.

Free The number of classes available to new jobs.

Maximum
The total number of configured classes in this cluster.

Related Information

Each machine periodically updates the central manager with a snapshot of its environment. Since the information returned by **llclass** is a collection of these snapshots, all taken at varying times, the total picture may not be completely consistent.

llctl - Control LoadLeveler Daemons

Purpose
Controls LoadLeveler daemons on all members of the LoadLeveler cluster.

Syntax
llctl [**-?**] [**-H**] [**-v**] [**-q**] [**-g** | **-h** *host*] [*keyword*]

Flags
-? Provides a short usage message.

-H Provides entended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

-g Indicates that the command applies globally to all machines in the administration file.

-h *host*
Indicates that the command applies to only the *host* machine in the LoadLeveler cluster. If neither **-h** nor **-g** is specified, the default is the machine on which the **llctl** command is issued.

keyword
Must be specified after all flags and can be the following:

purge *list_of_machines*
Forces a schedd to delete any queued transaction to the machines in the *list_of_machines*. If all jobs on the listed machines have completed, and there are no messages pending to that machine, this option is not necessary.

This option is intended for recovery and cleanup after a machine has permanently crashed or was inadvertantly removed from the LoadLeveler cluster before all activity on it was quiesced. Do not use this option unless the specified *list_of_machines* are guaranteed not to return to the LoadLeveler cluster.

If you need to return the machine to the cluster later, you must clear all files from the spool and execute directory of the machine which was deleted.

capture *eventname*
Captures accounting data for all jobs running on the designated machines. *eventname* is the name you associate with the data, and must be a character string containing no blanks. For more information, see “Collecting Job Resource Data Based on Events” on page 154.

drain [schedd|startd [*classlist* **|allclasses]]**
When you issue **drain** with no options, the following happens: (1) no more LoadLeveler jobs can begin running on this machine, and (2) no more LoadLeveler jobs can be submitted through this machine. When you issue **drain schedd**, the following happens: (1) the schedd machine accepts no more LoadLeveler jobs for submission, (2) jobs in the Starting or Running state in the schedd queue are allowed to continue running, and (3) jobs in the Idle state in the schedd queue are drained, meaning they will not get dispatched. When you issue **drain startd**, the following happens: (1) the

startd machine accepts no more LoadLeveler jobs to be run, and (2) jobs already running on the startd machine are allowed to complete. When you issue **drain startd** *classlist*, the classes you specify which are available on the startd machine are drained (made unavailable). When you issue **drain startd allclasses**, all available classes on the startd machine are drained.

flush
Terminates running jobs on this machine and sends them back, in the Idle state, to the negotiator to await redispatch (provided **restart=yes** in the job command file). No new jobs are sent to this machine until **resume** is issued. Forces a checkpoint if jobs are enabled for checkpointing. However, the checkpoint gets cancelled if it does not complete within a five minute period.

purgeschedd
Requests that all jobs scheduled by the specified *host* machine be purged (removed). To use this keyword, you must first specify **schedd_fenced=true** in the machine stanza for this *host*. The -g option cannot be specified with this keyword. For more information, see "How Do I Recover Resources Allocated by a schedd Machine?" in the *IBM LoadLeveler for AIX: Diagnosis and Messages Guide*.

reconfig
Forces all daemons to reread the configuration files.

recycle
Stops all LoadLeveler daemons and restarts them.

resume [schedd|startd [*classlist* **|allclasses]]**
When you issue **resume** with no options, job submission and job execution on this machine is resumed. When you issue **resume schedd**, the schedd machine resumes the submission of jobs. When you issue **resume startd**, the startd machine resumes the execution of jobs. When you issue **resume startd** *classlist*, the startd machine resumes the execution of those job classes you specify which are also configured (defined on the machine). When you issue **resume startd allclasses**, the startd machine resumes the execution of all configured classes.

start
Starts the LoadLeveler daemons on the specified machine. You must have rsh privileges to start LoadLeveler on a remote machine.

stop
Stops the LoadLeveler daemons on the specified machine.

suspend
Suspends all jobs on this machine. This is not supported for parallel jobs.

version
Displays version and release data at the screen.

Description

This command sends a message to the master daemon on the target machine requesting that action be taken on the members of the LoadLeveler cluster. Note the following when using this command:

- After you make changes to the configuration files for a running cluster, be sure to issue **llctl reconfig**. This command causes the LoadLeveler daemons to reread the configuration files, and prevents problems that can occur when the LoadLeveler commands are using a new configuration while the daemons are using an old configuration.

- The **llctl drain startd** *classlist* command drains classes on the startd machine, and the startd daemon remains operational. If you reconfigure the daemon, the draining of classes remains in effect. However, if the startd goes down and is brought up again (either by the master daemon or by a LoadLeveler administrator), the startd daemon is configured according to the global or local configuration file in effect, and therefore the draining of classes is lost.

  Draining all the classes on a startd machine is *not* equivalent to draining the startd machine. When you drain all the classes, the startd enters the Idle state. When you drain the startd, the startd enters the Drained state. Similarly, resuming all the classes on a startd machine is *not* equivalent to resuming the startd machine.
- If a parallel job is running on a machine that receives the **llctl recycle** command, or the **llctl stop** and **llctl start** commands, the running job is terminated. You can restart the job by resubmitting the job or by specifying the **restart=yes** option in the job command file.

  If a serial job is running on a machine that receives the **llctl recycle** command, or the **llctl stop** and **llctl start** commands, the running job is terminated. You can restart the job by resubmitting the job or by enabling checkpointing and specifying the **restart=yes** option in the job command file.
- If you find that the **llctl -g** command (even if it is specified with additional options) is taking a long time to complete, you should consider using the SP **dsh** command to send **llctl** commands (omitting the **-g** flag) to multiple nodes in a parallel fashion. For more information on **dsh**, see *IBM RS/6000 Scalable POWERparallel Systems: Administration Guide* , (SH26-2486).
- When a node running a schedd daemon fails, resources that have been allocated to any of the jobs scheduled by that schedd are unavailable until the schedd is restarted. Administrators can, however, recover these resources by using the llctl command's **purgeschedd** keyword to purge (remove) all of the jobs scheduled by the schedd on the down node. The purgeschedd keyword can only work in conjunction with the **schedd_fenced** keyword, which causes the central manager to ignore (fence) the target schedd node. You must reconfigure the central manager so it can recognize this fence. To use the purgeschedd keyword:
  1. Recognize that a node running a schedd daemon is down, and that the node will be down long enough to necessitate that you recover the resources allocated to jobs scheduled by that schedd.
  2. Add the statement ″schedd_fenced = true″ to the failed node's administration file machine stanza.
  3. Reconfigure the central manager node, so that the central manager recognizes the fenced node.
  4. Invoke ″llctl -h *host* purgeschedd″ to purge all of the jobs scheduled by the schedd on the failed node.
  5. Remove all of the files in the LoadLeveler spool directory for that node. Once the failed node is working again, remove the ″schedd_fenced = true″ statement from the administration file, then reconfigure the central manager node.

Examples

This example stops LoadLeveler on the machine named *iron*:

```
llctl -h iron stop
```

This example starts the LoadLeveler daemons on all members of the LoadLeveler cluster, starting with the central manager, as defined in the machine stanzas of the administration file:

```
llctl -g start
```

This example causes the LoadLeveler daemons on machine *iron* to re-read the configuration files, which may contain new configuration information for the *iron* machine:

```
llctl -h iron reconfig
```

For the next three examples, suppose the classes *small*, *medium*, and *large* are available on the machine called *iron*.

This example drains the classes *medium* and *large* on the machine named *iron*.

```
llctl -h iron drain startd medium large
```

This example drains the classes *medium* and *large* on all machines.

```
llctl -g drain startd medium large
```

This example stops all the jobs on the system, then allows only jobs of a certain class (*medium*) to run.

```
llctl -g drain startd allclasses
llctl -g flush
llctl -g resume
llctl -g resume startd medium
```

This example resumes the classes *medium* and *large* on the machine named *iron*.

```
llctl -h iron resume startd medium large
```

This example illustrates how to capture accounting information on a work shift called *day* on the machine *iron*:

```
llctl -h iron capture day
```

You can capture accounting information on all the machines in the LoadLeveler cluster by using the **-g** option, or you can collect accounting information on the local machine by simply issuing the following:

```
llctl capture day
```

Capturing information on the local machine is the default. For more information, see “Collecting Job Resource Data Based on Events” on page 154.

Assume the machine *earth* has crashed while running jobs. Its hard disk needs to be replaced. You try to cancel the jobs that were running on that machine. The schedd marks the job Remove Pending until it gets confirmation from *earth* that the jobs were removed. Since *earth* will be reinstalled, you need to inform schedd that it should not wait for confirmation.

Assume the schedd is named *mars*, and the running jobs are named *mars.1.0* and *mars.1.1*. First you want to tell the negotiator to remove the jobs:

```
llcancel  mars.1.0
llcancel  mars.1.1
```

Next, tell the schedd not to wait for confirmation from *earth* before marking the jobs removed:

```
llctl -h mars purge earth
```

Results

The following shows the result of the **llctl -h mars purge earth** command:

```
llctl: Sent purge command to host mars
```

lldcegrpmaint - LoadLeveler DCE group Maintenance Utility

Purpose

This command extracts the names of the DCE groups associated with the DCE_ADMIN_GROUP and DCE_SERVICES_GROUP keywords from the LoadLeveler configuration file. It will create these groups if they do not already exist. This command also adds the DCE principal names of the LoadLeveler daemons to the group specified by the DCE_SERVICES-GROUP keyword.

Syntax

lldcegrpmaint [**-?**] [**-H**] [**-v**] *config_pathname admin_pathname*

Flags

-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

config_pathname
Pathname of the LoadLeveler configuration file.

admin_pathname
Pathname of the LoadLeveler administration file.

Description

The lldcegrpmaint command is available to DCE administrators who have logged in to DCE as **cell_admin**. The command performs the following functions:

1. Extracts the names of the DCE groups associated with the DCE_ADMIN_GROUP and DCE_SERVICES_GROUP keywords from the LoadLeveler global configuration file. These groups are known generically as the LoadL-admin group and the LoadL-services group. The LoadL-admin group contains the DCE principal names of users who have administrative authority for LoadLeveler. The LoadL-services group contains the DCE principal names of all the LoadLeveler daemons which run in the current LoadLeveler cluster. The lldcegrpmaint command will create these groups if they do not already exist.
2. Populates the LoadL-services group with the DCE principal names of the LoadLeveler daemons. These names are derived from the DCE hostnames associated with the dce_host_name keyword in the LoadLeveler administration file, and LoadLeveler related information defined in the /usr/lpp/ssp/config/spsec_defaults file. In order for this step to work, the machine stanzas in the administration file must contain the DCE hostnames of the all the machines in the LoadLeveler cluster. The llextSDR command can be used to retrieve the DCE hostnames.

Before running the lldcegrpmaint command, a DCE administrator should make sure that basic DCE Security setup steps have been performed. If SMIT panels are used, the steps under the ″RS/6000 SP Security″ panel should be performed in sequence (from top to bottom) to properly update the DCE Registry. This measure is important for LoadLeveler, and for any other function that exploits DCE Security on the SP. For the purposes of the lldcegrpmaint command, the important actions are: (1) ″Create dcehostnames″ and (2) ″Configure SP Trusted Services to use DCE Authentication.″

Note: lldcegrpmaint does not add the names associated with the LOADL_ADMIN keyword in the configuration file to the LoadL-admin group. It is the administrator's responsibility to add appropriate DCE principals to this group.

Examples

In this example, it is assumed that the DCE cell name is /.../c163.ppd.pok.ibm.com and that LoadLeveler configuration and administration files are named /u/loadl/LoadL_config and /u/loadl/LoadL_admin, respectively, and contain the statements:

```
DCE_ENABLEMENT=TRUE
DCE_ADMIN_GROUP=LoadL-admin4
DCE_SERVICES_GROUP=LoadL-services4
```

and

```
c163n02.ppd.pok.ibm.com:  type = machine central_manager = true
machine_mode = general
schedd_host = true
dce_host_name = c163n02.ppd.pok.ibm.com

c163n03.ppd.pok.ibm.com:  type = machine central_manager = false
machine_mode = general
schedd_host = true
dce_host_name = c163n03.ppd.pok.ibm.com
```

It is also assumed that there is no override specification in the file /spdata/sys1/spsec/spsec_overrides and that the file /usr/lpp/ssp/config/spsec_defaults contains the following:

```
SERVICE:LoadL/Master:kw:root:system
SERVICE:LoadL/Negotiator:kw:root:system
SERVICE:LoadL/Schedd:kw:root:system
SERVICE:LoadL/Startd:kw:root:system
SERVICE:LoadL/Starter:kw:root:system
SERVICE:LoadL/Kbdd:kw:root:system
SERVICE:LoadL/GSmonitor:kw:root:system
```

Executing the command:

```
lldcegrpmaint /u/loadl/LoadL_config /u/loadl/LoadL_admin
```

results in:

1. The creation of the DCE groups:

   ```
   /.../c163.ppd.pok.ibm.com/LoadL-admin4
   /.../c163.ppd.pok.ibm.com/LoadL-services4
   ```

2. The population of the DCE group LoadL-services4 with the DCE principals:

   ```
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Master
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Negotiator
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Schedd
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Startd
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Starter
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Kbdd
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/GSmonitor
   /.../c163.ppd.pok.ibm.com/LoadL/c163n02.ppd.pok.ibm.com/Master
   /.../c163.ppd.pok.ibm.com/LoadL/c163n03.ppd.pok.ibm.com/Negotiator
   /.../c163.ppd.pok.ibm.com/LoadL/c163n03.ppd.pok.ibm.com/Schedd
   /.../c163.ppd.pok.ibm.com/LoadL/c163n03.ppd.pok.ibm.com/Startd
   /.../c163.ppd.pok.ibm.com/LoadL/c163n03.ppd.pok.ibm.com/Starter
   /.../c163.ppd.pok.ibm.com/LoadL/c163n03.ppd.pok.ibm.com/Kbdd
   /.../c163.ppd.pok.ibm.com/LoadL/c163n03.ppd.pok.ibm.com/GSmonitor
   ```

llextSDR - Extract adapter information from the SDR

Purpose

Extracts adapter information from the system data repository (SDR) and creates adapter and machine stanzas for each node in an RS/6000 SP partition. You can use the information in these stanzas in the LoadLeveler administration file. This command writes the stanzas to standard output.

Syntax

llextSDR [**-?**] [**-H**] [**-v**] [**-a** *adapter*]

Flags

-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-a *adapter*
Specifies that the interface name of the given *adapter* on each node is used as the label (machine stanza name) of the generated machine stanza. If you do not specify an *adapter*, the label used is the **initial_hostname** field of the Node class in the SDR.

Description

In the SDR, the Node class contains an entry for each node in the SP partition. The Adapter class contains an entry for each adapter configured on a node. This command extracts the information in the Adapter class and creates an adapter stanza. This command also creates a machine stanza which identifies the node and the adapters attached to the node. The generated machine stanza also includes the **spacct_excluse_enable** keyword, whose value is obtained from the spacct_excluse_enable attribute in the SP class of the SDR. For more information on adapter stanzas, see “Step 5: Specify Adapter Stanzas” on page 95. For more information on machine stanzas, see “Step 1: Specify Machine Stanzas” on page 75.

The partition for which information is extracted is either the default partition or that specified with the SP_NAME environment variable. For the control workstation, the default partition is the default system partition. For an SP node, the default partition is the partition to which the node belongs.

You must issue this command on a machine with the ssp.clients file set installed. If you issue this command from a non-SP workstation, you must set SP_NAME to the IP address of the appropriate SDR instance for the partition.

Examples

The following example creates adapter and machine stanzas for all nodes in a partition:

```
llextSDR
```

The following example creates machine stanzas with each node’s css0 interface name as the label:

```
llextSDR -a css0
```

Results

You may need to alter or add information to the stanzas produced by this command when you incorporate the stanzas into the administration file. For example,

administrators may want to have each **network_type** field use a value that reflects the type of nodes installed on the network. Users will need to know the values used for **network_type** so that they can specify an appropriate value in their job command files.

Also, the output of this command includes fully-qualified machine names. If your existing administration file uses short names, you may need to change either the command output or your existing administration file so that you use either all fully-qualified names or all short names.

This is sample output for the **llextSDR** command, where the default partition is c187s. This sample shows machine and adapter stanzas for three of the nodes in a 16-node partition.

```
#llextSDR: System Partition = "c187s" on Thu Sep 30 10:15:47 1999

c187n16.ppd.pok.ibm.com: type = machine
adapter_stanzas = c187sn16.ppd.pok.ibm.com c187n16.ppd.pok.ibm.com
spacct_excluse_enable = true
dce_host_name = c187n16.ppd.pok.ibm.com
alias = c187sn16.ppd.pok.ibm.com

c187sn16.ppd.pok.ibm.com: type = adapter
adapter_name = css0
network_type = switch
interface_address = 9.114.45.144
interface_name = c187sn16.ppd.pok.ibm.com
switch_node_number = 15
css_type = SP_Switch_MX_Adapter

c187n16.ppd.pok.ibm.com: type = adapter
adapter_name = en0
network_type = ethernet
interface_address = 9.114.45.80
interface_name = c187n16.ppd.pok.ibm.com

c187n14.ppd.pok.ibm.com: type = machine
adapter_stanzas = c187sn14.ppd.pok.ibm.com c187n14.ppd.pok.ibm.com
spacct_excluse_enable = true
dce_host_name = c187n14.ppd.pok.ibm.com
alias = c187sn14.ppd.pok.ibm.com

c187sn14.ppd.pok.ibm.com: type = adapter
adapter_name = css0
network_type = switch
interface_address = 9.114.45.142
interface_name = c187sn14.ppd.pok.ibm.com
switch_node_number = 13
css_type = SP_Switch_Adapter

c187n14.ppd.pok.ibm.com: type = adapter
adapter_name = en0
network_type = ethernet
interface_address = 9.114.45.78
interface_name = c187n14.ppd.pok.ibm.com
     .
     .
     .

c187n01.ppd.pok.ibm.com: type = machine
adapter_stanzas = c187sn01.ppd.pok.ibm.com c187n01.ppd.pok.ibm.com
spacct_excluse_enable = true
dce_host_name = c187n01.ppd.pok.ibm.com
alias = c187sn01.ppd.pok.ibm.com
```

```
c187sn01.ppd.pok.ibm.com: type = adapter
adapter_name = css0
network_type = switch
interface_address = 9.114.45.129
interface_name = c187sn01.ppd.pok.ibm.com
switch_node_number = 0
css_type = SP_Switch_MX_Adapter

c187n01.ppd.pok.ibm.com: type = adapter
adapter_name = en0
network_type = ethernet
interface_address = 9.114.45.65
interface_name = c187n01.ppd.pok.ibm.com
```

The following shows sample output for the **llextSDR -a css0** command for a single node:

```
k10sn09.ppd.pok.ibm.com: type = machine
 adapter_stanzas = k10sn09.ppd.pok.ibm.com k10n09.ppd.pok.ibm.com
 spacct_excluse_enable = true

k10sn09.ppd.pok.ibm.com: type = adapter
 adapter_name = css0
 network_type = switch
 interface_address = 9.114.51.137
 interface_name = k10sn09.ppd.pok.ibm.com
 switch_node_number = 8
 css_type = SP_Switch_MX_Adapter

k10n09.ppd.pok.ibm.com: type = adapter
 adapter_name = en0
 network_type = ethernet
 interface_address = 9.114.51,73
 interface_name = k10n09.ppd.pok.ibm.com
```

llfavorjob - Reorder System Queue by Job

Purpose
Sets specified jobs to a higher system priority than all jobs that are not favored. This command also *unfavors* previously favored job(s), restoring the original priority, when you specify the **-u** flag.

Syntax
llfavorjob [**-?**] [**-H**] [**-v**] [**-q**] [**-u**] *joblist*

Flags
-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

-u Unfavors previously favored jobs, requeuing them according to their original priority levels.

joblist
Is a blank-delimited list of jobs of the form *host.jobid.stepid* where:

- *host* is the name of the machine to which the job was submitted (delimited by dot). The default is the local machine.
- *jobid* is the job ID assigned to the job by LoadLeveler when it was submitted using the **llsubmit** command. *jobid* is required.
- *stepid* (delimited by dot) Is the job step ID assigned to the job by LoadLeveler when it was submitted using the **llsubmit** command. The default is to include all members of the job.

Description
If this command is issued against jobs that are already running, it has no effect. If the job vacates, however, and returns to the queue, the job gets re-ordered with the new priority.

If more than one job is affected by this command, then the jobs are ordered by the **sysprio** expression and are scanned before the not favored jobs. However, favored jobs which do not match the job requirements with available machines may run after not favored jobs. This command remains in effect until reversed with the **-u** option.

Examples
This example assigns jobs 12.4 on the machine *iron* and 8.2 on *zinc* the highest priorities in the system, with the jobs ordered by the **sysprio** expression:

```
llfavorjob iron.12.4 zinc.8.2
```

This example unfavors jobs 12.4 on the machine *iron* and 8.2 on the machine *zinc*:

```
llfavorjob -u iron.12.4 zinc.8.2
```

llfavoruser - Reorder System Queue by User

Purpose
Sets a user's job(s) to the highest priority in the system, regardless of the current setting of the job priority. Jobs already running are not affected. This command also *unfavors* the user's job(s), restoring the original priority, when you specify the **-u** flag.

Syntax
llfavoruser [**-?**] [**-H**] [**-v**] [**-q**] [**-u**] *userlist*

Flags
-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

-u Unfavors previously favored users, reordering their job(s) according to their original priority level(s). If **-u** is **not** specified, the user's job(s) are favored.

userlist
Is a blank-delimited list of users whose jobs are given the highest priority. If **-u** is specified, *userlist* jobs are *unfavored*.

Description
This command affects your current and future jobs until you remove the favor.

When the central manager daemon is restarted, any favor applied to users is revoked.

The user's jobs still remain ordered by user priority (which may cause jobs for the user to swap **sysprio**). If more than one user is affected by this command, the jobs of favored users are ordered by **sysprio** and are scanned before the jobs of not favored users. However, jobs of favored users which do not match job requirements with available machines may run after jobs of not favored users.

Examples
This example grants highest priority to all queued jobs submitted by users `ellen` and `fred` according to the **sysprio** expression:

```
llfavoruser ellen fred
```

This example unfavors all queued jobs submitted by users `ellen` and `fred`:

```
llfavoruser -u ellen fred
```

llhold - Hold or Release a Submitted Job

Purpose

Places jobs in user hold or system hold and releases jobs from both types of hold. Users can only move their own jobs into and out of user hold. Only LoadLeveler administrators can move jobs into and release them from system hold.

Syntax

llhold [**-?**] [**-H**] [**-v**] [**-q**] [**-s**] [**-r**] [**-u** *userlist*] [**-h** *hostlist*] [*joblist*]

Flags

-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

-s Puts job(s) in system hold. Only a LoadLeveler administrator can use this option.

If neither **-s** nor **-r** is specified, LoadLeveler puts the job(s) in user hold.

-r Releases a job from hold. A job in user hold is released unless it is also in system hold, where it remains. A job in system hold is released unless it is also in user hold, where it remains.

Only a LoadLeveler administrator can release jobs from system hold. Only an administrator or the owner of a job can release it from user hold.

If neither **-s** nor **-r** is specified, LoadLeveler puts the job(s) in user hold.

-u *userlist*
Is a blank-delimited list of users. When used with the **-h** option, only the user's jobs monitored on the machines in the *hostlist* are held or released. When used alone, only the user's jobs monitored on the schedd machine are held or released.

-h *hostlist*
Is a blank-delimited list of machine names. All jobs monitored on machines in this list are held or released. When issued with the **-u** option, the *userlist* is used to further select jobs for holding or releasing.

When issued by a non-administrator, this option only acts upon jobs that user has submitted to the machines in *hostlist*.

When issued by an administrator, all jobs monitored on the machines are acted upon unless the **-u** option is also used. In that case, the *userlist* is also part of the selection process, and only jobs both submitted by users in *userlist* and monitored on the machines in the *hostlist* are acted upon.

joblist
Is a blank-delimited list of jobs of the form *host.jobid.stepid* where:

- *host* is the name of the machine to which the job was submitted (delimited by dot). The default is the local machine.

  If the job was submitted from a submit-only machine, this is the name of the schedd machine that sent the job to the negotiator.

- *jobid* is the job ID assigned to the job when it was submitted using the **llsubmit** command. *jobid* is required.
- *stepid* (delimited by dot) is the step ID assigned to the job by LoadLeveler when it was submitted using the **llsubmit** command. The default is to include all steps of the job.

Description

This command does not affect a job step that is running unless the job step attempts to enter the Idle state. At this point, the job step is placed in the Hold state.

To ensure a job is released from both system hold and user hold, the administrator must issue the command with **-r** specified to release it from system hold. The administrator or the submitting user can reissue the command to release the job from user hold.

This command will fail if:

- a non-administrator attempts to move a job into or out of system hold.
- a non-administrator attempts to move a job submitted by someone else into or out of user hold.

Examples

This example places job 23, job step 0 and job 19, job step 1 on hold:

```
llhold 23.0 19.1
```

This example releases job 23, job step 0, job 19, job step 1, and job 20, job step 3 fron a hold state:

```
llhold -r 23.0 19.1 20.3
```

This example places all jobs from users abe, barbara, and carol2 in system hold:

```
llhold -s -u abe barbara carol2
```

This example releases from a hold state all jobs on machines bronze, iron, and steel:

```
llhold -r -h bronze iron steel
```

This example releases from a hold state all jobs on machines bronze, iron, and steel that smith submitted:

```
llhold -r -u smith -h bronze iron steel
```

Results

The following shows a sample system response for the **llhold -r -h bronze** command:

```
llhold: Hold command has been sent to the central manager.
```

llinit - Initialize Machines in the LoadLeveler Cluster

Purpose
Initializes a new machine as a member of the LoadLeveler hardware resource cluster

Syntax
llinit [**-?**] [**-H**] [**-q**] [**-prompt**] [**-local** *pathname*] [**-release** *pathname*] [**-cm** *machine*] [**-debug**]

Flags
-? Provides a short usage message.

-H Provides extended help information.

-q Specifies quiet mode: print no messages other than error messages.

-prompt
Prompts or leads you through a set of questions that help you to complete the **llinit** command.

-local *pathname*
Where *pathname* is the local directory on which to create the spool, execute, and log sub-directories. The default, if this flag is not used, is the home directory.

There must be a unique local directory for each LoadLeveler cluster member.

-release *pathname*
Where *pathname* is the release directory, where the LoadLeveler bin, lib, man, include, and samples subdirectories are located. The default, if this flag is not used, is the **/usr/lpp/LoadL/full** directory.

-cm *machine*
Where *machine* is the central manager machine, where the negotiator daemon runs.

-debug
Displays a large amount of messages, tracing the path through **llinit** during execution. This is intended for debugging purposes only.

Description
This command runs once on each machine during the installation process. It must be run by the user ID you have defined as the LoadLeveler user ID. The log, spool, and execute directories are created with the correct modes and ownerships. The LoadLeveler configuration and administration files, **LoadL_config** and **LoadL_admin**, respectively, are copied from LoadLeveler's release directory to LoadLeveler's home directory. The local configuration file, **LoadL_config.local**, is copied from LoadLeveler's release directory to LoadLeveler's local directory.

llinit initializes a new machine as a member of the LoadLeveler resource cluster by doing the following:

- Creates the following LoadLeveler subdirectories with the given permissions:
  spool subdirectory, with permissions set to 700.
  execute subdirectory, with permissions set to 1777.
  log subdirectory, with permissions set to 775.
- Copies the **LoadL_config** and **LoadL_admin** files from the release directory samples subdirectory into the loadl home directory.

- Copies the **LoadL_config.local** file from the release directory samples subdirectory into the local directory.
- Creates symbolic links from the loadl home directory to the spool, execute, and log subdirectories and the **LoadL_config.local** file in the local directory (if home and local directories are not identical).
- Creates symbolic links from the home directory to the bin, lib, man, samples, and include subdirectories in the release directory.
- Updates the **LoadL_config** with the release directory name.
- Updates the **LoadL_admin** with the central manager machine name.

Before running **llinit** ensure that your HOME environment variable is set to LoadLeveler's home directory. To run llinit you must have:
- Write privileges in the LoadLeveler home directory
- Write privileges in the LoadLeveler release directory
- Write privileges in the LoadLeveler local directory.

Examples

The following example initializes a machine, assigning **/var/loadl** as the local directory, **/usr/lpp/LoadL/full** as the release directory, and the machine named **bronze** as the central manager.

```
llinit -local /var/loadl -release /usr/lpp/LoadL/full -cm bronze
```

Results

The command:

```
llinit -local /home/ll_admin -release /usr/lpp/LoadL/full -cm mars
```

will yield the following output:

```
llinit: creating directory "/home/ll_admin/spool"
llinit: creating directory "/home/ll_admin/log"
llinit: creating directory "/home/ll_admin/execute"
llinit: set permission "700" on "/home/ll_admin/spool"
llinit: set permission "775" on "/home/ll_admin/log"
llinit: set permission "1777" on "/home/ll_admin/execute"
llinit: creating file "/home/ll_admin/LoadL_admin"
llinit: creating file "/home/ll_admin/LoadL_config"
llinit: creating file "/home/ll_admin/LoadL_config.local"
llinit: editing file /home/ll_admin/LoadL_config
llinit: editing file /home/ll_admin/LoadL_admin
llinit: creating symbolic link "/home/ll_admin/bin -> /usr/lpp/LoadL/full/bin"
llinit: creating symbolic link "/home/ll_admin/lib -> /usr/lpp/LoadL/full/lib"
llinit: creating symbolic link "/home/ll_admin/man -> /usr/lpp/LoadL/full/man"
llinit: creating symbolic link "/home/ll_admin/samples -> /usr/lpp/LoadL/full/samples"
llinit: creating symbolic link "/home/ll_admin/include -> /usr/lpp/LoadL/full/include"
llinit: program complete.
```

llprio - Change the User Priority of Submitted Job Steps

Purpose

Changes the user priority of one or more job steps in the LoadLeveler queue. You can adjust the priority by supplying a **+** (plus) or **−** (minus) immediately followed by an *integer* value. **llprio** does not affect a job step that is running, even if its priority is lower than other jobs steps, unless the job step goes into the Idle state.

Syntax

llprio [**-?**] [**-H**] [**-v**] [**-q**] [**+***integer* | **−***integer* | **-p** *priority*] *joblist*

Flags

-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

+ | − *integer*
Operates on the current priority of the job step, making it higher (closer to execution) or lower (further from execution) by adding or subtracting the value of *integer*.

-p *priority*
Is the new absolute value for priority. The valid range is 0–100 (inclusive) where 0 is the lowest possible priority and 100 is highest.

joblist
Is a blank-delimited list of jobs of the form *host.jobid.stepid* where:

- *host* is the name of the machine to which the job step was submitted (delimited by dot). The default is the local machine.

  If the job step was submitted from a submit-only machine, this is the name of the machine where the schedd daemon that sent the job to the negotiator resides.
- *jobid* is the job ID assigned to the job when it was submitted using the **llsubmit** command. *jobid* is required.
- *stepid* (delimited by dot) is the job step ID assigned to the job when it was submitted using the **llsubmit** command.

Description

The user priority of a job step ranges from 0 to 100 inclusively, with higher numbers corresponding to greater priority. The default priority is 50. Only the owner of a job step or the LoadLeveler administrator can change the priority of that job step. Note that the priority is not the UNIX *nice* priority.

Priority changes resulting in a value less than 0 become 0.

Priority changes resulting in a value greater than 100 become 100.

Any change to a job step's priority applied by a user is relative only to *that user's other job steps* in the same class. If you have three job steps enqueued, you can reorder those three job steps with **llprio** but the result does not affect job steps submitted by other users, regardless of their priority and position in the queue.

See "Setting and Changing the Priority of a Job" on page 28 for more information.

Examples

This example raises the priority of job 4, job step 1 submitted to machine bronze by a value of 25:

```
llprio +25 bronze.4.1
```

This example sets the priority of job 18, job step 4 submitted to machine silver to 100, the highest possible value:

```
llprio -p 100 silver.18.4
```

Results

The following shows a sample system response for the **llprio -p 100 silver.18.4** command:

```
llprio: Priority command has been sent to the central manager.
```

llq - Query Job Status

Purpose
Returns information about jobs that have been dispatched.

Syntax
llq [**-?**] [**-H**] [**-v**] [**-x**] [**-s**] [**-l**] [*joblist*] [**-u** *userlist*] [**-h** *hostlist*] [**-c** *classlist*] [**-f** *category_list*] [**-r** *category_list*]

Flags
-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-x Provides extended information about the selected job. If the **-x** flag is used with the **-r**, **-s**, or **-f** flag, an error message is generated.

CPU usage and other resource consumption information on active jobs can only be reported using the **-x** flag if the LoadLeveler administrator has enabled it by specifying A_ON and A_DETAIL for the ACCT keyword in the LoadLeveler configuration file.

Normally, **llq** connects with the central manager to obtain job information. When you specify **-x**, **llq** connects to the schedd machine that received the specified job to get extended job information.

When specified without **-l**, CPU usage for active jobs is reported in the short format. Using **-x** can produce a very long report and can cause excess network traffic.

-s Provides information on why a selected list of jobs remain in the NotQueued, Idle or Deferred state. Along with this flag, users must specify a list of jobs. The user can also optionally supply a list of machines to be considered when determining why the job(s) cannot run. If a list of machines is not provided, the default is the list of machines in the LoadLeveler cluster. For each job, **llq** determines why the job remains in one of the given states instead of Running.

-l Specifies that a long listing be generated for each job for which status is requested. Fields included in the long listing are shown in “Results” on page 195.

If **-l** is *not* specified, then the standard listing is generated as shown in “Results” on page 195.

joblist
Is a blank-delimited list of jobs of the form *host.jobid.stepid* where:

- *host* is the name of the machine to which the job was submitted (delimited by dot). The default is the local machine.

  If the job was submitted from a submit-only machine, this is the name of the machine where the schedd daemon that sent the job to the negotiator resides.
- *jobid* is the job id assigned to the job when it was submitted using the **llsubmit** command.

- *stepid* (delimited by dot) Is the step id assigned to the job when it was submitted using the **llsubmit** command. The default is to include all members of the cluster.

-u *userlist*
Is a blank-delimited list of users. When used with the **-h** option, only the user's jobs monitored on the machines in the *hostlist* are queried. When used alone, only the user's jobs monitored on the schedd machine are queried.

-h *hostlist*
Is a blank-delimited list of machines. If the **-s** flag is not specified, all jobs monitored on machines in this list are queried. If the **-s** flag is specified, the list of machines is considered when determining why a job remains in Idle state. When issued with the **-u** option, the *userlist* is used to further select jobs for querying.

-c *classlist*
Is a blank-delimited list of classes. When used with **-h**, only those jobs monitored on the machines in the *hostlist* are queried.

-f *category_list*
Is a blank-delimited list of categories you want to query. Each category you specify must be preceded by a percent sign. The *category_list* cannot contain duplicate entries. This flag allows you to create a customized version of the standard **llq** listing. You cannot use this flag with the **-l** flag. The output fields produced by this flag all have a fixed length. The output is displayed in the order in which you specify the categories. *category_list* can be one or more of the following:

%a	Account number
%c	Class
%cc	Completion code
%dc	Completion date
%dd	Dispatch Date
%dh	Hold date
%dq	Queue date
%gl	LoadLeveler group
%gu	UNIX group
%h	Host (First hostname if more than one is allocated to the job)
%id	Step ID
%is	Virtual image size
%jn	Job name
%jt	Job type
%nh	Number of hosts allocated to the job
%o	Job owner
%p	User priority
%sn	Step name
%st	Status

-r *category_list*
Is a blank-delimited list of formats (categories) you want to query. Each category you specify must be preceded by a percent sign. The *category_list* cannot contain duplicate entries. This flag allows you to create a customized version of the standard **llq** listing. You cannot use this flag with the **-l** flag. The output produced by this flag is considered raw, in that the fields can be variable in length. Output fields are separated by an exclamation point (!). The output is displayed in the order in which you specify the formats. *category_list* can be one or more of the formats listed under the **-f** flag.

If the **-u** or **-h** options are not specified, and if no *jobid* is specified, then all jobs are queried.

The **-u** and **-h** options override the *jobid* parameters.

Examples

This example generates a long listing for job 8, job step 2 submitted to machine *gold*:

```
llq -l gold.8.2
```

This example generates a standard listing for all job steps of job name 12 submitted to the local machine:

```
llq 12
```

Results

In this section, the term “job step” refers to either a serial job step or a parallel task.

Standard Listing: The standard listing is generated when you do *not* specify the **-l** option with the **llq** command. The following is sample output from the **llq -h mars** command, where the machine `mars` has two jobs running and one job waiting:

```
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
mars.498.0               brownap     5/20 11:31 R  100 silver       mars
mars.499.0               brownap     5/20 11:31 R  50  No_Class     mars
mars.501.0               brownap     5/20 11:31 I  50  silver

3 job steps in queue, 1 waiting, 0 pending, 2 running, 0 held.
```

The standard listing includes the following fields:

Id job identifier presented in the format: *host.jobid.stepid*. When the llq command returns information about a job owned by a schedd in the same domain, then the domain of the hostname won’t appear in the output. However, when the llq command reports information about a job owned by a schedd in a different domain, the fully qualified hostname is always included. Due to space limitations, the host’s domain may be truncated to fit in the space allocated to the Id field. If the domain is truncated, a dash (-) will appear at the end to indicate that characters have been left out. To see the full job ID, run llq with the -l flag.

Owner
userid of the job submitter.

Submitted
date and time of job submission.

ST current job status (state). Job status can be:

- **C** Completed
- **CA** Cancelled
- **CP** Complete Pending
- **D** Deferred
- **H** User Hold
- **HS** User Hold and System Hold
- **I** Idle
- **NR** Not Run
- **NQ** Not Queued
- **P** Pending

R Running
RM Removed
RP Remove Pending
S System Hold
ST Starting
SX Submission Error
TX Terminated
V Vacated
VP Vacate Pending
X Rejected
XP Reject Pending

For a detailed explanation of job states, see “LoadLeveler Job States” on page 18.

PRI user priority of the job, where the values are defined with the **user_priority** keyword in the job command file or changed by the **llprio** command. See “llprio - Change the User Priority of Submitted Job Steps” on page 191

Class job class.

Running On
if running, the machine the job is running on. This is blank when the job is not running. For parallel jobs, only the first machine is shown.

Customized, Formatted Standard Listing: A customized and formatted standard listing is generated when you specify **llq** with the **-f** flag. The following is sample output from this command:

```
llq -f %id %c %dq %dd %gl %h
```

```
Step Id           Class      Queue Date  Disp. Date  LL Group   Running On
----------------- ---------- ----------- ----------- ---------- ---------------
ll6.2.0           No_Class   04/08 09:19 04/08 09:21 No_Group   ll6.pok.ibm.com
ll6.1.0           No_Class   04/08 09:19 04/08 09:21 No_Group   ll6.pok.ibm.com
ll6.3.0           No_Class   04/08 09:19 04/08 09:21 No_Group   ll5.pok.ibm.com

3 job steps in queue, 0 waiting, 0 pending, 3 running, 0 held
```

Customized, Unformatted Standard Listing: A customized and unformatted (raw) standard listing is generated when you specify **llq** with the **-r** flag. Output fields are separated by an exclamation point (!). The following is sample output from this command:

```
llq -r %id %c %dq %dd %gl %h
```

```
ll6.pok.ibm.com.2.0!No_Class!04/08 09:19!04/08 09:21!No_Group!ll6.pok.ibm.com
ll6.pok.ibm.com.1.0!No_Class!04/08 09:19!04/08 09:21!No_Group!ll6.pok.ibm.com
ll6.pok.ibm.com.3.0!No_Class!04/08 09:19!04/08 09:21!No_Group!ll5.pok.ibm.com
```

The Long Listing: The long listing is generated when you specify the **-l** option with the **llq** command. This section contains sample output for two **llq** commands: one querying a serial job and one querying a parallel job. Following the sample output is an explanation of all possible fields displayed by the **llq** command.

The following is sample output for the **llq -l** command for the serial job “**c163n12.ppd.pok.ibm.com.9**”:

```
=============== Job Step c163n12.ppd.pok.ibm.com.9.0 ===============
        Job Step Id: c163n12.ppd.pok.ibm.com.9.0
           Job Name: c163n12.ppd.pok.ibm.com.9
          Step Name: batch_job_1
  Structure Version: 9
              Owner: loadl
         Queue Date: Mon Jun 28 10:33:59 EDT 1999
             Status: Running
      Dispatch Time: Mon Jun 28 10:34:02 EDT 1999
    Completion Date:
    Completion Code:
      User Priority: 50
       user_sysprio: 0
      class_sysprio: 45
      group_sysprio: 0
    System Priority: -4042
          q_sysprio: -4042
      Notifications: Complete
 Virtual Image Size: 1 kilobytes
         Checkpoint:
            Restart: yes
     Hold Job Until:
                Cmd: batch1.cmd
               Args: arg_1 arg_2 arg_3
                Env:
                 In: /dev/null
                Out: job1.c163n12.9.0.out
                Err: job1.c163n12.9.0.err
Initial Working Dir: /test/loadl
          Dependency:
           Resources: spice3f5(2)
       Requirements: (Memory > 32) && (Arch == "R6000") && (OpSys == "AIX43")
        Preferences: (Memory > 128) && (Feature == "ESSL")
          Step Type: Serial
     Min Processors:
     Max Processors:
     Allocated Host: c163n12.ppd.pok.ibm.com
         Node Usage: shared
    Submitting Host: c163n12.ppd.pok.ibm.com
        Notify User: loadl@c163n12.ppd.pok.ibm.com
              Shell: /bin/ksh
  LoadLeveler Group: No_Group
              Class: small
     Cpu Hard Limit: 1800 seconds
     Cpu Soft Limit: 600 seconds
    Data Hard Limit: -1
    Data Soft Limit: -1
    Core Hard Limit: -1
    Core Soft Limit: -1
    File Hard Limit: -1
    File Soft Limit: -1
   Stack Hard Limit: -1
   Stack Soft Limit: -1
     Rss Hard Limit: -1
     Rss Soft Limit: -1
Step Cpu Hard Limit: 3599 seconds
Step Cpu Soft Limit: 1769 seconds
Wall Clk Hard Limit: 4000 seconds
Wall Clk Soft Limit: 3600 seconds
            Comment: Test batch job 1.
            Account: 99999
         Unix Group: loadl
   NQS Submit Queue:
   NQS Query Queues:
Negotiator Messages:
Adapter Requirement:
Step CPUs:
Step Virtual Memory:
Step Real Memory:
Step Adapter Memory:
```

The following is sample output for the **llq -l -x c163n12.6.0.0** command, where c163n12.6.0 is a parallel job.

```
***************** llq -l -x : PARALLEL JOB *****************************

=============== Job Step c163n12.ppd.pok.ibm.com.6.0 ===============
        Job Step Id: c163n12.ppd.pok.ibm.com.6.0
           Job Name: c163n12.ppd.pok.ibm.com.6
          Step Name: 0
  Structure Version: 9
              Owner: loadl
         Queue Date: Mon Jun 28 09:35:21 EDT 1999
             Status: Running
      Dispatch Time: Mon Jun 28 09:35:21 EDT 1999
    Completion Date:
    Completion Code:
      User Priority: 50
       user_sysprio: 0
      class_sysprio: 30
      group_sysprio: 0
    System Priority: 0
          q_sysprio: 0
      Notifications: Complete
 Virtual Image Size: 376 kilobytes
         Checkpoint:
            Restart: yes
     Hold Job Until:
                Env: MANPATH=/usr/local/man:/usr/share/man: LANG=en_US LOGIN= ...
                 In: /dev/null
                Out: poe5_1.c163n12.6.0.out
                Err: poe5_1.c163n12.6.0.err
Initial Working Dir: /test/loadl
         Dependency:
      Task_geometry:
          Resources:
          Step Type: General Parallel
         Node Usage: not_shared
    Submitting Host: c163n12.ppd.pok.ibm.com
        Notify User: loadl
              Shell: /bin/ksh
  LoadLeveler Group: No_Group
              Class: Parallel
     Cpu Hard Limit: 3600 seconds
     Cpu Soft Limit: 1200 seconds
    Data Hard Limit: -1
    Data Soft Limit: -1
    Core Hard Limit: -1
    Core Soft Limit: -1
    File Hard Limit: -1
    File Soft Limit: -1
   Stack Hard Limit: -1
   Stack Soft Limit: -1
     Rss Hard Limit: -1
     Rss Soft Limit: -1
Step Cpu Hard Limit: 5400 seconds
Step Cpu Soft Limit: 2400 seconds
Wall Clk Hard Limit: 6000 seconds
Wall Clk Soft Limit: 3600 seconds
            Comment:
            Account: 99999
         Unix Group: loadl
      DCE Principal: tvdfs
 User Space Windows: 8
   NQS Submit Queue:
   NQS Query Queues:
Negotiator Messages:
Adapter Requirement: (css0,LAPI,shared,US),(css0,MPI,shared,US)
```

```
--------------- Detail for c163n12.ppd.pok.ibm.com.6.0 ---------------
       Running Host: c163n12.ppd.pok.ibm.com
      Machine Speed: 1.000000
  Starter User Time:   0+00:00:00.200000
Starter System Time:   0+00:00:00.340000
 Starter Total Time:   0+00:00:00.540000
     Starter maxrss: 1720
      Starter ixrss: 11392
      Starter idrss: 13520
      Starter isrss: 0
     Starter minflt: 1352
     Starter majflt: 2
      Starter nswap: 0
    Starter inblock: 0
    Starter oublock: 0
     Starter msgsnd: 0
     Starter msgrcv: 0
   Starter nsignals: 1
      Starter nvcsw: 76
     Starter nivcsw: 27
     Step User Time:   0+00:00:12.0
   Step System Time:   0+00:00:00.830000
    Step Total Time:   0+00:00:12.830000
        Step maxrss: 1368
         Step ixrss: 15528
         Step idrss: 426068
         Step isrss: 0
        Step minflt: 5947
        Step majflt: 12
         Step nswap: 0
       Step inblock: 0
       Step oublock: 0
        Step msgsnd: 0
        Step msgrcv: 0
      Step nsignals: 322
         Step nvcsw: 771
         Step nivcsw: 591
      Step CPUs: 18
      Step Virtual Memory: 180 megabytes
      Step Real Memory: 90 megabytes
      Step Adapter Memory: 2097152 bytes
--------------------------------------------------------------------------------
Node
----

   Name           :
   Requirements   :
   Preferences    :
   Node minimum   : 2
   Node maximum   : 2
   Node actual    : 2
   Allocated Hosts : c163n12.ppd.pok.ibm.com:RUNNING:css0(1,LAPI,US,1M),
                     css0(2,MPI,US,1M),css0(3,LAPI,US,1M),css0(4,MPI,US,1M)
                   + c163n11.ppd.pok.ibm.com:RUNNING:css0(1,LAPI,US,1M),
                     css0(2,MPI,US,1M),css0(3,LAPI,US,1M),css0(4,MPI,US,1M)
   Master Task
   -----------

      Executable   : /bin/poe
      Exec Args    : /test/loadl/ivp_600 -euilib us -ilevel 6 -labelio yes -pmdlog yes
      Num Task Inst: 1
      Task Instance: c163n12:-1

   Task
   ----

      Num Task Inst: 4
      Task Instance: c163n12:0:css0(1,LAPI,US,1M),css0(2,MPI,US,1M)
      Task Instance: c163n12:1:css0(3,LAPI,US,1M),css0(4,MPI,US,1M)
      Task Instance: c163n11:2:css0(1,LAPI,US,1M),css0(2,MPI,US,1M)
      Task Instance: c163n11:3:css0(3,LAPI,US,1M),css0(4,MPI,US,1M)
```

The long listing includes these fields:

Job Step ID
The job step identifier.

Job Name
The name of the job.

Step Name
The name of the job step

Structure Version
An internal version identifier.

Owner
The userid of the user submitting the job.

Queue Date
The date and time that LoadLeveler received the job.

Status
The status (state) of the job. A job's status can be:
Cancelled
Completed
Complete Pending
Deferred
Idle
Not Queued
Not Run
Pending
Rejected
Reject Pending
Removed
Remove Pending
Running
Starting
Submission Error
System Hold
System and User Hold
Terminated
User Hold
Vacated
Vacate Pending

For a detailed explanation of these job states, see "LoadLeveler Job States" on page 18.

Dispatch Time
the time the job was dispatched.

Completion Date
date and time job completed or exited.

Completion Code
the status returned by the wait3 UNIX system call.

User Priority
The priority of the job, as specified by the user in the job command, or changed by the **llprio** command.

user_sysprio
The user system priority of the job, where the value is defined in the administration file.

class_sysprio
The class priority of the job, where the value is defined in the administration files.

group_sysprio
The group priority of the job, where the value is defined in the administration files.

System Priority
The overall system priority of the job, where the value is defined by the SYSPRIO expression in the configuration file.

q_sysprio
The adjusted system priority of the job (See "How Does a Job's Priority Affect Dispatching Order?" on page 28.)

Notifications
The notification status for the job, where:

always
indicates notification is sent through the mail for all four notification categories below.

complete
indicates notification is sent through the mail only when the job completes.

error
indicates notification is sent through the mail only when the job terminates abnormally.

never
indicates notification is never sent.

start
indicates notification is sent through the mail only when starting or restarting the job.

Virtual Image Size
of the executable that was submitted.

Checkpoint
checkpoint status (yes or no)

Restart
restart status (yes or no)

Hold Job Until
job is deferred until this date and time.

Cmd name of the executable that was submitted.

Args arguments that were passed to the executable.

Env environment variables to be set before executable runs. Appears only when the **-x** option is specified.

In file to be used for stdin.

Out file to be used for stdout.

Err The file to be used for stderr.

Init Working Directory
The directory from which the job is run. The relative directory from which the stdio files are accessed, if appropriate.

Dependency
Job dependencies as specified at job submission.

Requirements
Job requirements as specified at job submission.

Preferences
Job preferences as specified at job submission.

Task_geometry
Reflects the settings for the task_geometry keyword in the job command file.

Resources
Reflects the settings for the resources keyword in the job command file.

Blocking
Reflects the settings for the blocking keyword in the job command file.

Step Type
Type of job step (serial or parallel).

Min Processors
The minimum number of processors needed for this job.

Max Processors
The maximum number of processors needed for this job.

Allocated Hosts
The machines that have been allocated for this job.

Node Usage
A request that a node be shared or not shared; the user specifies this request while submitting the job.

Submitting Host
The name of the machine to which the job is submitted.

Notify User
The user to be notified by mail of a job's status.

Shell The shell to be used when the job runs.

LoadLeveler Group
The LoadLeveler group associated with the job.

Class The job's class as specified at job submission.

CPU Hard Limit
CPU hard limit as specified at job submission.

CPU Soft Limit
CPU soft limit as specified at job submission.

Data Hard Limit
Data hard limit as specified at job submission.

Data Soft Limit
Data soft limit as specified at job submission.

Core Hard Limit
Core hard limit as specified at job submission.

Core Soft Limit
Core soft limit as specified at job submission.

File Hard Limit
File hard limits as specified at job submission.

File Soft Limit
File soft limit as specified at job submission.

Stack Hard Limit
Stack hard limit as specified at job submission.

Stack Soft Limit
Stack soft limit as specified at job submission.

Rss Hard Limit
RSS hard limit as specified when job was submitted.

Rss Soft Limit
RSS soft limit as specified at job submission.

Job Cpu Hard Limit
Job CPU hard limit as specified at job submission.

Job Cpu Soft Limit
Job CPU soft limit as specified at job submission.

Wall Clock Hard Limit
Wall clock hard limit as specified at job submission.

Wall Clock Soft Limit
Wall clock soft limit as specified at job submission.

NQS Submit Queue
The name of the NQS pipe queue to which the NQS job will be routed.

NQS Query Queue
The NQS queue names you can use to monitor the job.

Comment
The comment specified by the comment keyword in the job command file.

Account
The account number specified in the job command file.

UNIX Group
The effective UNIX group name.

DCE Principal
The DCE pricipal name associated with the process that submitted the job to LoadLeveler.

User Space Windows
The number of switch adapter windows assigned to the job.

Negotiator Messages
Informational message(s) for jobs in the Idle or NotQueued state.

Adapter Requirement
Reflects the settings of the network keyword in the job command file.

Step CPUs
The total Consumable CPUs for the job step.

Step Virtual Memory
The total Consumable Virtual Memory for the job step.

Step Real Memory
The total Consumable Memory for the job step.

Step Adapter Memory
The total adapter pinned memory for the job step.

Other fields displayed when issuing **llq -x -l** are:

maxrss
maximum resident set size utilized.

ixrss size of the text segment of the jobs.

idrss size of the data segement of the jobs.

isrss Integral unshared stack used.

minflt # Page faults (re-claimed).

majflt # Page faults (I/O required).

nswap
times swapped out.

inblock
times file system performed input.

oublock
times file system performed output.

msgsnd
# of IPC messages sent.

msgrcv
# of IPC messages received.

nsignals
# of signals delivered.

nvcsw
# of context switches due to voluntarily giving up processor.

nivcsw
# of involuntary context switches.

Other fields displayed for parallel jobs are:

Allocated Hosts
allocated hostname information in the format *hostname:task status:adapter usage*. The *adapter usage* information is in the format *adapter name (adapter window ID,network protocol,mode, adapter window memory)*.

Task Instance
task instance information in the format *hostname:task ID:adapter usage*. The *adapter usage* information is in the format *adapter name (adapter window ID,network protocol,mode, adapter window memory)*.

llstatus - Query Machine Status

Purpose

Returns status information about machines in the LoadLeveler cluster. It does not provide status on any NQS machine.

Syntax

llstatus [**-?**] [**-H**][**-R**][**-F**] [**-v**] [**-l**] [**-f** *category_list*] [**-r** *category_list*] [*hostlist*]

Flags

-? Provides a short usage message.

-H Provides extended help information.

-R Lists all of the machine consumable resources associated with all of the machines in the LoadLeveler cluster (when specified alone). When a host list is specified, the option only displays machine consumable resources associated with the specified hosts. This option should not be used with any other option.

-F Lists all of the floating consumable resources associated with the LoadLeveler cluster. This option should not be used with any other option.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-l Specifies that a long listing be generated for each machine for which status is requested. If **-l** is *not* specified, the standard list, described below, is generated.

-f *category_list*

Is a blank-delimited list of categories you want to query. Each category you specify must be preceded by a percent sign. The *category_list* cannot contain duplicate entries. This flag allows you to create a customized version of the standard **llstatus** listing. The output fields produced by this flag all have a fixed length. The output is displayed in the order in which you specify the categories. *category_list* can be one or more of the following:

%a	Hardware architecture
%act	Number of jobs dispatched by the schedd on this machine
%cm	Custom Metric value
%cpu	Number of CPUs on this machine
%d	Available disk space in the LoadLeveler execute directory
%i	Number of seconds since last keyboard or mouse activity
%inq	Number of jobs in queue that were scheduled from this machine
%l	Berkeley one-minute load average
%m	Physical memory on this machine
%mt	Maximum number of tasks that can run simultaneously on this machine
%n	Machine name
%o	Operating system on this machine
%r	Number of jobs running on this machine
%sca	Availability of the schedd daemon
%scs	State of the schedd daemon
%sta	Availability of the startd daemon
%sts	State of the startd daemon
%v	Available swap space of this machine

-r *category_list*

Is a blank-delimited list of categories you want to query. Each category you specify must be preceded by a percent sign. The *category_list* cannot contain duplicate entries. This flag allows you to create a customized version of the

standard **llstatus** listing. The output produced by this flag is considered raw, in that the fields can be variable in length. The output is displayed in the order in which you specify the formats. Output fields are separated by an exclamation point (!). *category_list* can be one or more of the categories listed under the **-f** flag.

hostlist
Is a blank-delimited list of machines for which status is requested.

Description

If no *hostlist* is specified, all machines are queried.

If you have more than a few machines configured for LoadLeveler, consider redirecting the output to a file when using the **-l** flag.

Each machine periodically updates the central manager with a snapshot of its situation. Since the information returned by using **llstatus** is a collection of such snapshots, all taken at varying times, the total picture may not be completely consistent.

Examples

This example requests a long status listing for machines named silver and gold:

```
llstatus -l silver gold
```

Results

In this section, the term “job step” refers to either a serial job step or a parallel task.

The Standard Listing: The standard listing is generated when you do *not* specify the **-l** option with the **llstatus** command. The following is sample output from the **llstatus** command, where there are two nodes in the cluster.

```
Name                      Schedd  InQ Act Startd Run LdAvg Idle Arch      OpSys
k10n09.ppd.pok.ibm.com    Avail     3   1 Run      1 2.72     0 R6000     AIX43
k10n12.ppd.pok.ibm.com    Avail     0   0 Idle     0 0.00   365 R6000     AIX43

R6000/AIX43            2 machines   3 jobs   1 running
Total Machines         2 machines   3 jobs   1 running

The Central Manager is defined on k10n09.ppd.pok.ibm.com

All machines on the machine_list are present.
```

The standard listing includes the following fields:

Name hostname of the machine.

Schedd
state of the schedd daemon, which can be one of the following:
Down
Drned (Drained)
Drning (Draining)
Avail (Available)

For a detailed explanation of these states, see “The schedd Daemon” on page 14.

InQ number of job steps in the queue that were scheduled from this machine.

Act number of job steps that the schedd has dispatched.

Startd state of the startd daemon, which can be:
Busy

Down
Drned (Drained)
Drning (Draining)
Flush
Idle
None
Run (Running)
Suspnd (Suspend)

For a detailed explanation of these states, see “The startd Daemon” on page 15.

Run The number of initiators used to run LoadLeveler jobs. One initiator is used for each serial job step. One initiator is used for each task of a parallel job step.

LdAvg
Berkeley one-minute load average on this machine.

Idle The number of seconds since keyboard or mouse activity in a login session was detected. Highest number displayed is 9999.

Arch The hardware architecture of the machine as listed in the configuration file.

OpSys
The operating system on this machine.

Consumable Resources Listing: The **llstatus** command, issued with the **-R** option, generates a listing of all of the consumable resources associated with all of the machines in the LoadLeveler cluster. When a host list is specified, this option will only display resources associated with the specified hosts. The following is sample output from this command:

```
llstatus -R
```

```
Machine                       Consumable Resource(Available, Total)
----------------------------- -------------------------------------------------
c163n11.ppd.pok.ibm.com       ConsumableCpus(2,4) resource_1(26,30)
c163n12.ppd.pok.ibm.com       resource_1(10,15) res_2(15,24) spice2g6(13,13)
ll6.pok.ibm.com               spice2g6(3,6) spice3f5(10,12)
ll7.pok.ibm.com               res_2(10,10) res_3(0,24) spice3f5(4,12)
```

Floating Consumable Resources Listing: The **llstatus** command, issued with the **-F** option, generates a listing of all of the floating consumable resources associated with all of the machines in the LoadLeveler cluster. This option should not be specified with any other option. The following is sample output from this command:

```
llstatus -F
```

```
Floating Resource             Available     Total
----------------------------- ------------- ---------------
EDA_licenses                  20            29
Frame5                        15            20
WorkBench6                    5             7
XYZ_software                  6             6
```

Customized, Formatted Standard Listing: A customized and formatted standard listing is generated when you specify **llstatus** with the **-f** option. The following is sample output from this command:

```
llstatus -f %n %scs %inq %m %v %sts %l %o
```

```
Name             Schedd  InQ    Memory      FreeVMemory Startd  LdAvg  OpSys
ll5.pok.ibm.com  Avail   0      128         22708       Run     0.23   AIX43
ll6.pok.ibm.com  Avail   3      224         16732       Run     0.51   AIX43

R6000/AIX43                2 machines      3  jobs      3  running
Total Machines             2 machines      3  jobs      3  running

The Central Manager is defined on ll5.pok.ibm.com

All machines on the machine_list are present.
```

Customized, Unformatted Standard Listing: A customized and unformatted (raw) standard listing is generated when you specify **llstatus** with the **-r** flag. Output fields are separated by an exclamation point (!). The following is sample output from this command:

llstatus -r %n %scs %inq %m %v %sts %l %o

```
ll5.pok.ibm.com!Avail!0!128!22688!Running!0.14!AIX43
ll6.pok.ibm.com!Avail!3!224!16668!Running!0.37!AIX43
```

The Long Listing: The long listing is generated when you specify the **-l** option with the **llstatus** command. Following the sample output is an explanation of all possible fields displayed by the **llstatus** command.

The following is sample output from the **llstatus -l ll6** command:

```
================================================================================
Name                = ll6.pok.ibm.com
Machine             = ll6.pok.ibm.com
Arch                = R6000
OpSys               = AIX43
SYSPRIO             = (0 -  QDate)
MACHPRIO            = (0 -  LoadAvg)
VirtualMemory       = 16640
Disk                = 23000
KeyboardIdle        = 600
Tmp                 = 48868
LoadAvg             = 0.302991
ConfiguredClasses   = No_Class(2) osl(1) small(2) medium(1) POE(2)
AvailableClasses    = No_Class(0) osl(1) small(2) medium(1) POE(2)
DrainingClasses     =
DrainedClasses      =
Pool                = 1
Fabric Connectivity = 1
Adapter             = css0(switch,c166sn39.ppd.pok.ibm.com,9.114.72.167,38,4/4,80M/80M,1,
                      READY) csss(striped,,,38,4/4,80M/80M,1,READY)
                      en0(ethernet,c168n07.ppd.pok.ibm.com,9.114.72.103)
Feature=
Max_Starters        = 2
Memory              = 224
FreeRealMemory      = 83
PagesFreed          = 0
PagesScanned        = 0
PagesPagedIn        = 0
PagesPagedOut       = 0
ConsumableResources = ConsumableCpus(4,4) resA(26,26)
ConfigTimeStamp     = Wed Apr  8 09:05:36 1998
Cpus                = 1
Speed               = 1.000000
Subnet              = 9.117.17
MasterMachPriority  = 0.000000
CustomMetric        = 1
StartdAvail         = 1
State               = Running
```

```
EnteredCurrentState = Wed Apr  8 09:46:33 1998
START               = T
SUSPEND             = F
CONTINUE            = T
VACATE              = F
KILL                = F
Machine Mode        = general
Running             = 2
ScheddAvail         = 1
ScheddState         = Avail
ScheddRunning       = 3
Pending             = 0
Starting            = 0
Idle                = 0
Unexpanded          = 0
Held                = 0
Removed             = 0
RemovedPending      = 0
Completed           = 0
TotalJobs           = 3
TimeStamp           = Wed Apr  8 09:47:45 1998
```

The long listing includes these fields:

Name hostname of the machine.

Running
The number of initiators used to run LoadLeveler jobs. One initiator is used for each serial job step. One initiator is used for each task of a parallel job step.

ScheddAvail
flag indicating if machine is running a schedd daemon (0=no, 1=yes).

StartdAvail
flag indicating if machine is running a startd daemon (0=no, 1=yes).

State state of the startd daemon, which can be:
Busy
Down
Drain
Flush
Idle
None
Running
Suspend

For a detailed explanation of these states, see “The startd Daemon” on page 15.

OpSys
operating system on this machine.

Arch hardware architecture of machine as listed in configuration file.

Machine
fully qualified name of the machine.

START
the expression, defined following C conventions in the configuration file, that evaluates to true or false (T/F). This determines whether jobs can be started on this machine.

SUSPEND
the expression, defined following C conventions in the configuration file, that

evaluates to true or false (T/F). This determines whether running jobs should be suspended on this machine.

CONTINUE
the expression, defined following C conventions in the configuration file, that evaluates to true or false (T/F). This determines whether suspended jobs are continued on this machine.

VACATE
the expression, defined following C conventions in the configuration file, that evaluates to true or false (T/F). This determines whether suspended jobs are vacated on this machine.

KILL the expression, defined following C conventions in the configuration file, that evaluates to true or false (T/F). This determines whether running jobs should be killed on this machine.

SYSPRIO
actual expression that determines overall system priority of the job, defined in the configuration file.

MACHPRIO
actual expression that determines machine priority, defined in the configuration file.

Machine Mode
the type of job this machine can run. This can be: batch, interactive, or general.

Virtual Memory
available swap space, in kilobytes, on this machine.

Entered Current State
date and time when machine state was set.

Disk available space, in kilobytes (less 512KB) in LoadLeveler's execute directory on this machine.

Keyboard Idle
number of seconds since last keyboard or mouse activity.

LoadAvg
Berkely one-minute load average on machine.

AvailableClasses
set of currently available classes.

DrainingClasses
set of names of classes which are currently being drained on this machine.

DrainedClasses
set of names of classes which have been drained on this machine and are therefore unavailable.

ConfiguredClasses
set of all classes supported on this machine, both those in use and those not in use, as defined in the configuration file.

Pool the identifier of the pool where this startd machine is located.

Adapter
Network adapter information associated with this machine. For a switch adapter, the format of this information is *adapter_name(network_type, interface_name, interface_address, switch_node_number,*

available_adapter_windows/ total_adapter_windows, available_device_memory/total_device_memory, adapter_fabric_connectivity, adapter_state). For non-switch adapters, the format is *adapter name (network_type, interface_name, interface_address)*.

Feature
set of all features on this machine.

Memory
physical memory, in megabytes, on this machine.

Max_Starters
maximum number of initiators that can be used simultaneously on this machine.

Config Time Stamp
date and time of last (re)configuration.

Cpus number of CPUs on this machine.

Speed speed associated with the machine.

MasterMachPriority
The machine priority for the parallel master node.

Subnet
The TCP/IP subnet that this machine resides on.

CustomMetric
The number that indicates the order of the machines for scheduling purposes.

ScheddRunning
The number of job steps submitted to this machine that are running somewhere in the LoadLeveler cluster.

Pending
The number of job steps in this state on this schedd machine.

Starting
The number of job steps in this state on this schedd machine.

Idle The number of job steps in this state on this schedd machine.

Unexpanded
The number of job steps in this state on this schedd machine.

Held The number of job steps in this state on this schedd machine.

Removed
The number of job steps in this state on this schedd machine.

Remove Pending
The number of job steps in this state on this schedd machine.

Completed
The number of job steps in this state on this schedd machine.

Total Jobs
The number of total job steps submitted to this schedd machine.

ScheddState
The state of the schedd on this schedd machine.

time stamp
The date and time the central manager last received a status update from this schedd machine.

FabricConnectivity
A boolean vector representing the current state of connectivity of this machine's switch adapter to the SP switch.

FreeRealMemory
Free real memory, in megabytes, on this machine. This value corresponds to the ″fre″ value of the vmstat command output, which is measured in page blocks.

PagesFreed
Pages freed per second. This value corresponds to the ″fr″ value of the vmstat command output.

PagesPaged In
Pages paged in from paging space per second. This value corresponds to the ″pi″ value of the vmstat command output.

PagesPagedOut
Pages paged out to paging space per second. This value corresponds to the ″po″ value of the vmstat command output.

PagesScanned
Pages scanned by the page-replacement algorithm per second. This value corresponds to the ″sr″ value of the vmstat command output.

ConsumableResources
Consumable resources associated with this machine. The format of this information is *resource_name(available, total)*.

llsubmit - Submit a Job

Purpose

Submits a job to LoadLeveler to be dispatched based upon job requirements in the job command file.

You can submit both LoadLeveler jobs and NQS jobs. To submit NQS jobs, the job command file must contain the shell script to be submitted to the NQS node.

Syntax

llsubmit [**-?**] [**-H**] [**-v**] [**-q**] [*cmdfile* | **–**]

Flags

-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-q Specifies quiet mode: print no messages other than error messages.

cmdfile
Is the name of the job command file containing LoadLeveler commands.

– Specifies that LoadLeveler commands that would normally be in the job command file are read from stdin. When entry is complete, press `Ctrl-D` to end the input.

Related Information

- Users with **uid** or **gid** equal to 0 are not allowed to issue the **llsubmit** command.
- When a LoadLeveler job ends, you may receive UNIX mail notification indicating the job exit status. For example, you could get the following mail message:

  ```
  Your LoadLeveler job
  myjob1
  exited with status 139.
  ```

  The return code 139 is from the user's job, and is not a LoadLeveler return code.
- For information on writing a program to filter job scripts when they are submitted, see "Filtering a Job Script" on page 296.

Examples

In this example, a job command file named *qtrlyrun.cmd* is submitted:

```
llsubmit qtrlyrun.cmd
```

Results

The following shows the results of the **llsubmit qtrlyrun.cmd** command issued from the machine **earth**:

```
llsubmit: The job "earth.505" has been submitted.
```

Note that `505` is the job ID generated by LoadLeveler.

llsummary - Return Job Resource Information for Accounting

Purpose
Returns job resource information on completed jobs for accounting purposes.

Syntax
llsummary [**-?**] [**-H**] [**-v**] [**-x**] [**-l**] [**-s** *MM/DD/YYYY* **to** *MM/DD/YYYY*]
[**-e** *MM/DD/YYYY* **to** *MM/DD/YYYY*] [**-u** *user*] [**-c** *class*] [**-g** *group*] [**-G** *unixgroup*]
[**-a** *allocated*] [**-r** *report*] [**-j** *host.jobid*] [**-d** *section*] [*filename*]

Flags
-? Provides a short usage message.

-H Provides extended help information.

-v Outputs the name of the command, release number, service level, service level date, and operating system used to build the command.

-x Provides extended information. Using **-x** can produce a very long report. This option is meaningful only when used with the **-l** option. You must enable the recording of accounting data in order to collect information with the **-x** flag. To do this, specify **ACCT=A_ON A_DETAIL** in your **LoadL_config** file.

-l Specifies that the long form of output is displayed.

-s Specifies a range for the start date (queue date) for accounting data to be included in this report. The format for entering the date is either *MM/DD/YYYY* (where *MM* is month, *DD* is day, and *YYYY* is year), *MM/DD/YY* (where *YY* is a two-digit year value), or a string of digits representing the number of seconds since 1970. If a two-digit year value is used, then 69-99 maps to 1969-1999, and 00-68 maps to 2000-2068. The default is to include all the data in the report.

-e Specifies a range for the end date (completion date) for accounting data to be included in this report. The format for entering the date is either *MM/DD/YYYY* (where *MM* is month, *DD* is day, and *YYYY* is year), *MM/DD/YY* (where *YY* is a two-digit year value), or a string of digits representing the number of seconds since 1970. The default is to include all the data in the report.

-u *user*
Specifies the user ID for whom accounting data is reported.

-c *class*
Specifies the class for which accounting data is reported. For reports of all formats (short, long and extended), llsummary will report information about every job which contains at least one step of the specified class. For the short format, llsummary also reports a job count and step count for each class; for these counts, a job's class is determined by the class of its first step.

-g *group*
Specifies the LoadLeveler group for which accounting data is reported. For reports of all formats (short, long and extended), llsummary reports information about every job which contains at least one step of the specified group. For the short format, llsummary also reports a job count and step count for each group; for these counts, a job's group is determined by the group of its first step.

-G *unixgroup*
Specifies the UNIX group for which accounting data is reported.

-a *allocated*
Specifies the hostname that was allocated to run the job. You can specify the allocated host in short or long form.

-r *report*
Specifies the report type. You can choose one or more of the following reports:

resource
Provides CPU usage for all submitted jobs, including those that did not run. This is the default.

avgthroughput
Provides average queue time, run time, and CPU time for jobs that ran for at least some period of time.

maxthroughput
Provides maximum queue time, run time, and CPU time for jobs that ran for at least some period of time.

minthroughput
Provides minimum queue time, run time, and CPU time for jobs that ran for at least some period of time.

throughput
Selects all throughput reports.

numeric
Reports CPU times in seconds rather than hours, minutes, and seconds

You must enable the recording of accounting data in order to generate any of the four throughput reports. To do this, specify **ACCT=A_ON A_DETAIL** in your **LoadL_config** file.

-d *section*
Specifies the category (data section) for which you want to generate a report. You can specify one or more of the following: **user**, **group**, **unixgroup**, **class**, **account**, **day**, **week**, **month**, **jobid**, **jobname**, **allocated**.

-j *host.jobid*
The job for which accounting data is reported. *host* is the name of the machine to which the job was submitted. The default is the local machine. *jobid* is the job ID assigned to the job when it was submitted using the **llsubmit** command. The entire *host.jobid* string is required.

filename
The file containing the accounting data. If not specified, the default is the local history file on the machine from which the command was issued. You can use the **llacctmrg** command to produce such a file.

Examples
The following example requests summary reports (standard listing) of all the jobs submitted on your machine between the days of September 12, 1999 and October 12, 1999:

```
llsummary -s 09/12/1999 to 10/12/1999
```

Results
The Standard Listing: The standard listing is generated when you do not specify **-l**. **-r**, or **-d** with **llsummary**. This sample report includes summaries of the following data:
- Number of jobs, Total CPU usage, per user.
- Number of jobs, Total CPU usage, per class.

- Number of jobs, Total CPU usage, per group.
- Number of jobs, Total CPU usage, per account number.

The following is an example of the standard listing:

```
      Name   Jobs   Steps        Job Cpu    Starter Cpu     Leverage
    krystal     15      36     0+00:09:50     0+00:00:10         59.0
     lixin3     18      54     0+00:08:28     0+00:00:16         31.8
      TOTAL     33      90     0+00:18:18     0+00:00:27         40.7

     Class   Jobs   Steps        Job Cpu    Starter Cpu     Leverage
     small      9      21     0+00:01:03     0+00:00:06         10.5
     large     12      36     0+00:13:45     0+00:00:11         75.0
      osl2      3       9     0+00:00:27     0+00:00:02         13.5
  No_Class      9      24     0+00:03:01     0+00:00:06         30.2
     TOTAL     33      90     0+00:18:18     0+00:00:27         40.7

     Group   Jobs   Steps        Job Cpu    Starter Cpu     Leverage
  No_Group     12      30     0+00:09:32     0+00:00:09         63.6
 chemistry      7      18     0+00:04:50     0+00:00:05         58.0
engineering     14      42     0+00:03:56     0+00:00:12         19.7
 TOTAL          33      90     0+00:18:18     0+00:00:27         40.7

   Account   Jobs   Steps        Job Cpu    Starter Cpu     Leverage
     33333     16      39     0+00:05:54     0+00:00:11         32.2
     22222     15      45     0+00:12:05     0+00:00:13         55.8
     99999      2       6     0+00:00:18     0+00:00:01         18.0
     TOTAL     33      90     0+00:18:18     0+00:00:27         40.7
```

The standard listing includes the following fields:

Name User ID submitting jobs.

Class Class specified or defaulted for the jobs.

Group User's login group.

Account
Account number specified for the jobs.

Jobs Count of the total number of jobs submitted by this user, class, group, or account.

Steps Count of the total number of job steps submitted by this user, class, group, or account.

Job CPU
Total CPU time consumed by user's jobs.

Starter CPU
Total CPU time consumed by LoadLeveler starter processes on behalf of the user jobs.

Leverage
Ratio of job CPU to starter CPU.

The -r Listing: The following is sample output from the **llsummary -r throughput** command. Only the user output is shown; the class, group, and account lines are not shown.

```
 Name   Jobs   Steps   AvgQueueTime    AvgRealTime     AvgCPUTime
loadl      1       4     0+00:00:03     0+00:05:27     0+00:05:17
user1      2       6     0+00:03:05     0+00:03:45     0+00:03:04
  ALL      3      10     0+00:01:52     0+00:04:26     0+00:03:58

 Name   Jobs   Steps   MinQueueTime    MinRealTime     MinCPUTime
loadl      1       4     0+00:00:01     0+00:02:49     0+00:02:44
user1      2       6     0+00:02:02     0+00:03:43     0+00:03:02
  ALL      3      10     0+00:00:01     0+00:02:49     0+00:02:44

 Name   Jobs   Steps   MaxQueueTime    MaxRealTime     MaxCPUTime
loadl      1       4     0+00:00:06     0+00:12:58     0+00:12:37
user1      2       6     0+00:06:21     0+00:03:48     0+00:03:07
  ALL      3      10     0+00:06:21     0+00:12:58     0+00:12:37
```

The **-r** listing includes the following fields:

AvgQueueTime
Average amount of time the job spent queued before running for this user, class, group, or account.

AvgRealTime
Average amount of accumulated wall clock time for jobs associated with this user, class, group, or account.

AvgCPUTime
Average amount of accumulated CPU time for jobs associated with this user, class, group, or account.

MinQueueTime
Time of the job that spent the least amount of time in queue before running for this user, class, group, or account.

MinRealTime
Time of the job with the least amount of wall clock time for this user, class, group, or account.

MinCPUime
Time of the job with the least amount of CPU time for this user, class, group, or account.

The MaxQueueTime, MaxRealTime, and MaxCPUTime fields display the time of the job with the greatest amount of queue, wall clock, and CPU time, respectively. The ALL line for the Average listing displays the average time for all users, classes, groups, and accounts. The ALL line for the Minimum listing displays the time of the job with the least amount of time for all users, classes, groups, and accounts. The ALL line for the Maximum listing displays the time of the job with the greatest amount of time for all users, classes, groups, and accounts.

The Long Listing: When you specify the **-x** option in conjunction with the **-l** option on the **llsummary** command, the long report resembles the following:

```
================== Job c163n12.ppd.pok.ibm.com 10 ==================
            Job Id: c163n12.ppd.pok.ibm.com 10
          Job Name: c163n12.ppd.pok.ibm.com.10
  Structure Version: 210
             Owner: loadl
        Unix Group: loadl
   Submitting Host: c163n12.ppd.pok.ibm.com
  Submitting Userid: 1064
 Submitting Groupid: 222
   Number of Steps: 1
```

```
------------------ Step c163n12.ppd.pok.ibm.com.10.0 ------------------
       Job Step Id: c163n12.ppd.pok.ibm.com.10.0
         Step Name: 0
        Queue Date: Mon Jun 28 11:27:28 EDT 1999
        Dependency:
            Status: Completed
     Dispatch Time: Mon Jun 28 11:27:28 EDT 1999
   Completion Date: Mon Jun 28 11:37:48 EDT 1999
   Completion Code: 0
       Start Count: 1
     User Priority: 50
      user_sysprio: 50
     class_sysprio: 30
     group_sysprio: 0
     Notifications: Complete
 Virtual Image Size: 376 kilobytes
         Checkpoint: no
            Restart: yes
    Hold Job Until:
                Cmd: /bin/poe
               Args: /test/loadl/ivp_600 -euilib us -ilevel 6 -labelio yes -pmdlog yes
                Env: MANPATH=/usr/local/man:/usr/share/man; LANG=en_US; LOGIN= ...
                 In: /dev/null
                Out: poe5_1.c163n12.10.0.out
                Err: poe5_1.c163n12.10.0.err
Initial Working Dir: /test/loadl
       Requirements: (Arch == "R6000") && (OpSys == "AIX43")
        Preferences:
          Step Type: General Parallel
    Min Processors: 2
    Max Processors: 2
  Alloc. Host Count: 2
    Allocated Host: c163n12.ppd.pok.ibm.com
                    c163n11.ppd.pok.ibm.com
        Node Usage: shared
       Notify User: loadl
             Shell: /bin/ksh
  LoadLeveler Group: No_Group
             Class: Parallel
    Cpu Hard Limit: 3600 seconds
    Cpu Soft Limit: 1200 seconds
   Data Hard Limit: -1
   Data Soft Limit: -1
   Core Hard Limit: -1
   Core Soft Limit: -1
   File Hard Limit: -1
   File Soft Limit: -1
  Stack Hard Limit: -1
  Stack Soft Limit: -1
    Rss Hard Limit: -1
    Rss Soft Limit: -1
Step Cpu Hard Limit: 5400 seconds
Step Cpu Soft Limit: 2400 seconds
Wall Clk Hard Limit: 6000 seconds
Wall Clk Soft Limit: 3600 seconds
           Comment:
           Account: 99999
   NQS Submit Queue:
   NQS Query Queues:
  Job Tracking Exit:
  Job Tracking Args:
      Task_geometry:
          Resources:
           Blocking: UNSPECIFIED
```

```
--------------- Detail for c163n12.ppd.pok.ibm.com.10.0 ---------------
       Running Host: c163n12.ppd.pok.ibm.com
      Machine Speed: 1.000000
              Event: System
         Event Name: started
      Time of Event: Mon Jun 28 11:27:28 EDT 1999
  Starter User Time:   0+00:00:00.0
Starter System Time:   0+00:00:00.0
 Starter Total Time:   0+00:00:00.0
                ...
                ...
              Event: System
         Event Name: completed
      Time of Event: Mon Jun 28 11:37:48 EDT 1999
  Starter User Time:   0+00:00:00.140000
Starter System Time:   0+00:00:00.190000
 Starter Total Time:   0+00:00:00.330000
     Starter maxrss: 1732
      Starter ixrss: 10720
                ...
                ...
       Running Host: c163n11.ppd.pok.ibm.com
      Machine Speed: 1.000000
              Event: System
         Event Name: started
      Time of Event: Mon Jun 28 11:28:31 EDT 1999
  Starter User Time:   0+00:00:00.0
Starter System Time:   0+00:00:00.0
 Starter Total Time:   0+00:00:00.0
                ...
                ...
              Event: System
         Event Name: completed
      Time of Event: Mon Jun 28 11:38:41 EDT 1999
  Starter User Time:   0+00:00:00.150000
Starter System Time:   0+00:00:00.190000
 Starter Total Time:   0+00:00:00.340000
 Starter maxrss: 1668
      Starter ixrss: 11088
      Starter idrss: 16452
      Starter isrss: 0
     Starter minflt: 1373
     Starter majflt: 0
      Starter nswap: 0
    Starter inblock: 0
    Starter oublock: 0
     Starter msgsnd: 0
     Starter msgrcv: 0
   Starter nsignals: 2
      Starter nvcsw: 50
     Starter nivcsw: 28
     Step User Time:   0+00:00:06.480000
   Step System Time:   0+00:00:01.690000
    Step Total Time:   0+00:00:08.170000
        Step maxrss: 1292
         Step ixrss: 17960
         Step idrss: 437844
         Step isrss: 0
        Step minflt: 2433
        Step majflt: 2
         Step nswap: 0
       Step inblock: 0
       Step oublock: 0
        Step msgsnd: 0
        Step msgrcv: 0
      Step nsignals: 3058
         Step nvcsw: 155458
        Step nivcsw: 498
        Step CPUs: 18
        Step Virtual Memory: 180 megabytes
        Step Real Memory: 90 megabytes
        Step Adapter Memory: 2097152 bytes
```

For an explanation of these fields, see the description of the output fields for the long listing of the **llq** command.

Part 5. The LoadLeveler Graphical User Interface

Chapter 10. Graphical User Interface Overview

This chapter provides some introductory information on the LoadLeveler graphical user interface (GUI). This section provides neither complete nor detailed instructions on using either the LoadLeveler GUI or any other graphical user interface. If this is the first time you are using a Motif-based GUI, you should refer to the appropriate Motif documentation for general GUI information.

This chapter also discusses how to customize your graphical user interface by modifying the **Xloadl** and **Xloadl_so** files and provides a discussion of the **skel.cmd** file.

Note that LoadLeveler provides an installation with two types of graphical user interfaces. One interface is for LoadLeveler users whose machines are interacting fully with LoadLeveler. The second interface is available to users whose machines are only participating on a limited basis. This second type of machine is called a submit-only machine.

Starting the Graphical User Interface

To start the GUI, check your PATH variable to ensure that it is pointing to the LoadLeveler binaries. Also, check to see that your DISPLAY variable is set to your display. Then, type one of the following to start the GUI in the background:

xloadl_so & (if you are running a submit-only machine)
xloadl & (for all other users)

Specifying Options

In general, you can specify GUI options in any of the following ways:

- Within the GUI using menu selections
- On the **xloadl** (or **xloadl_so**) command line. Enter **xloadl -h** or **xloadl_so -h** to see a list of the available options.
- In the **Xloadl** file. See “Customizing the Graphical User Interface” on page 241 for more information.

The LoadLeveler Main Window

LoadLeveler’s main window has three sub-windows, titled Jobs, Machines, and Messages, as shown in Figure 32 on page 224. Each of these sub-windows has its own menu bar.

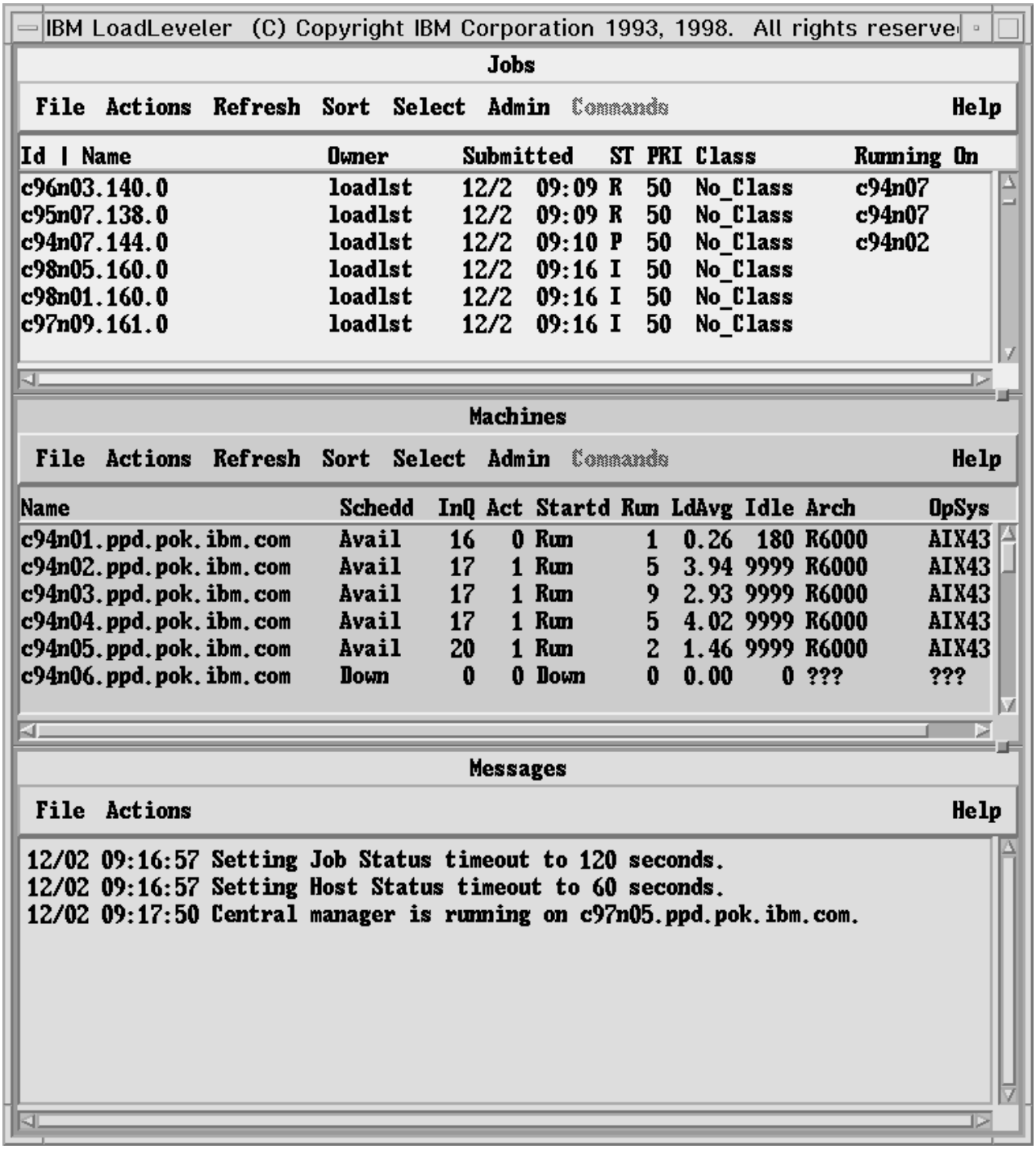

Figure 32. Main Window of the LoadLeveler GUI

The menu bar on the Jobs window relates to actions you can perform on jobs. The menu bar on the Machines window relates to actions you can perform on machines. Similarly, the menu bar on the Messages window displays actions you can perform related to LoadLeveler generated messages.

When you select an item from a menu bar, a pull-down menu appears. You can select an item from the pull-down menu to carry out an action or to bring up another pull-down menu originating from the first one.

Getting Help Using the Graphical User Interface

You can get help when using the GUI by pressing the Help key. This key is function key 1 (F1) on most keyboards. To receive help on specific parts of the LoadLeveler GUI, place the cursor over the area or field on which you want help and press F1. A help screen appears describing that area. You can also get help by using the Help pulldown menu and the Help push buttons available in pop-up windows.

Before you invoke the GUI, make sure your PATH statement includes the directory containing the LoadLeveler executable. Otherwise, some GUI functions may not work correctly.

Differences Between LoadLeveler's Graphical User Interface and Other Graphical User Interfaces

LoadLeveler's GUI contains many items common to other GUIs. There are, however, some differences that you should be aware of. These differences are:

- Accelerators or mnemonics do not appear on the menu bars.
- Submerged windows do not necessarily rise to the top when refreshed.

Building and Submitting Jobs Using the Graphical User Interface

This chapter explains how to build and submit a job to LoadLeveler using the GUI. In addition, you will learn how to perform other job related tasks. You can accomplish these same tasks by using the LoadLeveler commands. For information on these commands, refer to "Part 4. Command Reference" on page 165.

This manual presents step-by-step instructions for performing tasks. For each step in a task, a user action and a system response to the action are included. User actions appear in uppercase boldface type, for example: **SELECT**. The system response to an action follows a ▲. For example:

▲ The main window appears.

An action is sometimes represented by itself, for example:

SELECT OK

Other actions can require a selection or decision. Selection and decision actions are presented in tables.

Selection tables list all possible selections in the left column of the table. The following is an example of a selection table:

To	Do This
Submit a job	Refer to "Step 3: Submit a Job Command File" on page 235.
Cancel a job	Refer to "Step 9: Cancel a Job" on page 237.

Decision tables present a question or series of questions before indicating the action. The following is an example of a decision table:

Did the job you submitted complete processing?	
Yes	Submit another job.
No	Check the status of the job.

Selections from a menu bar are indicated with an →. For example, if a menu bar included an option called **Actions** and **Actions** included an option called **Cancel**, the instructions would read:

SELECT **Actions → Cancel**

Task Scenario Using the Graphical User Interface

The tasks described in this chapter are those that you, as a user might be interested in accomplishing and are presented in a typical step-by-step scenario. You do not have to follow the steps shown here and may perform certain tasks before others without any difficulty. Some tasks must be performed prior to others in order for succeeding tasks to work. For example, you cannot submit a job if you do not have a job command file that you built using either the GUI or an editor.

Step 1: Build a Parallel Job

From the Jobs window:

SELECT **File → Build a Job → Parallel**

▲ The dialog box shown in Figure 33 on page 227 appears:

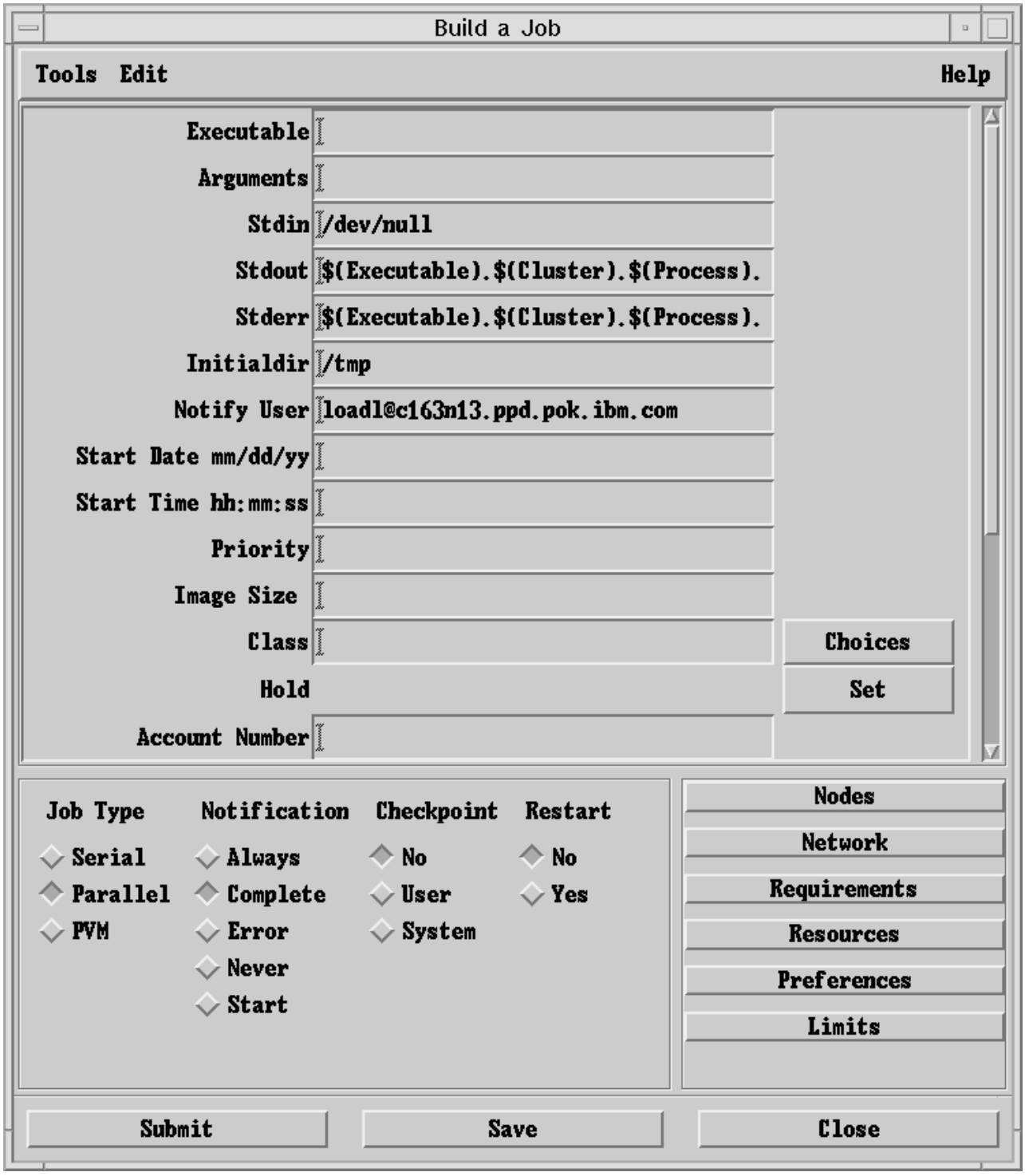

Figure 33. LoadLeveler Build a Job Window

Complete those fields for which you want to override what is currently specified in your **skel.cmd** defaults file. A sample **skel.cmd** file is found in **/usr/LoadL/full/samples**. You can update this file to define defaults for your site, and then update the ***skelfile** resource in **Xloadl** to point to your new **skel.cmd** file. If

you want a personal defaults file, copy **skel.cmd** to one of your directories, edit the file, and update the ***skelfile** resource in **.Xdefaults**.

Field	Input
Executable	Name of the program to run. It must be an executable file. Optional. If omitted, the command file is executed as if it were a shell script.
Arguments	Parameters to pass to the program. Required only if the executable requires them.
Stdin	Filename to use as standard input (stdin) by the program. Optional. The default is **/dev/null**.
Stdout	Filename to use as standard output (stdout) by the program. Optional. The default is **/dev/null**.
Stderr	Filename to use as standard error (stderr) by the program. Optional. The default is **/dev/null**.
Initialdir	Initial directory. LoadLeveler changes to this directory before running the job. Optional. The default is your current working directory.
Notify User	User id of person to notify regarding status of submitted job. Optional. The default is your userid.
StartDate	Month, day, and year in the format mm/dd/yy. The job will not start before this date. Optional. The default is to run the job as soon as possible.
StartTime	Hour, minute, second in the format hh:mm:ss. The job will not start before this time. Optional. The default is to run the job as soon as possible. If you specify *StartTime* but not *StartDate*, the default *StartDate* is the current day. If you specify *StartDate* but not StartTime, the default *StartTime* is 00:00:00. This means that the job will start as soon as possible on the specified date.
Priority	Number between 0 and 100, inclusive. Optional. The default is 50. This is the user priority. For more information on this priority, refer to “Setting and Changing the Priority of a Job” on page 28.
Image size	Number in kilobytes that reflects the maximum size you expect your program to grow to as it runs. Optional.
Class	Class type. The job will only run on machines that support the specified class type. Your system administrator defines the class types. Optional. You can press the Choices button to get a list of available classes. Press the Details button under the class list to verify your permissions.
Hold	Hold status of the submitted job. Permitted values are: **user** user hold **system** system hold (only valid for LoadLeveler administrators) **usersys** user and system hold (only valid for LoadLeveler administrators) Optional. The default is a no-hold state. Press the **set** button to this field.

Field	Input
Account Number	Number associated with the job. For use with the llacctmrg and llsummary commands for acquiring job accounting data. Optional. Required only if the **ACCT** keyword is set to **A_VALIDATE** in the configuration file.
Environment	Specifies your initial environment variables when your job starts. Separate environment specifications with semicolons. Optional.
Shell	The name of the shell to use for the job. Optional. If not specified, the shell used in the owner's password file entry is used. If none is specified, **/bin/sh** is used.
Group	The LoadLeveler group name to which the job belongs. Optional.
Step Name	The name of this job step. Optional.
Node Usage	How the node is used. Permitted values are: **shared** The node can be shared with other tasks of other job steps. This is the default. **not shared** The node cannot be shared. Optional. Press the **Set** button to set this field.
Dependency	A Boolean expression defining the relationship between the job steps. Optional.
Comments	Comments associated with the job. These comments help to distinguish one job from another job. Optional.
Note: The fields that appear in this table are what you see when viewing the Build a Job window. The text in these fields does not necessarily correspond with the keywords listed in "Job Command File Keywords" on page 36.	

See "Job Command File Keywords" on page 36 for information on the defaults associated with these keywords.

SELECT a Job Type if you want to change the job type you selected on the Build A Job cascading window.

Your choices are:

Serial Specifies a serial job.
Parallel Specifies a non-PVM parallel job.
PVM Specifies a PVM parallel job.

Note that the job type you select affects the choices that are active on the Build A Job window.

SELECT a Notification option

Your choices are:

Always Notify you when the job starts, completes, and if it incurs errors.
Complete Notify you when the job completes. This is the default option as initially defined in the skel.cmd file.

Error Notify you if the job cannot run because of an error.
Never Do not notify you.
Start Notify you when the job starts.

SELECT a Checkpoint option.

Your choices are:
No Do not checkpoint the job. This is the default.
User Yes, checkpoint the job at intervals you determine. See “checkpoint” on page 37 for more information.
System Yes, checkpoint the job at intervals determined by LoadLeveler. See “checkpoint” on page 37 for more information.

SELECT a Restart option

Your choices are:
No Do not restart the job.
Yes Yes, restart the job from an existing checkpoint file when you submit the job.

SELECT Nodes (available when the job type is parallel)

▲ The Nodes dialog box appears.

Complete the necessary fields to specify node information for a parallel job. Depending upon which model you choose, different fields will be available; any unavailable fields will be greyed out. LoadLeveler will assign defaults for any fields that you leave blank.

Field	Available in:	Input
Min # of Nodes	Tasks Per Node Model and Tasks with Uniform Blocking Model	Minimum number of nodes required for running the parallel job. For more information, see “node” on page 47. Optional. The default is one.
Max # of Nodes	Tasks Per Node Model	Maximum number of nodes required for running the parallel job. For more information, see “node” on page 47. Optional. The default is the minimum number of nodes.
Tasks per Node	Tasks Per Node Model	The number of tasks of the parallel job you want to run per node. For more information, see “tasks_per_node” on page 54. Optional.
Total Tasks	Tasks with Uniform Blocking Model, and Custom Blocking Model	The total number of tasks of the parallel job you want to run on all available nodes. For more information, see “total_tasks” on page 55. Optional for Uniform, required for Custom Blocking. The default is one.
Blocking	Custom Blocking Model	The number of tasks assigned (as a block) to each consecutive node until all of a job’s tasks have been assigned. For more information, see “blocking” on page 37
Task Geometry	Custom Geometry Model	The task ids of each task that you want to run on each node. You can use the ″Set Geometry″ button for step-by-step directions. For more information, see “task_geometry” on page 54

SELECT Close to return to the Build a Job dialog box.

SELECT Network (available when the job type is parallel)

▲ The Network dialog box appears.

Complete those fields for which you want to specify network information. For more information, see “network” on page 45.

Field	Input
MPI/LAPI	Choose one, both, or none of these boxes to specify the MPI (Message Passing Interface) protocol, the (LAPI Low-level Application Programming Interface) protocol, both protocols, or neither protocol. Optional.
Adapter/Network	Select an adapter name or a network type from the list. Required for each protocol you select.
Adapter Usage	Specifies that the adapter is either shared or not shared. Optional. The default is shared.
Communication Mode	Specifies the mode in which an SP switch adapter is used, and can be either IP (internet Protocol) or US (User Space). Optional. The default is IP.
Communication Level	Implies the amount of memory to be allocated to each window for the corresponding protocol, and can be Low, Average, or High.

SELECT Close to return to the Build a Job dialog box.

SELECT Requirements

▲ The Requirements dialog box appears.

Complete those fields for which you want to specify requirements. Defaults are used for those fields that you leave blank. LoadLeveler dispatches your job only to one of those machines with resources that matches the requirements you specify.

Field	Input
Architecture*	Machine type. The job will not run on any other machine type. Optional. The default is the architecture of your current machine.
Operating System*	Operating system. The job will not run on any other operating system. Optional. The default is the operating system of your current machine.
Disk	Amount of disk space in the execute directory. The job will only run on a machine with at least this much disk space. Optional. The default is defined in your local configuration file.
Memory	Amount of memory. The job will only run on a machine with at least this much memory. Optional. The default is defined in your local configuration file.
Machine(s)	Machine name(s). The job will only run on the specified machines. Optional.
Feature(s)	Features. The job will only run on machines with specified features. Optional.

Field	Input
LoadLeveler Version	Specifies the version of LoadLeveler, in dotted decimal format, on the machine where you want the job to run. For example: 2.1.0.0 specifies that your job will run on a machine running LoadLeveler Version 2.1.0.0 or higher. Optional.
Pool	Specifies the number associated with the pool you want to use. All available pools listed in the administration file appear as choices. The default is to select nodes from any pool.
Requirement	Requirements. The job will only run if these requirements are met.
Note: If you enter a resource that is not available, you will NOT receive a message. LoadLeveler holds your job in the Idle state until the resource becomes available. Therefore, ensure the spelling of your entry is correct. You can issue **llq -s** *jobID* to find out if you have a job for which requirements were not met.	
*If you do not specify an architecture or operating system, LoadLeveler assumes that your job can run only on your machine's architecture and operating system. If your job is not a shell script that can be run successfully on any platform, you should specify a required architecture and operating system.	

SELECT Close to return to the Build a Job dialog box.

SELECT Resources

▲ The Resources dialog box appears.

This dialog box allows you to set the amount of defined consumable resources required for a job step. Resources with an ″*″ appended to their names are not in the SCHEDULE_BY_RESOURCES list. For more information, see "resources" on page 52.

SELECT Close to return to the Build a Job dialog box.

SELECT Preferences

▲ The Preferences dialog box appears.

This dialog box is similar to the Requirements dialog box, with the exception of the Adapter choice, which is not supported as a Preference. Complete the fields for those parameters that you want to specify. These parameters are not binding. For any preferences that you specify, LoadLeveler attempts to find a machine that matches these preferences along with your requirements. If it cannot find the machine, LoadLeveler chooses the first machine that matches the requirements.

SELECT Close to return to the Build a Job dialog box.

SELECT Limits

▲ The Limits dialog box appears.

Complete the fields for those limits that you want to impose upon your job. If you type *copy* in any field, the limits in effect on the submit machine are used. If you leave any field blank, the default limits in effect for your userid on the machine that runs the job are used.

Field	Input
CPU Limit	Maximum amount of CPU time that the submitted job can use. Express the amount as: `[hours:[minutes:][seconds][ .fraction]` For example, 12:56:21 is 12 hours, 56 minutes, and 21 seconds. Optional
Data Limit	Maximum amount of the data segment that the submitted job can use. Express the amount as: `integer[.fraction][units]` where integer and fraction represent strings of up to eight digits. Optional
Core Limit	Maximum size of a core file. Optional
RSS Limit	Maximum size of the resident set size. It is the largest amount of physical memory a user's process can allocate. Optional
File Limit	Maximum size of a file that is created. Optional
Stack Limit	Maximum size of the stack. Optional
Job CPU Limit	Maximum amount of CPU a single job step can use per processor. Optional
Wall Clock Limit	Maximum amount of elapsed time for which a job can run. Optional

SELECT Close to return to the Build a Job dialog box.

SELECT PVM to select a PVM job.

▲ The PVM dialog box appears.

Complete those fields for which you want to specify requirements. Defaults are used for those fields that you leave blank.

Field	Input
Min # of Processors	Minimum number of processors required for running the PVM job. Optional. The default is one.
Max # of Processors	Maximum number of processors required for running the PVM job. Optional. The default is one.
Parallel Path	The directory that defines where the PVM3 executables are located.
PVM	Specifies that an adapter is used for this PVM job.
Adapter/Network	Select an adapter name or a network type from the list. Required.

Field	Input
Adapter Usage	Specifies that the adapter is either shared or not shared. Optional. The default is shared.

SELECT Close to return to the Build a Job dialog box.

Step 2: Edit the Job Command File

There are several ways that you can edit the job command file that you just built:

1. Using the Jobs window:

   SELECT **File → Submit a Job**

   ▲ The Submit a Job dialog box appears.

   SELECT the job file you want to edit from the file column.

   SELECT **Edit**

   ▲ Your job command file appears in a window. You can use any editor to edit the job command file. The default editor is specified in your .Xdefaults file.

   If you have an icon manager, an icon may appear. An icon manager is a program that creates a graphic symbol, displayed on a screen, that you can point to with a device such as a mouse in order to select a particular function or application. Select this icon to view your job command file.

2. Using the **Tools Edit** pulldown menus on the Build a Job window:

   Using the Edit pulldown menu, you can modify the job command file. Your choices appear in the following table:

To	Select
Add a step to the job command file	Add a Step
Delete a step from the job command file	Delete a Step
Clear the fields in the Build a Job window	Clear Fields
Select defaults to use in the fields	Set Field Defaults
Note: Other options include Go to Next Step, Go to Previous Step, and Go to Last Step that allow you to edit various steps in the job command file.	

   Using the **Tools** pulldown menu, you can modify the job command file. Your choices appear in the following table:

To	Select
Name the job	Set Job Name
Open a window where you can enter a script file	Append Script
Fill in the fields using another file	Restore from File
View the job command file in a window	View Entire Job
Determine which step you are viewing	What is step #
Start a new job command file	Start a new job

To	Do This	
Save the information you entered into a file which you can submit later	**SELECT**	**Save** ▲ A window appears prompting you to enter a job filename.
	ENTER	a job filename in the text entry field.
	SELECT	**OK** ▲ The window closes and the information you entered is saved in the file you specified.
Submit the program immediately and discard the information you entered	**SELECT**	**Submit**
	GO TO	Step 4

If you already submitted your job, go to “Step 4: Display, Refresh and Obtain Job Status”. Otherwise, go to “Step 3: Submit a Job Command File”.

Step 3: Submit a Job Command File

After building a job command file, you can submit it to one or more machines for processing. In addition to scripts with LoadLeveler keywords, you can also submit scripts that contain NQS options. You cannot, however, in this release of LoadLeveler, combine NQS and LoadLeveler options.

To submit a job, from the Jobs window:

SELECT **File → Submit a Job**

▲ The Submit a Job dialog box appears.

SELECT the job file that you want to submit from the file column.

You can also use the filter field and the directories column to select the file or you can type in the file name in the text entry field.

SELECT **Submit**

▲ The job is submitted for processing.

You can now submit another job or you can press Close to exit the window.

Go to the next step.

Step 4: Display, Refresh and Obtain Job Status

When you submit a job, the status of the job is automatically displayed in the Jobs window. You can update or refresh this status using the Jobs window and selecting one of the following:

- **Refresh → Refresh Jobs**
- **Refresh → Refresh All**.

To change how often the amount of time should pass before the jobs window is automatically refreshed, use the Jobs window.

SELECT **Refresh → Set Auto Refresh**

▲ A window appears.

TYPE IN a value for the number of seconds to pass before the Jobs window is updated.

Automatic refresh can be expensive in terms of network usage and CPU cycles. You should specify a refresh interval of 120 seconds or more for normal use.

SELECT **OK**

▲ The window closes and the value you specified takes effect.

To receive detailed information on a job:

SELECT **Actions → Extended Status** to receive additional information on the job. Selecting this option is the same as typing **llq -x** command.

You can also get information in the following way:

SELECT **Actions → Extended Details**

Selecting this option is the same as typing **llq -x -l** command. You can also double click on the job in the Jobs window to get details on the job.

Note: Obtaining extended status or details on multiple jobs can be expensive in terms of network usage and CPU cycles.

SELECT **Actions → Job Status**

You can also use the **llq -s** command to determine why a submitted job remains in the Idle or Deferred state.

For more information on these states, see “llq - Query Job Status” on page 193.

Go to the next step.

Step 5: Sort the Jobs Window

You can specify up to two sorting options for the Jobs window. The options you specify determine the order in which the jobs appear in the Jobs window.

From the Jobs window:

Action	Select Sort →	Type of Sort
Sort jobs by the machine from which they were submitted	**Sort by Submitting Machine →**	[Primary|Secondary]
Sort by owner	**Sort by Owner →**	[Primary|Secondary]
Sort by the time the jobs were submitted	**Sort by Submission Time →**	[Primary|Secondary]
Sort by the state of the job	**Sort by State →**	[Primary|Secondary]
Sort jobs by their user priority (last job listed runs first)	**Sort by Priority →**	[Primary|Secondary]
Sort by the class of the job	**Sort by Class →**	[Primary|Secondary]
Sort by the group associated with the job	**Sort by Group →**	[Primary|Secondary]
Sort by the machine running the job	**Sort by Running Machine →**	[Primary|Secondary]
Sort by dispatch order	**Sort by Dispatch Order →**	[Primary|Secondary]
Not specify a sort	**No Sort**	[Primary|Secondary]

Each sorting option contains a cascading window which allows you to select this option as either a Primary or Secondary sorting option. For example, suppose you select Sort by Owner as the primary sorting option and Sort by Class as the secondary sorting option. The Jobs window is sorted by owner and, within each owner, by class.

Go to the next step.

Step 6: Change Priorities of Jobs in a Queue

If your job has not yet begun to run and is still in the queue, you can change the priority of the job in relation to your other jobs in the queue that belong to the same class. This only affects the user priority of the job. For more information on this priority, refer to “Setting and Changing the Priority of a Job” on page 28. Only the owner of a job or the LoadLeveler administrator can change the priority of a job.

From the Jobs window:

SELECT a job by clicking on it with the mouse

SELECT **Actions → Priority**

▲ A window appears.

TYPE IN a number between 0 and 100, inclusive, to indicate a new priority.

SELECT **OK**

▲ The window closes and the priority of your job changes.

Go to the next step.

Step 7: Hold a Job

Only the owner of a job or the LoadLeveler administrator can place a hold on a job.

From the Jobs window:

SELECT the job you want to hold by clicking on it with the mouse

SELECT **Actions → Hold**

▲ The job is put on hold and its status changes in the Jobs window.

Go to the next step.

Step 8: Release a Hold on a Job

Only the owner of a job or the LoadLeveler administrator can release a hold on a job.

From the Jobs window:

SELECT the job you want to release by clicking on it with the mouse

SELECT **Actions → Release from Hold**

▲ The job is released from hold and its status is updated in the Jobs window.

Go to the next step.

Step 9: Cancel a Job

Only the owner of a job or the LoadLeveler administrator can cancel a job.

From the Jobs window:

SELECT the job you want to cancel by clicking on it with the mouse

SELECT **Actions → Cancel**

▲ A warning dialog box appears prompting you to confirm your cancellation request. Once you confirm your request, LoadLeveler cancels the job and the job information disappears from the Jobs window.

Go to the next step.

Step 10: Display and Refresh Machine Status

The status of the machines is automatically displayed in the Machines window. You can update or refresh this status using the Machines window and selecting one of the following:

- **Refresh → Refresh Machines**
- **Refresh → Refresh All**.

To specify an amount of time to pass before the Machines window is automatically refreshed, from the Machines window:

SELECT **Refresh → Set Auto Refresh**

▲ A window appears.

TYPE IN a value for the number of seconds to pass before the Machines window is updated.

Automatic refresh can be expensive in terms of network usage and CPU cycles. You should specify a refresh interval of 120 seconds or more for normal use.

SELECT **OK**

▲ The window closes and the value you specified takes effect.

Go to the next step.

Step 11: Sort the Machines Window

You can specify up to two sorting options for the Machines window. The options you specify determine the order in which machines appear in the window.

From the Machines window:

Action	Select Sort →	Sort Type
Sort by machine name	**Sort by Name →**	[Primary|Secondary]
Sort by schedd state	**Sort by Schedd →**	[Primary|Secondary]
Sort by total number of jobs scheduled	**Sort by InQ →**	[Primary|Secondary]
Sort by number of running jobs scheduled by this machine	**Sort by Act →**	[Primary|Secondary]
Sort by startd state	**Sort by Startd →**	[Primary|Secondary]
Sort by the number of jobs running on this machine	**Sort by Run →**	[Primary|Secondary]
Sort by load average	**Sort by LdAvg →**	[Primary|Secondary]
Sort by keyboard idle time	**Sort by Idle →**	[Primary|Secondary]
Sort by hardware architecture	**Sort by Arch →**	[Primary|Secondary]
Sort by operating system type	**Sort by OpSys →**	[Primary|Secondary]

Action	Select Sort →	Sort Type
Not specify a sort	**No Sort**	[Primary|Secondary]

Each sorting option contains a cascading window which allows you to select this option as either a Primary or Secondary sorting option. For example, suppose you select Sort by Arch as the primary sorting option and Sort by Name as the secondary sorting option. The Machines window is sorted by by hardware architecture, and within each architecture type, by machine name.

Go to the next step.

Step 12: Find the Location of the Central Manager

The LoadLeveler administrator designates one of the nodes in the LoadLeveler cluster as the central manager. When jobs are submitted at any node, the central manager is notified and decides where to schedule the jobs. In addition, it keeps track of the status of machines in the cluster and the jobs in the system by communicating with each node. LoadLeveler uses this information to make the scheduling decisions and to respond to queries.

To find the location of the central manager, from the Machines window:

SELECT **Actions → Find Central Manager**

▲ A message appears in the message window declaring on which machine the central manager is located.

Go to the next step.

Step 13: Find the Location of the Public Scheduling Machines

Public scheduling machines are those machines that participate in the scheduling of LoadLeveler jobs on behalf of the submit-only machines.

To get a list of these machines in your cluster, use the Machines window:

SELECT **Actions → Find Public Scheduler**

▲ A message appears displaying the names of these machines.

Go to the next step.

Step 14: Specify Which Jobs Appear in the Jobs Window

Normally, only your jobs appear in the Jobs window. You can, however, specify which jobs you want to appear by using the Select pull-down menu on the Jobs window.

To Display	Select Select →
All jobs in the queue	**All**
All jobs belonging to a specific user (or users)	**By User** ▲ A window appears prompting you to enter the user IDs whose jobs you want to view.

To Display	Select Select →
All jobs submitted to a specific machine (or machines)	**By Machine** ▲ A window appears prompting you to enter the machine names on which the jobs you want to view are running.
All jobs belonging to a specific group (or groups)	**By Group** ▲ A window appears prompting you to enter the LoadLeveler group names to which the jobs you want to view belong.
All jobs having a particular ID	**By Job Id** A dialog box prompts you to enter the id of the job you want to appear. This ID appears in the left column of the Jobs window. Type in the ID and press OK.
Note: When you choose By User, By Machines, or By Group, you can use a UNIX regular expression enclosed in parentheses. For example, you can enter **(^k10)** to display all machines beginning with the characters “k10”.	

SELECT **Select → Show Selection** to show the selection parameters.

Go to the next step.

Step 15: Specify Which Machines Appear in Machines Window

You can specify which machines will appear in the Machines window. The default is to view all of the machines in the LoadLeveler pool.

From the Machines window:

To	Select Select →
View all of the machines	**All**
View machines by operating system	**by OpSys** ▲ A window appears prompting you to enter the operating system of those machines you want to view.
View machines by hardware architecture	**by Arch** ▲ A window appears prompting you to enter the hardware architecture of those machines you want to view.

To	Select Select →
View machines by state	**by State** ▲ A cascading pulldown menu appears prompting you to select the state of the machines that you want to view.

SELECTt **Select → Show Selection** to show the selection parameters.

Go to the next step.

Step 16: Save LoadLeveler Messages in a File

Normally, all the messages that LoadLeveler generates appear in the Messages window. If you would also like to have these messages written to a file, use the Messages window.

SELECT **Actions → Start logging to a file**

▲ A window appears prompting you to enter a filename in which to log the messages.

TYPE IN the filename in the text entry field.

SELECT **OK**

▲ The window closes.

Customizing the Graphical User Interface

You can customize the GUI to suit your needs by overriding the default settings of the LoadLeveler resource variables. For example, you can set the color, initial size, and location of the main window.

This section tells you how to customize the GUI by modifying either (or both) of the following files:

Xloadl for fully participating machines

Xloadl_so for submit-only machines

If the system administrator has set up these resource files, the files are located in the **/usr/lib/X11/app-defaults** directory. Otherwise, the files are located in the lib directory of the LoadLeveler release directory. This is **/usr/lpp/LoadL/full/lib** and **/usr/lpp/LoadL/so/lib**, respectively. These files contain the default values for the graphical user interface. This section discusses the syntax of these files, and gives you an overview of some of the resources you can modify.

An administrator with root authority can make changes to the resources for the entire installation by editing the **Xloadl** file. Any user can make local changes by placing the resource names with their new values in the user's **.Xdefaults** file.

Syntax of an Xloadl File

- Comments begin with !
- Resource variables may begin with *
- Colons follow resource variables
- Resource variable values follow colons.

Modifying Windows and Buttons

All of the windows and buttons that are part of the GUI have certain characteristics in common. For example, they all have a foreground and background color, as well as a size and a location. Each one of these characteristics is represented by a resource variable. For example, the foreground characteristic is represented by the resource variable **foreground**. In addition, every resource variable has a value associated with it. The values of the resource variable **foreground** are a range of colors.

Before customizing a window, you need to locate the resource variables associated with the desired window. To do this, search for the window identifier in your Xloadl file. The following table lists the windows and their respective identifiers:

Table 14. Window Indentifiers in the Xloadl File

Window	**Identifier**
Jobs	job_status
Machines	machine_status
Messages	message_area
Build a Job	builder
Submit a Job	submit
Requirements	requirements
Preferences	preferences
Limits	limits
Account Report Data	reporter
Nodes	nodes
Network	network
PVM	pvm
Script	script

The following table lists the resource variables for all the windows and the buttons along with a description of each resource variable. Use the information in this table to modify your graphical user interface by changing the values of desired resource variables. The values of these resource variables depend upon Motif requirements.

Resource Variable	**Description**
geometry	The location of the object
foreground	The foreground color of the object
background	The background color of the object
width	The width of the object
height	The height of the object
labelString	The text associated with the object

Creating Your Own Pulldown Menus

You can add a pulldown menu to both the Jobs window and the Machines window.

To add a pulldown menu to the Jobs window, in the **Xloadl** file:

1. Set **userJobPulldown** to **True**

2. Set **userJob.labelString** to the name of your menu.
3. Fill in the appropriate information for your first menu item, **userJob_Option1**
4. To define more menu items, fill in the appropriate information for **userJob_Option2**, **userJob_Option3**, and so on. You can define up to ten menu items.

For more information, refer to the comments in the **Xloadl** file.

To add a pulldown menu to the Machines window, in the **Xloadl** file:

1. Set **userMachinePulldown** to **True**
2. Set **userMachine.labelString** to the name of your menu.
3. Fill in the appropriate information for your first menu item, **userMachine_Option1**
4. To define more menu items, fill in the appropriate information for **userMachine_Option2**, **userMachine_Option3**, and so on. You can define up to ten menu items.

Example – Creating a New Pulldown

Suppose you want to create a new menu bar item containing a selection which executes the **ping** command against a machine you select on the Machines window.

```
*userMachinePulldown: True
*userMachine.labelString: Commands
*userMachine_Option1: True
*userMachine_Option1_command: ping -c1
*userMachine_Option1.labelString: ping
*userMachine_Option1_parameter: True
*userMachine_Option1_output: Window
```

Figure 34. Creating a New Pulldown Menu

The **Xloadl** definitions shown in the Figure 34 create a menu bar item called “Commands”. The first item in the Commands pulldown menu is called “ping”. When you select this item, the command **ping -c1** is executed, with the machine you selected on the Machines window passed to this command. Your output is displayed in an informational window.

For more information, refer to the comments in the **Xloadl** file.

Customizing Fields on the Jobs Window and the Machines Window

You can control which fields are displayed and which fields are not displayed on the Jobs window and the Machine window by changing the **Xloadl** file. Look in the **Xloadl** file for “Resources for specifying lengths of fields displayed in the Jobs and Machines windows”.

In most cases, you can remove a field from a window by setting its associated resource value to 0. To remove the Arch field from the Machines window, enter the following:

```
*mach_arch_len : 0
```

Note that the Job ID and Machine Name fields must always be displayed and therefore cannot be set to 0.

All fields have a minimum length value. If you specify a smaller value, the minimum is used.

Modifying Help Panels

Help panels have the same characteristics as all of the windows plus a few unique ones:

Resource Variable	Values	Description
help*work_area.width	Any integer*	The width of the help panel.
help*work_area.height	Any integer*	The height of the help panel.
help*scrollHorizontal	[true|false] The default is False.	Sets the scrolling option on or off.
help*wordWrap	[true|false] The default is True.	Sets word wrapping on or off.
Note: * The work area and height depend upon your screen limitations.		

Administrative Uses for the Graphical User Interface

The end user can perform many tasks more efficiently and faster using the graphical user interface (GUI) but there are certain tasks that end users cannot perform unless they have the proper authority. If you are defined as a LoadLeveler administrator in the LoadLeveler configuration file then you are immediately granted administrative authority and can perform the administrative tasks discussed in this section. To find out how to grant someone administrative authority, see “Step 1: Define LoadLeveler Administrators” on page 99.

You can access LoadLeveler administrative commands using the **Admin** pulldown menu on both the Jobs window and the Machines window of the GUI. The **Admin** pulldown menu on the Jobs window corresponds to the command options available in the **llhold**, **llfavoruser**, and **llfavorjob** commands. The **Admin** pulldown menu on the Machines window corresponds to the command options available in the **llctl** command.

The main window of the GUI, as shown in Figure 32 on page 224, has three sub-windows: one for job status with pull-down menus for job-related commands, one for machine status with pull-down menus for machine-related commands, and one for messages and logs. There are a variety of facilities available that allow you to sort and select the items displayed.

Job Related Administrative Actions

You access the administrative commands that act on jobs through the **Admin** pulldown menu in the Jobs window of the GUI.

You can perform the following tasks with this menu:

Favor Users Allows you to favor users. This means that you can select one or more users whose jobs you want to move up in the job queue. This corresponds to the **llfavoruser** command.

Select **Admin** from the Jobs window

Select **Favor User**

▲The **Order by User** window appears.

Type in
the name of the user for whom you want to favor their jobs.

Press OK

Unfavor Users
Allows you to unfavor users. This means that you want to unfavor the user's jobs which you previously favored. This corresponds to the **llfavoruser** command.

Select **Admin** from the Jobs window

Select **Unfavor User**

▲The **Order by User** window appears.

Type in
the name of the user for whom you want to unfavor their jobs.

Press OK

Favor Jobs Allows you to select a job that you want to favor. This corresponds to the **llfavorjob** command.

Select one or more jobs from the Jobs window

Select **Admin** from the Jobs window

Select **Favor Jobs**

▲The selected jobs are favored.

Press OK

Unfavor Jobs
Allows you select a job that you want to unfavor. This corresponds to the **llfavorjob** command.

Select one or more jobs from the Jobs window

Select **Admin** from the Jobs window

Select **Unfavor Jobs**

▲Unfavors the jobs that you previously selected.

Syshold Allows you to place a system hold on a job. This corresponds to the **llhold** command.

Select a job from the Jobs window

Select **Admin** pulldown menu from the Jobs window

Select **Syshold** to place a system hold on the job.

Release From Hold
Allows you to release the system hold on a job. This corresponds to the **llhold** command.

Select a job from the Jobs window

Select **Admin** pulldown menu from the Jobs window

Select **Release From Hold** to release the system hold on the job.

Machine Related Administrative Actions

You access the administrative commands that act on machines using the **Admin** pulldown menu in the Machines window of the GUI.

Using the GUI pulldown menu, you can perform the tasks described in this section.

Start All Starts LoadLeveler on all machines listed in machine stanzas beginning with the central manager. Use this option when specifying alternate central managers.

Select Admin from the Machines window.

Select Start All

Start LoadLeveler
Allows you to start LoadLeveler on selected machines.

Select one or more machines on which you want to start LoadLeveler.

Select Admin from the Machines window.

Select Start LoadLeveler

Stop LoadLeveler
Allows you to stop LoadLeveler on selected machines.

Select on or more machines on which you want to stop LoadLeveler.

Select Admin from the Machines window.

Select Stop LoadLeveler.

Stop All Stops LoadLeveler on all machines listed in machine stanzas. Use this option when specifying alternate central managers.

Select Admin from the Machines window.

Select Stop All

reconfig forces all daemons to reread the configuration files.

Select the machine on which you want to operate. To reconfigure this **xloadl** session, choose **reconfig** but do not select a machine.

Select Admin from the Machines window.

Select reconfig.

recycle stops all LoadLeveler daemons and restarts them.

Select the machine on which you want to operate.

Select Admin from the Machines window.

Select recycle.

Configuration Tasks
starts Configuration Tasks TaskGuide

Select Admin from the Machines window.

Select Config Tasks

Note: Use the invoking script **lltg** to start the TaskGuide outside of **xloadl**. This option will appear on the pulldown only if the LoadL.tguides fileset is installed.

drain allows no more LoadLeveler jobs to begin running on this machine but it does allow running jobs to complete.

Select the machine on which you want to operate.

Select **Admin** from the Machines window.

Select **drain**.

A cascading menu allows you to select either **daemons**, **schedd**, **startd**, or **startd by class**. If you select **daemons**, both machines will be drained. If you select **schedd**, only the schedd on the selected machine will be drained. If you select **startd**, only the startd on the selected machine will be drained. If you select **startd by class**, a window appears which allows you to select classes to be started.

flush terminates running jobs on this host and sends them back to the system queue to await redispatch. No new jobs are redispatched to this machine until **resume** is issued. Forces a checkpoint if jobs are enabled for checkpointing.

Select the machine on which you want to operate.

Select **Admin** from the Machines window.

Select **flush**.

suspend suspends all jobs on this host.

Select the machine on which you want to operate.

Select **Admin** from the Machines window.

Select **suspend**.

resume resumes all jobs on this machine.

Select the machine on which you want to operate.

Select **Admin** from the Machines window.

Select **resume**.

A cascading menu allows you to select either **daemons**, **schedd**, **startd**, or **startd by class**. If you select **daemons**, both machines will be resumed. If you select **schedd**, only the schedd on the selected machine will be resumed. If you select **startd**, only the startd on the selected machine will be resumed. If you select **startd by class**, a window appears which allows you to select classes to be resumed.

Capture Data collects information on the machines selected.

Select the machine on which you want to operate.

Select **Admin** from the Machines window.

Select **Capture Data**.

Collect Account Data
collects accounting data on the machines selected.

Select the machine on which you want to operate.

Select **Admin** from the Machines window.

Select **Collect Account Data**.

A window appears prompting you to enter the name of the directory in which you want the collected data stored.

Create Account Report

creates an accounting report for you.

Select Admin → Create Account Report...

Note: If you want to receive an extended accounting report, select the **extended** cascading button.

A window appears prompting you to enter the following information:

- A short, long, or extended version of the output. The short version is the default.
- The user ID
- The class name
- The LoadL (LoadLeveler) group name
- The UNIX group name
- The Allocated host
- The job ID
- The report Type
- The section
- A start and end date for the report. If no date is specified, the default is to report all of the data in the report.
- The name of the input data file.
- The name of the output data file. This is the same as stdout.

Press OK

The window closes and you return to the main window. The report appears in the Messages window if no output data file was specified.

version displays version and release data for LoadLeveler on the machines selected in an information window.

Select the machine on which you want to operate.

Select Admin from the Machines window.

Select version.

Part 6. The LoadLeveler Application Programming Interfaces

Chapter 11. LoadLeveler APIs

LoadLeveler provides several Application Programming Interfaces (API) that you can use. LoadLeveler's APIs are interfaces that allow application programs written by customers to interact with the LoadLeveler environment by using specific data or functions that are a part of LoadLeveler. These interfaces can be subroutines within a library or installation exits. This chapter also describes configuration file keywords required to enable these APIs.

This chapter discusses the following:

The header file **llapi.h** defines all of the API data structures and subroutines. This file is located in the **include** subdirectory of the LoadLeveler release directory. You must include this file when you call any API subroutine.

The library **libllapi.a** is a shared library containing all of the LoadLeveler API subroutines. This library is located in the **lib** subdirectory of the LoadLeveler release directory.

Attention: These APIs are not *thread safe*; they should not be linked to by a threaded application.

Accounting API

LoadLeveler provides two subroutines for accounting: one for account validation and one for extracting accounting data.

Account Validation Subroutine

LoadLeveler provides the **llacctval** executable to perform account validation.

Purpose

llacctval compares the account number a user specifies in a job command file with the account numbers defined for that user in the LoadLeveler administration file. If the account numbers match, **llacctval** returns a value of zero. Otherwise, it returns a non-zero value.

Syntax

```
program user_name user_group user_acct# acct1 acct2 ...
```

Parameters

program

Is the name of the program that performs the account validation. The default is **llacctval**. The name you specify here must match the value specified on the **ACCT_VALIDATION** keyword. in the configuration file.

user_name
Is the name of the user whose account number you want to validate.

user_group
Is the login group name of the user.

user_acct#
Is the account number specified by the user in the job command file.

acct1 acct2 ...
Are the account numbers obtained from the user stanza in the LoadLeveler administration file.

Description

llacctval is invoked from within the **llsubmit** command. If the return code is non-zero, **llsubmit** does not submit the job.

You can replace **llacctval** with your own accounting user exit (see below).

To enable account validation, you must specify the following keyword in the configuration file:

```
ACCT = A_VALIDATE
```

To use your own accounting exit, specify the following keyword in the configuration file:

```
ACCT_VALIDATION = pathname
```

where *pathname* is the name of your accounting exit.

Return Values

If the validation succeeds, the exit status must be zero. If it does not succeed, the exit status must be a non-zero number.

Report Generation Subroutine

LoadLeveler provides the **GetHistory** subroutine to generate accounting reports.

Purpose

GetHistory processes local or global LoadLeveler history files.

Library

LoadLeveler API library **libllapi.a**

Syntax

```
#include "llapi.h"

int GetHistory(char *filename, int (*func) (LL_job *), int version);
```

Parameters

filename
Specifies the name of the history file.

(*func***) (LL_job *)**
Specifies the user-supplied function you want to call to process each history record. The function must return an integer and must accept as input a pointer to the LL_job structure. The LL_job structure is defined in the **llapi.h** file.

version
Specifies the version of the history record you want to create. LL_JOB_VERSION in the **llapi.h** file creates an LL_job history record.

Description

GetHistory opens the history file you specify, reads one LL_job accounting record, and calls a user-supplied routine, passing to the routine the address of an LL_job structure. **GetHistory** processes all history records one at a time and then closes the file. Any user can call this subroutine.

The user-supplied function must include the following files:

```
#include <sys/resource.h>
#include <sys/types.h>
#include <sys/time.h>
```

The ll_event_usage structure is part of the LL_job structure and contains the following LoadLeveler defined data:

int *event*
Specifies the event identifier. This is an integer whose value is one of the following:
1 Represents a LoadLeveler-generated event.
2 Represents an installation-generated event.

**char ***_name_
Specifies a character string identifying the event. This can be one of the following:

- An installation generated string that uses the command **llctl capture** *eventname*.
- LoadLeveler-generated strings, which can be the following:
  started
  checkpoint
  vacated
  completed
  rejected
  removed

Return Values

GetHistory returns a zero when successful.

Error Values

GetHistory returns -1 to indicate that the version is not supported or that an error occurred opening the history file.

Examples

Makefiles and examples which use this API are located in the **samples/llphist** subdirectory of the release directory. The examples include the executable **llpjob**, which invokes **GetHistory** to print every record in the history file. In order to compile **llpjob**, the sample Makefile must update the RELEASE_DIR field to represent the current LoadLeveler release directory. The syntax for **llpjob** is:

```
llpjob history_file
```

Where *history_file* is a local or global history file.

Serial Checkpointing API

This section describes **ckpt**, the subroutine used for user-initiated checkpointing of serial jobs. “Step 14: Enable Checkpointing” on page 117 describes how to checkpoint your jobs in various ways including system-initiated and user-initiated. For information of checkpointing parallel jobs, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

ckpt Subroutine

Purpose
Specify the **ckpt** subroutine in a FORTRAN, C, or C++ program to activate user-initiated checkpointing. Whenever this subroutine is invoked, a checkpoint of the program is taken.

C++ Syntax
```
extern "C"{void ckpt();}
```

C Syntax
```
void ckpt();
```

FORTRAN Syntax
```
call ckpt()
```

Related Information
FORTRAN, C, and C++ programs can be compiled with the crxlf, crxlc, and crxlC programs, respectively. These programs are found in the **bin** subdirectory of the LoadLeveler release directory. See “Ensure all User’s Jobs are Linked to Checkpointing Libraries” on page 120 for information on using these compile programs.

The Submit API

This API allows you to submit jobs to LoadLeveler. The submit API consists of the **llsubmit** subroutine, the **llfree_job_info** subroutine, and the monitor program.

llsubmit Subroutine

llsubmit is both the name of a LoadLeveler command used to submit jobs as well as the subroutine described here.

Purpose
The **llsubmit** subroutine submits jobs to LoadLeveler for scheduling.

Syntax
```
int llsubmit (char *job_cmd_file, char *monitor_program,
char *monitor_arg, LL_job *job_info, int job_version);
```

Parameters
job_cmd_file
Is a pointer to a string containing the name of the job command file.

monitor_program
Is a pointer to a string containing the name of the monitor program to be invoked when the state of the job is changed. It is set to NULL if a monitoring program is not provided.

monitor_arg
Is a pointer to a string which is stored in the job object and is passed to the monitor program. The maximum length of the string is 1023 bytes. If the length exceeds this value, it is truncated to 1023 bytes. The string is set to NULL if an argument is not provided.

job_info
Is a pointer to a structure defined in the **llapi.h** header file. No fields are required to be filled in. Upon return, the structure will contain the number of job steps in the job command file and a pointer to an array of pointers to

information about each job step. Space for the array and the job step information is allocated by **llsubmit**. The caller should free this space using the **llfree_job_info** subroutine.

job_version
Is an integer indicating the version of **llsubmit** being used. This argument should be set to **LL_JOB_VERSION** which is defined in the **llapi.h** include file.

Description
LoadLeveler must be installed and configured correctly on the machine on which the submit application is run.

The uid and gid in effect when **llsubmit** is invoked is the uid and gid used when the job is run.

Return Values
0 The job was submitted.

Error Values
-1 The job was not submitted. Error messages are written to stderr.

llfree_job_info Subroutine

Purpose
llfree_job_info frees space for the array and the job step information used by **llsubmit**.

Syntax
```
void llfree_job_info(LL_job *job_info, int job_version);
```

Parameters
job_info
Is a pointer to a **LL_job** structure. Upon return, the space pointed to by the **step_list** variable and the space associated with the **LL_job** step structures pointed to by the **step_list** array are freed. All fields in the **LL_job** structure are set to zero.

job_version
Is an integer indicating the version of **llfree_job_info** being used. This argument should be set to **LL_JOB_VERSION** which is defined in the **llapi.h** header file.

The Monitor Program

Purpose
You can create a monitor program that monitors jobs submitted using the **llsubmit** subroutine. The schedd daemon invokes this monitor program if the **monitor_program** argument to **llsubmit** is not null. The monitor program is invoked each time a job step changes state. This means that the monitor program will be informed when the job step is started, completed, vacated, removed, or rejected. If you suspect the monitor program encountered problems or didn't run, you should check the listing in the **schedd** log. In the event of a monitor program failure, the job is still run.

Syntax
```
monitor_program job_id user_arg state exit_status
```

Parameters

monitor_program
Is the name of the program supplied in the `monitor_program` argument passed to the **llsubmit** function.

job_id
Is the full ID for the job step.

user_arg
The string supplied to the `monitor_arg` argument that is passed to the **llsubmit** function.

state
Is the current state of the job step. Possible values for the state are:

JOB_STARTED
The job step has started.

JOB_COMPLETED
The job step has completed.

JOB_VACATED
The job step has been vacated. The job step will be rescheduled if the job step is restartable or if it is checkpointable.

JOB_REJECTED
A **startd** daemon has rejected the job. The job will be rescheduled to another machine if possible.

JOB_REMOVED
The job step was cancelled or could not be started.

JOB_NOTRUN
The job step cannot be run because a dependency cannot be met.

exit_status
Is the exit status from the job step. The argument is meaningful only if the state is JOB_COMPLETED.

Data Access API

This API gives you access to LoadLeveler objects and allows you to retrieve specific data from the objects. You can use this API to query the negotiator daemon for information about its current set of jobs and machines. The Data Access API consists of the following subroutines:**ll_query**, **ll_set_request**, **ll_reset_request**, **ll_get_objs**, **ll_get_data**, **ll_next_obj**, **ll_free_objs**, and **ll_deallocate**.

Using the Data Access API

To use this API, you need to call the data access subroutines in the following order:

- Call **ll_query** to initialize the query object. See “ll_query Subroutine” on page 257 for more information.
- Call **ll_set_request** to filter the objects you want to query. See “ll_set_request Subroutine” on page 257 for more information.
  - Call **ll_get_objs** to retrieve a list of objects from a LoadLeveler daemon. See “ll_get_objs Subroutine” on page 260 for more information.
    - Call **ll_get_data** to retrieve specific data from an object. See “ll_get_data Subroutine” on page 272 for more information.
  - Call **ll_next_obj** to retrieve the next object in the list. See “ll_next_obj Subroutine” on page 273 for more information.

- Call **ll_free_objs** to free the list of objects you received. See “ll_free_objs Subroutine” on page 274 for more information.
- Call **ll_deallocate** to end the query. See “ll_deallocate Subroutine” on page 274 for more information.

To see code that uses these subroutines, refer to “Examples of Using the Data Access API” on page 275. For more information on LoadLeveler objects, see “Understanding the LoadLeveler Job Object Model” on page 262.

ll_query Subroutine

Purpose
The **ll_query** subroutine initializes the query object and defines the type of query you want to perform. The **LL_element** created and the corresponding data returned by this function is determined by the *query_type* you select.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

LL_element * ll_query(enum QueryType query_type);
```

Parameters
query_type
Can be JOBS (to query job information) or MACHINES (to query machine information, or CLUSTER (to query cluster information).

Description
query_type is the input field for this subroutine.

This subroutine is used in conjunction with other data access subroutines to query information about job and machine objects. You must call **ll_query** prior to using the other data access subroutines.

Return Values
This subroutine returns a pointer to an **LL_element** object. The pointer is used by subsequent data access subroutine calls.

Error Values
NULL The subroutine was unable to create the appropriate pointer.

Related Information
Subroutines: **ll_get_data**, **ll_set_request**, **ll_reset_request**, **ll_get_objs**, **ll_free_objs**, **ll_next_obj**, **ll_deallocate**.

ll_set_request Subroutine

Purpose
The **ll_set_request** subroutine determines the data requested during a subsequent **ll_get_objs** call to query specific objects. You can filter your queries based on the *query_type*, *object_filter*, and *data_filter* you select.

Library
LoadLeveler API library **libllapi.a**

Syntax

```
#include "llapi.h"

int ll_set_request(LL_element *query_element,QueryFlags query_flags,
char **object_filter,DataFilter data_filter);
```

Parameters

query_element
Is a pointer to the **LL_element** returned by the **ll_query** subroutine.

query_flags
When *query_type* (in **ll_query**) is JOBS, *query_flags* can be the following:

QUERY_ALL
Query all jobs.

QUERY_JOBID
Query by job ID.

QUERY_STEPID
Query by step ID.

QUERY_USER
Query by user ID.

QUERY_GROUP
Query by LoadLeveler group.

QUERY_CLASS
Query by LoadLeveler class.

QUERY_HOST
Query by machine name.

When *query_type* (in **ll_query**) is MACHINES, *query_flags* can be the following:

QUERY_ALL
Query all machines.

QUERY_HOST
Query by machine names.

object_filter
Specifies search criteria. The value you specify for *object_filter* is related to the value you specify for *query_flags*:

- If you specify QUERY_ALL, you do not need an *object_filter.*
- If you specify QUERY_JOBID, the *object_filter* must contain a list of job IDs (in the form *schedd_host.cluster*).
- If you specify QUERY_STEPID, the *object_filter* must contain a list of step IDs (in the form *schedd_host.cluster.step*).
- If you specify QUERY_USER, the *object_filter* must contain a list of user IDs.
- If you specify QUERY_CLASS, the *object_filter* must contain a list of LoadLeveler class names.
- If you specify QUERY_GROUP, the *object_filter* must contain a list of LoadLeveler group names.
- If you specify QUERY_HOST, the *object_filter* must contain a list of LoadLeveler machine names. When the query type is JOBS, the machine names must be the names of machines to which the jobs are submitted.

The last entry in the *object_filter* array must be NULL.

data_filter
Filters the data returned from the object you query. The value you specify for *data_filter* is related to the value you specify for *query_type*:

- If you specify JOBS, *data_filter* can be ALL_DATA (the default), which returns the entire object, or Q_LINE, which returns the same information returned by the **llq -f** flag. For more information, see "llq - Query Job Status" on page 193.
- If you specify MACHINES, *data_filter* can be ALL_DATA (the default), which returns the entire object, or STATUS_LINE, which returns the same information returned by the **llstatus -f** flag. For more information, see "llstatus - Query Machine Status" on page 205.

Description

query_element, *query_flags*, *object_filter*, and *data_filter* are the input fields for this subroutine.

You can request a combination of object filters by calling **ll_set_request** more than once. When you do this, the query flags you specify are or-ed together. The following are valid combinations of object filters:

- QUERY_JOBID and QUERY_STEPID. The result is the union of both queries.
- QUERY_HOST and QUERY_USER. The result is the intersection of both queries.
- QUERY_HOST and QUERY_CLASS. The result is the intersection of both queries.
- QUERY_HOST and QUERY_GROUP. The result is the intersection of both queries.

That is, to query jobs owned by certain users and on a specific machines, issue **ll_set_request** first with QUERY_USER and the appropriate user IDs, and then issue it again with QUERY_HOST and the appropriate host names.

For example, suppose you issue **ll_set_request** with a user ID list of anton and meg, and then issue it again with a host list of k10n10 and k10n11. The objects returned are all of the jobs on k10n10 and k10n11 which belong to anton or meg.

Note that if you use two consecutive calls with the same flag, the second call will replace the previous call.

Also, you should not use the QUERY_ALL flag in combination with any other flag, since QUERY_ALL will replace any existing requests.

Return Values

This subroutine returns a zero to indicate success.

Error Values

-1 You specified an invalid *query_element*.
-2 You specified an invalid *query_flag*.
-3 You specified an invalid *object_filter*.
-4 You specified an invalid *data_filter*.
-5 A system error occurred.

Related Information

Subroutines: **ll_get_data**, **ll_query**, **ll_reset_request**, **ll_get_objs**, **ll_free_objs**, **ll_next_obj**, **ll_deallocate**.

ll_reset_request Subroutine

Purpose
The **ll_reset_request** subroutine resets the request data to NULL for the *query_element* you specify.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_reset_request(LL_element *query_element);
```

Parameters
query_element
Is a pointer to the **LL_element** returned by the **ll_query** function.

Description
query_element is the input field for this subroutine.

This subroutine is used in conjunction with **ll_set_request** to change the data requested with the **ll_get_objs** subroutine.

Return Values
This subroutine returns a zero to indicate success.

Error Values
-1 The subroutine was unable to reset the appropriate data.

Related Information
Subroutines: **ll_get_data**, **ll_set_request**, **ll_query**, **ll_get_objs**, **ll_free_objs**, **ll_next_obj**, **ll_deallocate**.

ll_get_objs Subroutine

Purpose
The **ll_get_objs** subroutine sends a query request to the daemon you specify along with the request data you specified in the **ll_set_request** subroutine. **ll_get_objs** receives a list of objects matching the request.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

LL_element * ll_get_objs(LL_element *query_element,LL_Daemon query_daemon,
char *hostname,int *number_of_objs,int *error_code);
```

Parameters
query_element
Is a pointer to the **LL_element** returned by the **ll_query** function.

query_daemon
Specifies the LoadLeveler daemon you want to query. The enum LL_Daemon is defined in llapi.h as:

```
enum LL_Daemon {LL_STARTD, LL_SCHEDD, LL_CM, LL_MASTER, LL_STARTER};
```

The following indicates which daemons respond to which query flags. When *query_type* (in **ll_query**) is JOBS, the *query_flags* (in **ll_set_request**) listed in the lefthand column are responded to by the daemons listed in the righthand column:

QUERY_ALL	negotiator (LL_CM) or schedd (LL_SCHEDD)
QUERY_JOBID	negotiator (LL_CM) or schedd (LL_SCHEDD)
QUERY_STEPID	negotiator (LL_CM)
QUERY_USER	negotiator (LL_CM)
QUERY_GROUP	negotiator (LL_CM)
QUERY_CLASS	negotiator (LL_CM)
QUERY_HOST	negotiator (LL_CM)

When *query_type* (in **ll_query**) is MACHINES, the *query_flags* (in **ll_set_request**) listed in the lefthand column are responded to by the daemons listed in the righthand column:

QUERY_ALL	negotiator (LL_CM)
QUERY_HOST	negotiator (LL_CM)

hostname
Specifies the host name where the **schedd** daemon is queried. If you specify NULL, the **schedd** daemon on the local machine is queried. To contact the negotiator daemon, you do not need to specify a *hostname*.

number_of_objs
Is a pointer to an integer representing the number of objects received from the daemon.

error_code
Is a pointer to an integer representing the error code issued when the function returns a NULL value. See “Error Values”.

Description

query_element, *query_daemon*, and *hostname* are the input fields for this subroutine. *number_of_objs* and *error_code* are output fields.

Each LoadLeveler daemon returns only the objects that it knows about.

Return Values

This subroutine returns a pointer to the first object in the list. You must use the **ll_next_obj** subroutine to access the next object in the list.

Error Values

This subroutine a NULL to indicate failure. The *error_code* parameter is set to one of the following:

-1 You specified an invalid *query_element*.
-2 You specified an invalid *query_daemon*.
-3 The API could not resolve the *hostname*.
-4 You set an invalid request type for the specified daemon.
-5 A system error occurred.
-6 No objects exist matching your request.
-7 An internal error occurred.
-9 Connection to daemon failed.

Related Information

Subroutines: **ll_get_data**, **ll_set_request**, **ll_query**, **ll_get_objs**, **ll_free_objs**, **ll_next_obj**, **ll_deallocate**.

Understanding the LoadLeveler Job Object Model

The **ll_get_data** subroutine of the data access API allows you to access the LoadLeveler job model. The LoadLeveler job model consists of objects that have attributes and connections to other objects. An attribute is a characteristic of the object and generally has a primitive data type (such as integer, float, or character). The job name, submission time and job priority are examples of attributes.

Objects are connected to one or more other objects via relationships. An object can be connected to other objects through more than one relationship, or through the same relationship. For example, A Job object is connected to a Credential object and to Step objects through two different relationships. A Job object can be connected to more than one Step object through the same relationship of “having a Step.” When an object is connected through different relationships, different specifications are used to retrieve the appropriate object.

When an object is connected to more than one object through the same relationship, there are Count, GetFirst and GetNext specifications associated with the relationship. The Count operation returns the number of connections. You must use the GetFirst operation to initialize access to the first such connected object. You must use the GetNext operation to get the remaining objects in succession. You can not use GetNext after the last object has been retrieved.

You can use the **ll_get_data** subroutine to access both attributes and connected objects. See “ll_get_data Subroutine” on page 272 for more information.

The root of the job model is the Job object, as shown in Figure 35 on page 263. The job is queried for information about the number of steps it contains and the time it was submitted. The job is connected to a single Credential object and one or more Step objects. Elements for these objects can be obtained from the job.

You can query the Credential object to obtain the ID and group of the submitter of the job.

The Step object represents one executable unit of the job (all the tasks that are executed together). It contains information about the execution state of the step, messages generated during execution of the step, the number of nodes in the step, the number of unique machines the step is running on, the time the step was dispatched, the execution priority of the step, the unique identifier given to the step by LoadLeveler, the class of the step and the number of processes running for the step (task instances). The Step is connected to one or more Switch Table objects, one or more Machine objects and one or more Node objects. The list of Machines represents all of the hosts where one or more nodes of the step are running. If two or more nodes are running on the same host, the Machine object for the host occurs only once in the step’s Machine list. The Step object is connected to one Switch Table object for each of the protocols (MPI and/or LAPI) used by the Step.

Each Node object manages a set of executables that share common requirements and preferences. The Node can be queried for the number of tasks it manages, and is connected to one or more Task objects.

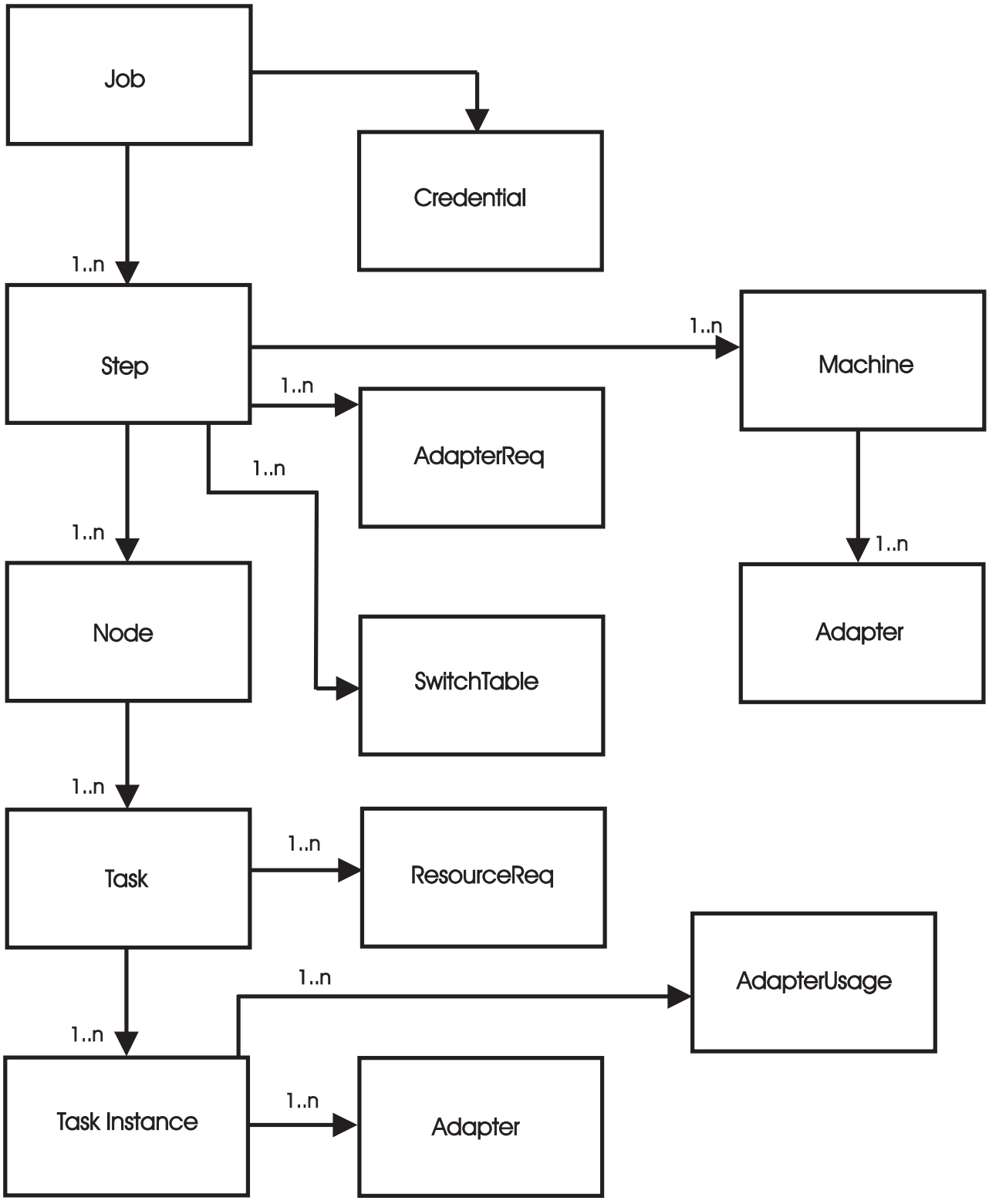

Figure 35. LoadLeveler Job Object Model

The Task object represents one or more copies of the same executable. The Task object can be queried for the executable, the executable arguments, and the number of instances of the executable.

Table 15 describes the specifications and elements available when you use the **ll_get_data** subroutine. Each specification name describes the object you need to specify and the attribute returned. For example, the specification **LL_JobGetFirstStep** includes the object you need to specify (**LL_Job**) and the value returned (**GetFirstStep**).

This table is sorted alphabetically by object; within each object the specifications are also sorted alphabetically.

When using the 2.1.0 release APi of **ll_get_data**, you must use the new 2.1 release keywords. For instance, you can not use the **min_processors** and **max_processors** from the 1.3.0 release with the 2.1 release API **ll_get_data**. You must use the new keyword, **node**.

Table 15. Specifications for ll_get_data Subroutine

Object	Specification	Resulting Data Type	Description
Adapter	LL_AdapterAvailWindowCount	int*	A pointer to an integer indicating the number of windows not in use.
Adapter	LL_AdapterCommInterface	char*	A pointer to a string containing the adapter's communication interface.
Adapter	LL_AdapterInterfaceAddress	char*	A pointer to a string containing the adapter's interface IP address.
Adapter	LL_AdapterMaxWindowSize	int*	A pointer to the integer indicating the maximum allocatable window memory.
Adapter	LL_AdapterMemory	int*	A pointer to the integer indicating the amount of total adapter memory.
Adapter	LL_AdapterMinWindowSize	int*	A pointer to the integer indicating the minimum allocatable window memory.
Adapter	LL_AdapterName	char*	A pointer to a string containing the adapter name.
Adapter	LL_AdapterTotalWindowCount	int*	A pointer to the integer indicating the number of windows on the adapter.
Adapter	LL_AdapterUsageMode	char*	A pointer to a string containing the mode used for css IP or US.
Adapter	LL_AdapterUsageProtocol	char*	A pointer to a string containing the task's protocol.
Adapter	LL_AdapterUsageWindow	char*	A pointer to a string containing the window assigned to the task.
Adapter	LL_AdapterUsageWindowMemory	char*	A pointer to the integer indicating the number of bytes used by the window.
AdapterReq	LL_AdapterReqCommLevel	int*	A pointer to the integer indicating the adapter's communication level.
AdapterReq	LL_AdapterReqUsage	char*	A pointer to a string containing the requested adapter usage.
Cluster	LL_ClusterGetFirstResource	LL_element*	A pointer to the element associated with the first resource.
Cluster	LL_ClusterGetNextResource	LL_element*	A pointer to the element associated with the next resource.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Cluster	LL_ClusterDefinedResources	char**	A pointer to an array containing the names of consumable resources defined in the cluster. The array ends with a NULL string.
Cluster	LL_ClusterDefinedResourceCount	int*	A pointer to an integer indicating the number of consumable resources defined in the cluster.
Cluster	LL_ClusterSchedulingResources	char**	A pointer to an array containing the names of consumable resources considered by the scheduler for the cluster. The array ends with a NULL string.
Cluster	LL_ClusterSchedulingResourceCount	int*	A pointer to an integer indicating the number of consumable resources considered by the scheduler for the cluster.
Credential	LL_CredentialGid	int*	A pointer to an integer containing the UNIX gid of the user submitting the job.
Credential	LL_CredentialGroupName	char*	A pointer to a string containing the UNIX group name of the user submitting the job.
Credential	LL_CredentialUid	int*	A pointer to an integer containing the UNIX uid of the person submitting the job.
Credential	LL_CredentialUserName	char*	A pointer to a string containing the user ID of the user submitting the job.
Job	LL_JobCredential	LL_element*	A pointer to the element associated with the job credential.
Job	LL_JobGetFirstStep	LL_element*	A pointer to the element associated with the first step of the job, to be used in subsequent **ll_get_data** calls.
Job	LL_JobGetNextStep	LL_element*	A pointer to the element associated with the next step.
Job	LL_JobName	char*	A pointer to a character string containing the job name.
Job	LL_JobStepCount	int*	A pointer to an integer indicating the number of steps connected to the job.
Job	LL_JobStepType	int*	A pointer to an integer indicating the type of job, which can be INTERACTIVE_JOB or BATCH_JOB.
Job	LL_JobSubmitHost	char*	A pointer to a character string containing the name of the host machine from which the job was submitted.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Job	LL_JobSubmitTime	time_t*	A pointer to the time_t structure indicating when the job was submitted.
Job	LL_JobVersionNum	int*	A pointer to an integer indicating the job's version number
Machine	LL_MachineAdapterList	char**	A pointer to an array containing the list of adapters associated with the machine. The array ends with a NULL string.
Machine	LL_MachineArchitecture	char*	A pointer to a string containing the machine architecture.
Machine	LL_MachineAvailableClassList	char**	A pointer to an array containing the currently available job classes defined on the machine. The array ends with a NULL string.
Machine	LL_MachineConfiguredClassList	char**	A pointer to an array containing the initiators on the machine. The array ends with a NULL string.
Machine	LL_MachineCPUs	int*	A pointer to an integer containing the number of CPUs on the machine.
Machine	LL_MachineDisk	int*	A pointer to an integer indicating the disk space in KBs on the machine.
Machine	LL_MachineFeatureList	char**	A pointer to an array containing the features defined on the machine. The array ends with a NULL string.
Machine	LL_MachineFreeRealMemory	int*	A pointer to an integer indicating the amount of free real memory in MBs on the machine.
Machine	LL_MachineGetFirstAdapter	LL_element*	A pointer to the element associated with the machine's first adapter.
Machine	LL_MachineGetFirstResource	LL_element*	A pointer to the element associated with the machine's first resource.
Machine	LL_MachineGetNextAdapter	LL_element*	A pointer to the element associated with the machine's next adapter.
Machine	LL_MachineGetNextResource	LL_element*	A pointer to the element associated with the machine's next resource.
Machine	LL_MachineKbddIdle	int*	A pointer to an integer indicating the number of seconds since the kbdd daemon detected keyboard mouse activity.
Machine	LL_MachineLoadAverage	double*	A pointer to a double containing the load average on the machine.
Machine	LL_MachineMaxTasks	int*	A pointer to an integer indicating the maximum number of tasks this machine can run at one time.
Machine	LL_MachineMachineMode	char*	A pointer to a string containing the configured machine mode.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Machine	LL_MachineName	char*	A pointer to a string containing the machine name.
Machine	LL_MachineOperatingSystem	char*	A pointer to a string containing the operating system on the machine.
Machine	LL_MachinePagesFreed	int*	A pointer to an integer indicating the number of pages freed per second by the page replacement algorithm.
Machine	LL_MachinePagesPagedIn	int*	A pointer to an integer indicating the number of pages paged in per second from paging space.
Machine	LL_MachinePagesPagedOut	int*	A pointer to an integer indicating the number of pages paged out per second to paging space.
Machine	LL_MachinePagesScanned	int*	A pointer to an integer indicating the number of pages scanned per second by the page replacement algorithm.
Machine	LL_MachinePoolList	int**	A pointer to an array indicating the pool numbers to which this machine belongs. The size of the array can be determined by using LL_MachinePoolListSize.
Machine	LL_MachinePoolListSize	int*	A pointer to an integer indicating the numbers of pools configured for the machine.
Machine	LL_MachineRealMemory	int*	A pointer to an integer indicating the physical memory in MBs on the machine.
Machine	LL_MachineScheddRunningJobs	int*	A pointer to an integer indicating a list of the running jobs assigned to schedd.
Machine	LL_MachineScheddState	int*	A pointer to an integer indicating the machine's schedd state.
Machine	LL_MachineScheddTotalJobs	int*	A pointer to an integer indicating the total number of jobs assigned to the schedd.
Machine	LL_MachineSpeed	double*	A pointer to a double containing the configured speed of the machine.
Machine	LL_MachineStartdRunningJobs	int*	A pointer to an integer containing the number of running jobs known by the startdd daemon.
Machine	LL_MachineStartdState	char*	A pointer to a string containing the state of the startdd daemon.
Machine	LL_MachineStepList	char**	A pointer to an array containing the steps running on the machine. The array ends with a NULL string.
Machine	LL_MachineTimeStamp	time_t*	A pointer to a time_t structure indicating the time the machine last reported to the negotiator.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Machine	LL_MachineVirtualMemory	int*	A pointer to an integer indicating the free swap space in KBs on the machine.
Node	LL_NodeGetFirstTask	LL_element*	A pointer to the element associated with the first task for this node.
Node	LL_NodeGetNextTask	LL_element*	A pointer to the element associated with the next task for this node.
Node	LL_NodeInitiatorCount	int*	A pointer to an integer indicating the number of tasks running on the node.
Node	LL_NodeMaxInstances	int*	A pointer to an integer indicating the maximum number of machines requested.
Node	LL_NodeMinInstances	int*	A pointer to an integer indicating the minimum number of machines requested.
Node	LL_NodeRequirements	char*	A pointer to a string containing the node requirements.
Node	LL_NodeTaskCount	int*	A pointer to an integer indicating the different types of tasks running on the node.
Resource	LL_ResourceAvailableValue	int*	A pointer to an integer indicating the value of available resources.
Resource	LL_ResourceName	char*	A pointer to a string containing the resource name.
Resource	LL_ResourceInitialValue	int*	A pointer to an integer indicating the initial resource value.
ResourceReq	LL_ResourceRequirementName	char*	A pointer to a string containing the resource requirement name.
ResourceReq	LL_ResourceRequirementValue	int*	A pointer to an integer indicating the value of the resource requirement.
Step	LL_StepAccountNumber	char*	A pointer to a string containing the account number specified by the user submitting the job.
Step	LL_StepComment	char*	A pointer to a string indicating the comment specified by the user submitting the job.
Step	LL_StepCompletionCode	int*	A pointer to an integer indicating the completion code of the step.
Step	LL_StepCompletionDate	time_t*	A pointer to a time_t structure indicating the completion date of the step.
Step	LL_StepCoreLimitHard	int*	A pointer to an integer indicating the core hard limit set by the user in the **core_limit** keyword.
Step	LL_StepCoreLimitSoft	int*	A pointer to an integer indicating the core soft limit set by the user in the **core_limit** keyword.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Step	LL_StepCpuLimitHard	int*	A pointer to an integer indicating the CPU hard limit set by the user in the **cpu_limit** keyword.
Step	LL_StepCpuLimitSoft	int*	A pointer to an integer indicating the CPU soft limit set by the user in the **cpu_limit** keyword.
Step	LL_StepCpuStepLimitHard	int*	A pointer to an integer indicating the CPU step hard limit set by the user in the **job_cpu_limit** keyword.
Step	LL_StepCpuStepLimitSoft	int*	A pointer to an integer indicating the CPU step soft limit set by the user in the **job_cpu_limit** keyword.
Step	LL_StepDataLimitHard	int*	A pointer to an integer indicating the data hard limit set by the user in the **data_limit** keyword.
Step	LL_StepDataLimitSoft	int*	A pointer to an integer indicating the data soft limit set by the user in the **data_limit** keyword.
Step	LL_StepDispatchTime	time_t*	A pointer to a time_t structure indicating the time the negotiator dispatched the job.
Step	LL_StepEnvironment	char*	A pointer to a string containing the environment variables set by the user in the executable.
Step	LL_StepErrorFile	char*	A pointer to a string containing the standard error file name used by the executable.
Step	LL_StepExecSize	int*	A pointer to an integer indicating the executable size.
Step	LL_StepFileLimitHard	int*	A pointer to an integer indicating the file hard limit set by the user in the **file_limit** keyword.
Step	LL_StepFileLimitSoft	int*	A pointer to an integer indicating the file soft limit set by the user in the **file_limit** keyword.
Step	LL_StepGetFirstAdapterReq	LL_element*	A pointer to the element associated with the first adapter requirement.
Step	LL_StepGetFirstMachine	LL_element*	A pointer to the element associated with the first machine in the step.
Step	LL_StepGetFirstNode	LL_element*	A pointer to the element associated with the first node of the step.
Step	LL_StepGetFirstSwitchTable	LL_element*	A pointer to the element associated with the first switch table for this step.
Step	LL_StepGetMasterTask	LL_element*	A pointer to the element associated with the master task of the step.
Step	LL_StepGetNextAdapterReq	LL_element*	A pointer to the element associated with the next adapter requirement.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Step	LL_StepGetNextMachine	LL_element*	A pointer to the element associated with the next machine of the step.
Step	LL_StepGetNextNode	LL_element*	A pointer to the element associated with the next node of the step.
Step	LL_StepGetNextSwitchTable	LL_element*	A pointer to the element associated with the next switch table for this step.
Step	LL_StepHoldType	int*	A pointer to an integer indicating the hold state of the step (user, system, etc). The value returned is in the HoldType enum.
Step	LL_StepHostList	char**	A pointer to an array containing the list of hosts in the **host.list** file associated with the step. The array ends with a null string.
Step	LL_StepID	char*	A pointer to a string containing the ID of the step.
Step	LL_StepImageSize	int*	A pointer to an integer indicating the image size of the executable.
Step	LL_StepInputFile	char*	A pointer to a string containing the standard input file name used by the executable.
Step	LL_StepIwd	char*	A pointer to a string containing the initial working directory name used by the executable.
Step	LL_StepJobClass	char*	A pointer to a string containing the class of the step.
Step	LL_StepLoadLevelerGroup	char*	A pointer to a string containing the name of the LoadLeveler group specified by the step.
Step	LL_StepMachineCount	int*	A pointer to an integer indicating the number of machines assigned to the step.
Step	LL_StepMessages	char*	A pointer to a string containing a list of messages from LL
Step	LL_StepName	char*	A pointer to a string containing the name of the step.
Step	LL_StepNodeCount	int*	A pointer to an integer indicating the number of node objects associated with the step.
Step	LL_StepNodeUsage	int*	A pointer to an integer indicating the node usage specified by the user, which can be SHARED or NOT_SHARED.
Step	LL_StepOutputFile	char*	A pointer to a character string containing the standard output file name used by the executable.
Step	LL_StepParallelMode	int*	A pointer to an integer indicating the mode of the step.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Step	LL_StepPriority	int*	A pointer to an integer indicating the priority of the step.
Step	LL_StepRssLimitHard	int*	A pointer to an integer indicating the RSS hard limit set by the user in the **rss_limit** keyword.
Step	LL_StepRssLimitSoft	int*	A pointer to an integer indicating the RSS soft limit set by the user in the **rss_limit** keyword.
Step	LL_StepShell	char*	A pointer to a character string containing the shell name used by the executable.
Step	LL_StepStackLimitHard	int*	A pointer to an integer indicating the stack hard limit set by the user in the **stack_limit** keyword.
Step	LL_StepStackLimitSoft	int*	A pointer to an integer indicating the stack soft limit set by the user in the **stack_limit** keyword.
Step	LL_StepStartCount	int*	A pointer to an integer indicating the number of times the step has been started.
Step	LL_StepStartDate	time_t*	A pointer to a time_t structure indicating the value the user specified in the **startdate** keyword.
Step	LL_StepState	int*	A pointer to an integer indicating the state of the Step (Idle, Pending, Starting, etc.) The value returned is in the StepState enum.
Step	LL_StepTaskInstanceCount	int*	A pointer to an integer indicating the number of task instances in the step. This is only available from the schedd daemon.
Step	LL_StepWallClockLimitHard	int*	A pointer to an integer indicating the wall clock hard limit set by the user in the **wall_clock_limit** keyword.
Step	LL_StepWallClockLimitSoft	int*	A pointer to an integer indicating the wall clock soft limit set by the user in the **wall_clock_limit** keyword.
Task	LL_TaskExecutable	char*	A pointer to a string containing the name of the executable.
Task	LL_TaskExecutableArguments	char*	A pointer to a string containing the arguments passed by the user in the executable.
Task	LL_TaskGetFirstResourceRequirement	LL_element	A pointer to the element associated with the first resource requirement.
Task	LL_TaskGetFirstTaskInstance	LL_element*	A pointer to the element associated with the first task instance.
Task	LL_TaskGetNextResourceRequirement	LL_element*	A pointer to the element associated with the next resource requirement.

Table 15. Specifications for ll_get_data Subroutine (continued)

Object	Specification	Resulting Data Type	Description
Task	LL_TaskIsMaster	int*	A pointer to an integer indicating whether this is the master task.
Task	LL_TaskTaskInstanceCount	int*	A pointer to an integer indicating the number of task instances.
Task	LL_TaskGetNextTaskInstance	LL_element*	A pointer to the element associated with the next task instance.
Task Instance	LL_TaskInstanceAdapterCount	int*	A pointer to the integer indicating the number of adapters.
Task Instance	LL_TaskInstanceGetFirstAdapter	LL_element*	A pointer to the element associated with the first adapter.
Task Instance	LL_TaskInstanceGetFirstAdapterUsage	LL_element*	A pointer to the element associated with the first adapter usage.
Task Instance	LL_TaskInstanceGetNextAdapter	LL_element*	A pointer to the element associated with the next adapter.
Task Instance	LL_TaskInstanceGetNextAdapterUsage	LL_element*	A pointer to the element associated with the next adapter usage.
Task Instance	LL_TaskInstanceMachineName	char*	A pointer to the string indicating the machine assigned to a task.
Task Instance	LL_TaskInstanceTaskID	int*	A pointer to the integer indicating the task ID.

ll_get_data Subroutine

Before you use this subroutine, make sure you are familiar with “Understanding the LoadLeveler Job Object Model” on page 262.

Purpose

The **ll_get_data** subroutine returns data from a valid **LL_element**.

Library

LoadLeveler API library **libllapi.a**

Syntax

```
#include "llapi.h"

int ll_get_data(LL_element *element, enum LLAPI_Specification specification,
void* resulting_data_type);
```

Parameters

element
Is a pointer to the **LL_element** returned by the **ll_get_objs** subroutine or by the **ll_get_data** subroutine. For example: Job, Machine, Step, etc.

specification
Specifies the data field within the data object you want to read.

resulting_data_type
Is a pointer to where you want the data stored. If this parameter is equal to NULL, then an error has occurred and the value could not be stored.

Description

object and *specification* are input fields, while *resulting_data_type* is an output field.

The **ll_get_data** subroutine of the data access API allows you to access LoadLeveler objects. The parameters of **ll_get_data** are a LoadLeveler object (**LL_element**), a specification that indicates what information about the object is being requested, and a pointer to the area where the information being requested should be stored.

If the specification indicates an attribute of the element that is passed in, the result pointer must be the address of a variable of the appropriate type, and must be initialized to NULL. The type returned by each specification is found in Table 15 on page 264. If the specification queries the connection to another object, the returned value is of type **LL_element**. You can use a subsequent **ll_get_data** call to query information about the new object.

The data type **char*** and any arrays of type **int** or **char** must be freed by the caller.

LL_element pointers cannot be freed by the caller.

When using the 2.1.0 release APi of **ll_get_data**, you must use the new 2.1 release keywords. For instance, you can not use the **min_processors** and **max_processors** from the 1.3.0 release with the 2.1 release API **ll_get_data**. You must use the new keyword, **node**.

Return Values
This subroutine returns a zero to indicate success.

Error Values
-1 You specified an invalid *object*.
-2 You specified an invalid LLAPI_Specification.

Related Information
Subroutines: **ll_query**, **ll_set_request**, **ll_reset_request**, **ll_get_objs**, **lL_next_obj**, **ll_free_objs**, **ll_deallocate**.

ll_next_obj Subroutine

Purpose
The **ll_next_obj** subroutine returns the next object in the *query_element* list you specify.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

LL_element * ll_next_obj(LL_element *query_element);
```

Parameters
query_element
Is a pointer to the **LL_element** returned by the **ll_query** function.

Description
query_element is the input field for this subroutine.

Use this subroutine in conjunction with the **ll_get_objs** subroutine to “loop” through the list of objects queried.

Return Values
This subroutine returns a pointer to the next object in the list.

Error Values
NULL Indicates an error or the end of the list of objects.

Related Information
Subroutines: **ll_get_data**, **ll_set_request**, **ll_query**, **ll_get_objs**, **ll_free_objs**, **ll_deallocate**.

ll_free_objs Subroutine

Purpose
The **ll_free_objs** subroutine frees all of the **LL_element** objects in the *query_element* list that were obtained by the **ll_get_objs** subroutine. You must free the *query_element* by using the **ll_deallocate** subroutine.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_free_objs(LL_element *query_element);
```

Parameters
query_element
Is a pointer to the **LL_element** returned by the **ll_query** function.

Description
query_element is the input field for this subroutine.

Return Values
This subroutine returns a zero to indicate success.

Error Values
-1 You specified an invalid *query_element*.

Related Information
Subroutines: **ll_get_data**, **ll_set_request**, **ll_query**, **ll_get_objs**, **ll_reset_request**, **ll_free_objs**.

ll_deallocate Subroutine

Purpose
The **ll_deallocate** subroutine deallocates the *query_element* allocated by the **ll_query** subroutine.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_deallocate(LL_element *query_element);
```

Parameters

query_element
Is a pointer to the **LL_element** returned by the **ll_query** function.

Description
query_element is the input field for this subroutine.

Return Values
This subroutine returns a zero to indicate success.

Error Values
-1 You specified an invalid *query_element*.

Related Information
Subroutines: **ll_get_data**, **ll_set_request**, **ll_query**, **ll_get_objs**, **ll_reset_request**, **ll_next_obj**, **ll_free_objs**.

Examples of Using the Data Access API

Example 1: The following example shows how LoadLeveler's Data Access API can be used to obtain machine, job, and cluster information. The program consists of three steps:

1. Getting information about selected hosts in the LoadLeveler cluster
2. Getting information about jobs of selected classes
3. Getting floating consumable resource information in the LoadLeveler cluster

```
#include <stdio.h>
#include "llapi.h"

main(int argc, char *argv[])
{
   LL_element *queryObject, *machine, *resource, *cluster;
   LL_element *job, *step, *node, *task, *credential, *resource_req;
   int rc, obj_count, err_code, value;
   double load_avg;
   enum StepState step_state;
   char **host_list, **class_list;
   char *name, *res_name, *step_id, *job_class, *node_req;
   char *task_exec, *ex_args, *startd_state;

   /* Step 1: Display information of selected machines in the LL cluster */

   /* Initialize the query: Machine query */
   queryObject = ll_query(MACHINES);
   if (!queryObject) {
      printf("Query MACHINES: ll_query() returns NULL.\n"); exit(1);
   }

   /* Set query parameters: query specific machines by name */
   host_list = (char **)malloc(3*sizeof(char *));
   host_list[0] = "c163n12.ppd.pok.ibm.com";
   host_list[1] = "c163n11.ppd.pok.ibm.com";
   host_list[2] = NULL;
   rc = ll_set_request(queryObject, QUERY_HOST, host_list, ALL_DATA);
   if (rc) {
      printf("Query MACHINES: ll_set_request() return code is non-zero.\n"); exit(1);
   }

   /* Get the machine objects from the LoadL_negotiator (central manager) daemon */
   machine = ll_get_objs(queryObject, LL_CM, NULL, &obj_count, &err_code);
   if (machine == NULL) {
      printf("Query MACHINES: ll_get_objs() returns NULL. Error code = %d\n", err_code);
   }
```

```
printf("Number of machines objects returned = %d\n", obj_count);

/* Process the machine objects */
while(machine) {
   rc = ll_get_data(machine, LL_MachineName, &name);
   if (!rc) {
      printf("Machine name: %s ------------------\n", name); free(name);
   }
   rc = ll_get_data(machine, LL_MachineStartdState, &stard_state);
   if (rc) {
      printf("Query MACHINES: ll_get_data() return code is non-zero.\n"); exit(1);
   }
   printf("Startd State: %s\n", startd_state);
   if (strcmp(startd_state, "Down") != 0) {
      rc = ll_get_data(machine, LL_MachineRealMemory, &value);
      if (!rc) printf("Total Real Memory: %d\n", value);
      rc = ll_get_data(machine, LL_MachineVirtualMemory, &value);
      if (!rc) printf("Free Swap Space: %d\n", value);
      rc = ll_get_data(machine, LL_MachineLoadAverage, &load_avg);
      if (!rc) printf("Load Average: %f\n", load_avg);
   }
   free(startd_state);
  /* Consumable Resources associated with this machine */
   resource = NULL;
   ll_get_data(machine, LL_MachineGetFirstResource, &resource);
   while(resource) {
      rc = ll_get_data(resource, LL_ResourceName, &res_name);
      if (!rc) {printf("Resource Name = %s\n", res_name); free (res_name);}
      rc = ll_get_data(resource, LL_ResourceInitialValue, &value);
      if (!rc) printf("   Total: %d\n", value);
      rc = ll_get_data(resource, LL_ResourceAvailableValue, &value);
      if (!rc) printf("   Available: %d\n", value);
      resource = NULL;
      ll_get_data(machine, LL_MachineGetNextResource, &resource);
   }
   machine = ll_next_obj(queryObject);
}

/* Free objects obtained from Negotiator */
ll_free_objs(queryObject);
/* Free query element */
ll_deallocate(queryObject);

/* Step 2: Display information of selected jobs */

/* Initialize the query: Job query  */
queryObject = ll_query(JOBS);
if (!queryObject) {
   printf("Query JOBS: ll_query() returns NULL.\n");
   exit(1);
}

/* Query all class "Parallel" and "No_Class"  jobs submitted to c163n11, c163n12 */
class_list = (char **)malloc(3*sizeof(char *));
class_list[0] = "Parallel";
class_list[1] = "No_Class";
class_list[2] = NULL;
rc = ll_set_request(queryObject, QUERY_HOST, host_list, ALL_DATA);
if (rc) {printf("Query JOBS: ll_set_request() return code is  non-zero.\n"); exit(1);}
rc = ll_set_request(queryObject, QUERY_CLASS, class_list, ALL_DATA);
if (rc) {printf("Query JOBS: ll_set_request() return code is  non-zero.\n"); exit(1);}

/* Get the requested job objects from the Central Manager */
job = ll_get_objs(queryObject, LL_CM, NULL, &obj_count, &err_code);
if (job == NULL) {
   printf("Query JOBS: ll_get_objs() returns NULL. Error code = %d\n", err_code);
}
```

```
printf("Number of job objects returned = %d\n", obj_count);

/*  Process the job objects and display selected information of each job step.
 *
 *  Notes:
 *    1. Since LL_element is defined as "void" in llapi.h, when using
 *       ll_get_data it is important that a valid "specification"
 *       parameter be used for a given "element" argument.
 *    2. Checking of return code is not always made in the following
 *        loop to minimize the length of the listing.
 */

while(job) {
   rc = ll_get_data(job, LL_JobName, &name);
   if (!rc) {printf("Job name: %s\n", name); free(name);}

   rc = ll_get_data(job, LL_JobCredential, &credential);
   if (!rc) {
      rc = ll_get_data(credential, LL_CredentialUserName, &name);
      if (!rc) {printf("Job owner: %s\n", name); free(name);}
      rc = ll_get_data(credential, LL_CredentialGroupName, &name);
      if (!rc) {printf("Unix Group: %s\n", name); free(name);}
   }
   step = NULL;
   ll_get_data(job, LL_JobGetFirstStep, &step);
   while(step) {
      rc = ll_get_data(step, LL_StepID, &step_id);
      if (!rc) {printf("  Step ID: %s\n", step_id); free(step_id);}
      rc = ll_get_data(step, LL_StepJobClass, &job_class);
      if (!rc) {printf("  Step Job Class: %s\n", job_class); free(job_class);}
      rc = ll_get_data(step, LL_StepState, &step_state);
      if (!rc) {
         if (step_state == STATE_RUNNING) {
            printf("  Step Status: Running\n");
            printf("  Allocated Hosts:\n");
            machine = NULL;
            ll_get_data(step, LL_StepGetFirstMachine, &machine);
            while(machine) {
               rc = ll_get_data(machine, LL_MachineName, &name);
               if (!rc) { printf("    %s\n", name); free(name); }
               machine = NULL;
               ll_get_data(step, LL_StepGetNextMachine, &machine);
            }
         }else {
            printf("  Step Status: Not Running\n");
         }
      }
      node = NULL;
      ll_get_data(step, LL_StepGetFirstNode, &node);
      while(node) {
         rc = ll_get_data(node, LL_NodeRequirements, &node_req);
         if (!rc) {printf("    Node Requirements: %s\n", node_req); free(node_req);}
         task = NULL;
         ll_get_data(node, LL_NodeGetFirstTask, &task);
         while(task) {
            rc = ll_get_data(task, LL_TaskExecutable, &task_exec);
            if (!rc) {printf("      Task Executable: %s\n", task_exec); free(task_exec);}
            rc = ll_get_data(task, LL_TaskExecutableArguments, &ex_args);
            if (!rc) {printf("      Task Executable Arguments: %s\n",ex_args);
            free(ex_args);}
            resource_req = NULL;
            ll_get_data(task, LL_TaskGetFirstResourceRequirement, &resource_req);
            while(resource_req) {
               rc = ll_get_data(resource_req, LL_ResourceRequirementName, &name);
               if (!rc) {printf("        Resource Req Name: %s\n", name); free(name);}
               rc = ll_get_data(resource_req, LL_ResourceRequirementValue, &value);
```

```
                  if (!rc) {printf("        Resource Req Value: %d\n", value);}
                  resource_req = NULL;
                  ll_get_data(task, LL_TaskGetNextResourceRequirement, &resource_req);
               }
               task = NULL;
               ll_get_data(node, LL_NodeGetNextTask, &task);
            }
            node = NULL;
            ll_get_data(step, LL_StepGetNextNode, &node);
         }
         step = NULL;
         ll_get_data(job, LL_JobGetNextStep, &step);
      }
      job = ll_next_obj(queryObject);
   }
   ll_free_objs(queryObject);
   ll_deallocate(queryObject);

   /* Step 3: Display Floating Consumable Resources information of LL cluster. */

   /* Initialize the query: Cluster query  */
   queryObject = ll_query(CLUSTERS);
   if (!queryObject) {
      printf("Query CLUSTERS: ll_query() returns NULL.\n");
      exit(1);
   }
   ll_set_request(queryObject, QUERY_ALL, NULL, ALL_DATA);
   cluster = ll_get_objs(queryObject, LL_CM, NULL, &obj_count, &err_code);
   if (!cluster) {
      printf("Query CLUSTERS: ll_get_objs() returns NULL. Error code = %d\n", err_code);
   }
   printf("Number of Cluster objects = %d\n", obj_count);
   while(cluster) {
      resource = NULL;
 ll_get_data(cluster, LL_ClusterGetFirstResource, &resource);
      while(resource) {
         rc = ll_get_data(resource, LL_ResourceName, &res_name);
         if (!rc) {printf("Resource Name = %s\n", res_name); free(res_name);}
         rc = ll_get_data(resource, LL_ResourceInitialValue, &value);
         if (!rc) {printf("Resource Initial Value = %d\n", value);}
         rc = ll_get_data(resource, LL_ResourceAvailableValue, &value);
         if (!rc) {printf("Resource Available Value = %d\n", value);}
         resource = NULL;
         ll_get_data(cluster, LL_ClusterGetNextResource, &resource);
      }
      cluster = ll_next_obj(queryObject);
   }
   ll_free_objs(queryObject);
   ll_deallocate(queryObject);
}
```

Parallel Job API

If you are using any of the parallel operating environments already supported by LoadLeveler, you do not have to use the parallel API. However, if you have another application environment that you want to use, you need to use the subroutines described here to interface with LoadLeveler.

The parallel job API consists of two subroutines. **ll_get_hostlist** acquires the list of LoadLeveler selected parallel nodes, and **ll_start_host** starts the parallel task under the LoadLeveler starter.

The following section describes how parallel job submission works. Understanding this will help you to better understand the parallel API.

Interaction Between LoadLeveler and the Parallel API

This API does not give you access to any new LoadLeveler functions from Version 2 Release 1.0, or later releases.

Program applications which use the parallel APIs to interface with LoadLeveler are supported under a job type called **parallel**. When a user submits a job specifying the keyword **job_type** equal to parallel, the LoadLeveler API job control flow is as follows:

The negotiator selects nodes based on the resources you request. Once the nodes have been obtained, the negotiator contacts the schedd to start the job. The schedd marks the job pending and contacts the affected startds to start their starter processes.

One machine becomes the **Master Starter**. The Master Starter is one of the selected parallel nodes. After all starters are started and have completed inititialization, the Master Starter starts the executable specified in the job command file. The executable referred to as the **Parallel Master** uses this API to start tasks on remote nodes. A LOADLBATCH environment variable is set to YES so that the Parallel Master can distinguish between callers.

The Parallel Master must:

- Obtain the machine list through the **ll_get_hostlist** API.
- Start a task on all allocated machines through the **ll_start_host** API. It is mandatory that one and only one task be started on each machine. Each task is considered a Parallel Slave. Acquiring the task name, path and arguments is the responsibility of the Parallel Master. The user may pass this information through the **arguments** or **environment** keywords in the job command file.

When the Parallel Master starts, the job is marked Running. Once the Parallel Master and all tasks exit, the job is marked Complete.

Termination Paths

The Parallel Master is expected to cleanup and exit when:

- All of the Parallel Slaves have exited.
- A negative value is returned by either the **ll_get_hostlist** or **ll_start_host** subroutine.
- A SIGCONT, followed by a SIGTERM, is received. A possible reason for this is that LoadLeveler receives a job cancel request.

  The SIGTERM is also sent to all parallel tasks.
- A SIGCONT, followed by a SIGUSR1, is received. Reasons for this include:
  - The Parallel Master receives a VACATE or FLUSH request.
  - LoadLeveler receives a stop LoadLeveler daemons command.

  The SIGUSR1 is also sent to all parallel tasks.

A SIGKILL is issued to any process which does not exit within two minutes of receiving a termination signal.

Note that a SIGUSER1 indicates the job must terminate but will be restarted, while a SIGTERM indicates the job must terminate but will not be restarted.

ll_get_hostlist Subroutine

Purpose
This subroutine obtains a list of machines from the Master Starter machine so that the Parallel Master can start the Parallel Slaves. The Parallel Master is the LoadLeveler executable specified in the job command file and the Parallel Slaves are the processes started by the Parallel Master through the **ll_start_host** API.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
int ll_get_hostlist(struct JM_JOB_INFO* jobinfo);
```

Parameters
jobinfo is a pointer to the JM_JOB_INFO structure defined in **llapi.h**. No fields are required to be filled in. **ll_get_hostlist** allocates storage for an array of JM_NODE_INFO structures and returns the pointer in the *jm_min_node_info pointer*. It is the caller's responsibility to free this storage.

```
struct JM_JOB_INFO {
   int  jm_request_type;
   char  jm_job_description[50];
   enum  JM_ADAPTER_TYPE jm_adapter_type;
   int  jm_css_authentication;
   int jm_min_num_nodes;
   struct JM_NODE_INFO *jm_min_node_info;
};
struct JM_NODE_INFO {
   char jm_node_name [MAXHOSTNAMELEN];
   char jm_node_address [50];
   int jm_switch_node_number;
   int  jm_pool_id;
   int  jm_cpu_usage;
   int  jm_adapter_usage;
   int  jm_num_virtual_tasks;
   int  *jm_virtual_task_ids;
   enum  JM_RETURN_CODE jm_return_code;
};
```

The following data is filled in for the JM_JOB_INFO structure:

jm_min_num_nodes
: Is the number of elements in the array of JM_NODE_INFO structures. It is the number of hosts allocated to a job.

jm_min_node_info
: Is the pointer to the array of JM_NODE_INFO structures. The first entry in this array describes the node which is mapped to task 0. The second entry is mapped to task 1, and so on.

The following data is filled in for each JM_NODE_INFO structure:

jm_node_name
: Is the name of the node.

jm_node_address
: Is the address corresponding to the adapter requested.

jm_switch_node_number
: Is the relative node number set only for job running on the SP switch adapter. For all other jobs it is set to -1.

Description
The Parallel Master must:

- Issue error messages as appropriate.
- Exit when **ll_get_hostlist** returns with a negative return value. The Parallel Master exit status is included in the job mail returned to the user.

Return Values
This subroutine returns a zero to indicate success.

Error Values

-2 Cannot get LoadLeveler step ID from environment.

-5 Cannot make socket. This means that the UNIX stream socket could not be created. This socket is needed to establish communications with the starter for both of the API's functions.

-6 Cannot connect to host.

-8 Cannot get hostlist.

ll_start_host Subroutine

Purpose
This subroutine starts a task on a selected machine.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
int ll_start_host(char *host, char *start_cmd);
```

Parameters

host
Is the name of the node on which you want to start the task.

start_cmd
Is the actual command to execute on the node, including flags and arguments.

Description
This function must be invoked for all the machines returned from the **ll_get_hostlist** subroutine once and only once by the Parallel Master. Acquiring the **start_cmd** is the responsibility of the Parallel Master. The user may pass this information through the **arguments** or **environment** keywords in the job command file.

The Parallel Master must:

- Issue error messages as appropriate.
- Exit when **ll_start_host** returns a negative value. The Parallel Master exit status is included in the job mail returned to the user.

Return Values
This subroutine returns an integer greater than one to indicate the socket connected to the Parallel Slave's standard I/O (stdio).

Error Values

-2 Cannot get LoadLeveler step ID from environment

-4 Nameserver cannot resolve host

-6 Cannot connect to host

-7 Cannot send PASS_OPEN_SOCKET command to remote startd

-9 The command you specified failed.

Examples

A sample program called **para_api.c** is provided in the **samples/llpara** subdirectory of the release directory, usually **/usr/lpp/LoadL/full**.

In order to run this example, you need to do the following:

1. Copy the sample Makefile and the sample program called **para_api.c** to your home directory.
2. Update the **startCmd** variable in **para_api.c** to reflect your home directory versus **/usr/lpp/LoadL/full/samples/llpara**. For example:

   ```
   char *startCmd = "/home/user/para_api -s";
   ```
3. Issue **make** to create the executable **para_api**.
4. Update your job command file as follows:

```
#!/bin/ksh
# @ initialdir     = /home/user
# @ executable     = para_api
# @ output         = para_api.$(cluster).$(process).out
# @ error          = para_api.$(cluster).$(process).err
# @ job_type       = parallel
# @ min_processors = 2
# @ max_processors = 2
# @ queue
```

5. Submit the job command file to LoadLeveler.

   The syntax to invoke the Parallel Master is:

   ```
   para_api
   ```

   The syntax to invoke the Parallel Slave is:

   ```
   para_api -s
   ```

   The Parallel Master does the following:

   - Acquires the hostlist through the **ll_get_hostlist** API and prints out the returned fields.
   - Starts a Parallel Slave task by executing the command specified in the **StartCmd** variable on all hosts returned in the hostlist.
   - Acquires the socket connected to the Parallel Slave's standard I/O (stdio).
   - Writes a command over the socket to verify stdin.
   - Reads acknowledgments over the socket to verify stderr and stdout.
   - Prints out host names and acknowledgments.

   Example output follows:

   ```
   num_nodes=2

   name=host1.kgn.ibm.com address=9.115.8.162 switch_number=-1

   name=host2.kgn.ibm.com address=9.115.8.164 switch_number=-1

   Connected to host1.kgn.ibm.com at sock 3
   Received acko "8000" and acke "10000" from host 0
   ```

```
Connected to host2.kgn.ibm.com at sock 4
Received acko "8001" and acke "10001" from host 1

<Master Exiting>
```

The Parallel Slave does the following:
- Reads command from stdin.
- Writes acknowledgment to stdout and stderr.

Workload Management API

The workload management API consists of three subroutines, **ll_control**, **ll_start_job**, and **ll_terminate_job**. The ll_control subroutine can be used to perform most of the LoadLeveler control operations and is designed for general use. The ll_start_job, and ll_terminate_job subroutines are intended to be used in conjunction with an external scheduler.

To use an external scheduler, you must specify the following keyword in the global LoadLeveler configuration file:

```
SCHEDULER_API = YES
```

Specifying YES disables the default LoadLeveler scheduling algorithm. When you disable the default LoadLeveler scheduler, jobs do not start unless requested to do so by the ll_start_job subroutine.

You can toggle between the default LoadLeveler scheduler and an external scheduler.

If you are running the default LoadLeveler scheduler, this is how you can switch to an external scheduler:
1. In the configuration file, set **SCHEDULER_API = YES**
2. On the central manager machine, issue the **llctl -g stop** and then **llctl -g start** commands

If you are running an external scheduler, this is how you can re-enable the LoadLeveler scheduling algorithm:
1. In the configuration file, set **SCHEDULER_API = NO**
2. On the central manager machine, issue the **llctl -g stop** and then **llctl -g start** commands

Note that the ll_start_job and ll_terminate_job subroutines automatically connect to an alternate central manager if they cannot contact the primary central manager.

An example of an external scheduler you can use is the Extensible Argonne Scheduling sYstem (EASY), developed by Argonne National Laboratory and available as public domain code.

You should use ll_start_job and ll_terminate_job in conjuction with the query API. The query API collects information regarding which machines are available and which jobs need to be scheduled. See “Query API” on page 291 for more information.

ll_control Subroutine

Purpose
This subroutine allows an application program to perform most of the functions that are currently available through the standalone commands: **llctl**, **llfavorjob**, **llfavoruser**, **llhold**, and **llprio**.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_control(int control_version, enum LL_control_op control_op,
char **host_list,char **user_list, char **job_list, char **class_list,
int priority);
```

Parameters
int *control_version*
An integer indicating the version of **ll_control** being used. This argument should be set to **LL_CONTROL_VERSION**.

enum *LL_control_op*
The control operation to be performed. The enum **LL_control_op** is defined in **llapi.h** as:

```
enum LL_control_op {
LL_CONTROL_RECYCLE, LL_CONTROL_RECONFIG, LL_CONTROL_START, LL_CONTROL_STOP,
LL_CONTROL_DRAIN, LL_CONTROL_DRAIN_STARTD, LL_CONTROL_DRAIN_SCHEDD,
LL_CONTROL_PURGE_SCHEDD, LL_CONTROL_FLUSH, LL_CONTROL_SUSPEND,
LL_CONTROL_RESUME, LL_CONTROL_RESUME_STARTD, LL_CONTROL_RESUME_SCHEDD,
LL_CONTROL_FAVOR_JOB, LL_CONTROL_UNFAVOR_JOB, LL_CONTROL_FAVOR_USER,
LL_CONTROL_UNFAVOR_USER, LL_CONTROL_HOLD_USER, LL_CONTROL_HOLD_SYSTEM,
LL_CONTROL_HOLD_RELEASE, LL_CONTROL_PRIO_ABS, LL_CONTROL_PRIO_ADJ };
```

char ***host_list*
A NULL terminated array of host names.

char ***user_list*
A NULL terminated array of user names.

char ***job_list*
A NULL terminated array of job names. The job name that an element of job_list points to is a character string with one of the following formats: *"host.jobid.stepid," "jobid.stepid," "jobid"*. *host* is the name of the machine to which the job was submitted (the default is the local machine), *jobid* is the job ID assigned to the job by LoadLeveler, and *stepid* is the job step ID assigned to a job step by LoadLeveler (the default is to include all the steps of a job).

char ***class_list*
A NULL terminated array of class names.

int *priority*
An integer representing the new absolute value of user priority or adjustment to the current user priority of job steps.

Description
The **ll_control** subroutine performs operations that are essentially equivalent to those performed by the standalone commands: **llctl**, **llfavorjob**, **llfavoruser**, **llhold**,

and **llprio**. Because of this similarity, descriptions of the **ll_control** operations are grouped according to the standalone command they resemble.

llctl type of operations: These are the ll_control operations which mirror operations performed by the llctl command. This summary includes a brief description of each of the allowed llctl types of operations. For more information on the llctl command, see "llctl - Control LoadLeveler Daemons" on page 175.

LL_CONTROL_START:
Starts the LoadLeveler daemons on the specified machines. The calling program must have rsh privileges to start LoadLeveler daemons on remote machines.

LL_CONTROL_STOP:
Stops the LoadLeveler daemons on the specified machines.

LL_CONTROL_RECYCLE:
Stops, and then restarts, all of the LoadLeveler daemons on the specified machines.

LL_CONTROL_RECONFIG:
Forces all of the LoadLeveler daemons on the specified machines to reread the configuration files.

LL_CONTROL_DRAIN:
When this operation is selected, the following happens: (1) No LoadLeveler jobs can start running on the specified machines, and (2) No LoadLeveler jobs can be submitted to the specified machines.

LL_CONTROL_DRAIN_SCHEDD:
No LoadLeveler jobs can be submitted to the specified machines.

LL_CONTROL_DRAIN_STARTD:
Keeps LoadLeveler jobs from starting on the specified machines. If a *class_list* is specified, then the classes specified will be drained (made unavailable). The literal string ″**allclasses**″ can be used as an abbreviation for all of the classes.

LL_CONTROL_FLUSH:
Terminates running jobs on the specified machines and send them back to the negotiator to await redispatch (if restart=yes).

LL_CONTROL_PURGE_SCHEDD:
Purges the specified schedd host's job queue; a *host_list* consisting of one host name must be specified.

LL_CONTROL_SUSPEND:
Suspends all jobs on the specified machines. This operation is not supported for parallel jobs.

LL_CONTROL_RESUME:
Resumes job submission to, and job execution on, the specified machines.

LL_CONTROL_RESUME_STARTD:
Resumes job execution on the specified machines; if a *class_list* is specified, then execution of jobs associated with these classes is resumed.

LL_CONTROL_RESUME_SCHEDD:
Resumes job submission to the specified machines.

For these **llctl** type of operations, the *user_list*, *job_list*, and *priority* arguments are not used and should be set to **NULL** or zero. The *class_list* argument is meaningful only if the operation is LL_CONTROL_DRAIN_STARTD, or LL_CONTROL_RESUME_STARTD. If *class_list* is not being used, then it should be

set to **NULL**. If *host_list* is NULL, then the scope of the operation is all machines in the LoadLeveler cluster. Unlike the standalone **llctl** command, where the scope of the operation is either global or one host, **ll_control** operations allow the user to specify a list of hosts (through the *host_list* argument). To perform these operations, the calling program must have LoadLeveler administrator authority.

llfavorjob type of operations: The **llfavorjob** type of control operations are: **LL_CONTROL_FAVOR_JOB**, and **LL_CONTROL_UNFAVOR_JOB**. For these operations, the *user_list*, *host_list*, *class_list*, and *priority* arguments are not used and should be set to **NULL** or zero. LL_CONTROL_FAVOR_JOB is used to set specified job steps to a higher system priority thatn all job steps that are not favored. LL_CONTROL_UNFAVOR_JOB is used to unfavor previously favored job steps, restoring the original priorities. The calling program must have LoadLeveler administrator authority to perform these operations.

llfavoruser type of operations: The **llfavoruser** type of control operations are: **LL_CONTROL_FAVOR_USER**, and **LL_CONTROL_UNFAVOR_USER**. For these operations, the *host_list*, *job_list*, *class_list*, and *priority* arguments are not used and should be set to **NULL** or zero. LL_CONTROL_FAVOR_USER sets jobs of one or more users to the highest priority in the system, regardless of the current setting. Jobs already running are not affected. LL_CONTROL_UNFAVOR_USER is used to unfavor previously favored user's jobs, restoring the original priorities. The calling program must have LoadLeveler administrator authority to perform these operations.

llhold type of operations: The **llhold** type of control operations are: **LL_CONTROL_HOLD_USER**, **LL_CONTROL_HOLD_SYSTEM**, and **LL_CONTROL_HOLD_RELEASE**. For these operations, the *class_list* and *priority* arguments are not used, and should be set to **NULL** or zero. LL_CONTROL_HOLD_USER and LL_CONTROL_HOLD_SYSTEM place jobs in user hold and system hold, respectively. LL_CONTROL_HOLD_RELEASE is used to release jobs from both types of hold. The calling program must have LoadLevler administrator authority to put jobs into system hold, and to release jobs from system hold. If a job is in both user and system holds then the LL_CONTROL_HOLD_RELEASE operation must be performed twice to release the job from both types of hold. If the user is not a LoadLeveler administrator then the **llhold** types of operations have no effect on jobs that do not belong to him/her.

llprio type of operations: The **llprio** type of control operations are: **LL_CONTROL_PRIO_ABS**, and **LL_CONTROL_PRIO_ADJ**. For these operations, the *user_list*, *host_list*, and *class_list* arguments are not used, and should be set to **NULL**. **llprio** type of operations change the user priority of one or more job steps in the LoadLeveler queue. LL_CONTROL_PRIO_ABS specifies a new absolute value of the user priority, and LL_CONTROL_PRIO_ADJ specifies an adjustment to the current user priority. The valid range of LoadLeveler user priorities is 0–100 (inclusive); 0 is the lowest possible priority, and 100 is the highest. The **llprio** type of operations have no effect on a running job step unless this job step returns to **Idle** state. If the user is not a LoadLeveler administrator, then an **llprio** type of operation has no effect on jobs that do not belong to him/her.

Return Values

0 The specified command has been sent to the appropriate LoadLeveler daemon.

-2 The specified command cannot be sent to the central manager.

-3 The specified command cannot be sent to one of the **LoadL_master** daemons.

-4 ll_control encountered an error while processing the administration or configuration file.

-6 A data transmission failure has occurred.

-7 The calling program does not have LoadLeveler administrator authority.

-19 An incorrect **ll_control** version has been specified.

-20 A system error has occurred.

-21 The system cannot allocate memory.

-22 An invalid **control_op** operation has been specified.

-23 The **job_list** argument contains one or more errors.

-24 The **host_list** argument contains one or more errors.

-25 The **user_list** argument contains one or more errors.

-26 Incompatible arguments have been specified for **HOLD** operation.

-27 Incompatible arguments have been specified for **PRIORITY** operation.

-28 Incompatible arguments have been specified for **FAVORJOB** operation.

-29 Incompatible arguments have been specified for **FAVORUSER** operation.

-30 An error occurred while ll_control tried to start a child process.

-31 An error occurred while ll_control tried to start the **LoadL_master** daemon.

-32 An error occurred while ll_control tried to execute the **llpurgeschedd** command.

-33 The **class_list** argument contains incompatible information.

-34 ll_control cannot create a file in the **/tmp** directory.

-35 LoadLeveler has encountered miscellaneous incompatible input specifications.

Related Information
Commands: **llprio**, **llhold**, **llfavoruser**,**llfavorjob**,**llctl**.

ll_start_job Subroutine

Purpose
This subroutine tells the LoadLeveler negotiator to start a job on the specified nodes.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_start_job(LL_start_job_info *ptr);
```

Parameters
ptr Specifies the pointer to the **LL_start_job_info** structure that was allocated by the caller. The **LL_start_job_info** members are:

int *version_num*
Represents the version number of the **LL_start_job_info** structure. Should be set to LL_PROC_VERSION

LL_STEP_ID *StepId*
Represents the step ID of the job step to be started.

**char ****nodeList*
Is a pointer to an array of node names where the job will be started. The first member of the array is the parallel master node. The array must be ended with a NULL.

Description

You must set **SCHEDULER_API = YES** in the global configuration file to use this subroutine.

Only jobs steps currently in the Idle state are started.

Only processes having the LoadLeveler administrator user ID can invoke this subroutine.

An external scheduler uses this subroutine in conjunction with the **ll_get_nodes** and **ll_get_jobs** subroutines of the query API. The query API returns information about which machines are avialable for scheduling and which jobs are currently in the job queue waiting to be scheduled.

Return Values

This subroutines return a value of zero to indicate the start job request was accepted by the negotiator. However, a return code of zero does not necessarily imply the job started. You can use the **llq** command to verify the job started. Otherwise, this subroutine returns an integer value defined in the **llapi.h** file.

Error Values

-1 There is an error in the input parameter.

-2 The subroutine cannot connect to the central manager.

-4 An error occurred reading parameters from the administration or the configuration file.

-5 The negotiator cannot find the specified *StepId* in the negotiator job queue.

-6 A data transmission failure occurred.

-7 The subroutine cannot authorize the action because you are not a LoadLeveler administrator.

-8 The job object version number is incorrect.

-9 The *StepId* is not in the Idle state.

-10 One of the nodes specified is not available to run the job.

-11 One of the nodes specified does not have an available initiator for the class of the job.

-12 For one of the nodes specified, the requirements statement does not satisfy the job requirements.

-13 The number of nodes specified was less than the minimum or more than the maximum requested by the job.

-14 The LoadLeveler default scheduler is enabled; that is, **SCHEDULING_API=NO**.

-15 The same node was specified twice in **ll_start_job** *nodeList*.

Examples
Makefiles and examples which use this subroutine are located in the **samples/llsch** subdirectory of the release directory. The examples include the executable **sch_api**, which invokes the query API and the job control API to start the second job in the list received from **ll_get_jobs** on two nodes. You should submit at least two jobs prior to running the sample. To compile **sch_api**, copy the sample to a writeable directory and update the RELEASE_DIR field to represent the current LoadLeveler release directory.

Related Information
Subroutines: **ll_get_jobs**, **ll_terminate_job**, **ll_get_nodes**.

ll_terminate_job Subroutine

Purpose
This subroutine tells the negotiator to cancel the specified job step.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_terminate_job(LL_terminate_job_info *ptr);
```

Parameters
ptr Specifies the pointer to the **LL_terminate_info** structure that was allocated by the caller. The **LL_terminate_job_info** members are:

int *version_num*
Represents the version number of the **LL_terminate_job_info** structure. Should be set to LL_PROC_VERSION

LL_STEP_ID *StepId*
Represents the step ID of the job step to be terminated.

char **msg*
A pointer to a null terminated array of characters. If this pointer is null or points to a null string, a default message is used. This message will be available through **ll_get_data** to tell the process why a program was terminated.

Description
You do not need to disable the default LoadLeveler scheduler in order to use this subroutine.

Only processes having the LoadLeveler administrator user ID can invoke this subroutine.

An external scheduler uses this subroutine in conjunction with the **ll_get_job** subroutine (of the job control API) and **ll_start_jobs** subroutine (of the query API). The external scheduler must use this subroutine to return errors from **ll_start_job** to interactive parallel jobs.

Return Values

This subroutine returns a value of zero when successful, to indicate the terminate job request was accepted by the negotiator. However, a return code of zero does not necessarily imply the negotiator cancelled the job. Use the **llq** command to verify the job was cancelled. Otherwise, this subroutine returns an integer value defined in the **llapi.h** file.

Error Values

-1 There is an error in the input parameter.

-4 An error occurred reading parameters from the administration or the configuration file.

-6 A data transmission failure occurred.

-7 The subroutine cannot authorize the action because you are not a LoadLeveler administrator or you are not the user who submitted the job.

-8 The job object version number is incorrect.

Examples

Makefiles and examples which use this subroutine are located in the **samples/llsch** subdirectory of the release directory. The examples include the executable **sch_api**, which invokes the query API and the job control API to terminate the first job reported by the **ll_get_jobs** subroutine. You should submit at least two jobs prior to running the sample. To compile **sch_api**, copy the sample to a writeable directory and update the RELEASE_DIR field to represent the current LoadLeveler release directory.

Related Information

Subroutines: **ll_get_jobs**, **ll_start_job**, **ll_get_nodes**.

Usage Notes

It is important to know how LoadLeveler keywords and commands behave when you disable the default LoadLeveler scheduling algorithm. LoadLeveler scheduling keywords and commands fall into the following categories:

- Keywords not involved in scheduling decisions are unchanged.
- Keywords kept in the job object or in the machine which are used by the LoadLeveler default scheduler have their values maintained as before and passed to the query API.
- Keywords used only by the LoadLeveler default scheduler have no effect.

The following sections discuss some specific keywords and commands and how they behave when you disable the default LoadLeveler scheduling algorithm.

Job Command File Keywords

class – This value is provided by the query APIs. Machines chosen by **ll_start_job** *must* have the class of the job available or the request will be rejected.

dependency – Supported as before. Job objects for which dependency cannot be evaluated (because a previous step has not run) are maintained in the NotQueued state, and attempts to start them via **ll_start_job** will result in an error. If the dependency is met, **ll_start_job** can start the proc.

hold – **ll_start_job** cannot start a job that is in Hold status.

min_processors – **ll_start_job** must specify at least this number of processors.

max_processors – **ll_start_job** must specify no more than this number of processors.

preferences – Passed to the query API.
requirements – **ll_start_job** returns an error if the machine(s) specified do not match the requirements of the job. This includes Disk and Virtual Memory requirements.
startdate – The job remains in the Deferred state until the **startdate** specified in the job is reached. **ll_start_job** cannot start a job in the Deferred state.
user_priority – Used in calculating the system priority (as described in “How Does a Job’s Priority Affect Dispatching Order?” on page 28). The system priority assigned to the job is available through the query API. No other control of the order in which jobs are run is enforced.

Administration File Keywords

master_node_exclusive is ignored.
master_node_requirement is ignored.
maxidle is supported.
maxjobs is ignored.
maxqueued is supported.
max_jobs_scheduled is ignored.
priority is used to calculate the system priority (where appropriate).
speed is available through the query API.

Configuration File Keywords

MACHPRIO is calculated but is not used.
SYSPRIO is calulated and available to the query API.
MAX_STARTERS is calculated, and if starting the job causes this value to be exceeded, **ll_start_job** returns an error.
NEGOTIATOR_PARALLEL_DEFER is ignored.
NEGOTIATOR_PARALLEL_HOLD is ignored.
NEGOTIATOR_RESCAN_QUEUE is ignored.
NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL works as before. Set this value to 0 if you do not want the SYSPRIOs of job objects recalulated.

Query API

This API provides information about the jobs and machines in the LoadLeveler cluster. You can use this in conjuction with the job control API, since the job control API requires you to know which machines are available and which jobs need to be scheduled. See “Workload Management API” on page 283 for more information.

The query API consists of the following subroutines: **ll_get_jobs**, **ll_free_jobs**, **ll_get_nodes**, and **ll_free_nodes**.

ll_get_jobs Subroutine

Purpose

This subroutine, available to any user, returns information about all jobs in the LoadLeveler job queue.

Library

LoadLeveler API library **libllapi.a**

Syntax

```
#include "llapi.h"

int ll_get_jobs(LL_get_jobs_info *);
```

Parameters

ptr Specifies the pointer to the **LL_get_jobs_info** structure that was allocated by the caller. The **LL_get_jobs_info** members are:

int *version_num*
Represents the version number of the **LL_start_job_info** structure. This should be set to LL_PROC_VERSION.

int *numJobs*
Represents the number of entries in the array.

**LL_job ****_JobList_
Represents the pointer to an array of **LL_job** structures. The **LL_job** structure is defined in **llapi.h**.

Description

The **LL_get_jobs_info** structure contains an array of **LL_job** structures indicating each job in the LoadLeveler system.

Some job information, such as the start time of the job, is not available to this API. (It is recommended that you use the dispatch time, which is available, in place of the start time.) Also, some accounting information is not available to this API.

Return Values

This subroutines returns a value of zero when successful. Otherwise, it returns an integer value defined in the **llapi.h** file.

Error Values

-1 There is an error in the input parameter.

-2 The API cannot connect to the central manager.

-3 The API cannot allocate memory.

-4 A configuration error occurred.

Examples

Makefiles and examples which use this subroutine are located in the **samples/llsch** subdirectory of the release directory.

Related Information

Subroutines: **ll_free_jobs**, **ll_free_nodes**, **ll_get_nodes**.

ll_free_jobs Subroutine

Purpose

This subroutine, available to any user, frees storage that was allocated by **ll_get_jobs**.

Library

LoadLeveler API library **libllapi.a**

Syntax

```
#include "llapi.h"

int ll_free_jobs(LL_get_jobs_info *ptr);
```

Parameters

ptr Specifies the address of the **LL_get_jobs_info** structure to be freed.

Description
This subroutine frees the storage pointed to by the **LL_get_jobs_info** pointer.

Return Values
This subroutines returns a value of zero when successful. Otherwise, it returns an integer value defined in the **llapi.h** file.

Error Values

-8 The *version_num* member of the **LL_get_jobs_info** structure did not match the current version.

Examples
Makefiles and examples which use this subroutine are located in the **samples/llsch** subdirectory of the release directory.

Related Information
Subroutines: **ll_get_jobs**, **ll_free_nodes**, **ll_get_nodes**.

ll_get_nodes Subroutine

Purpose
This subroutine, available to any user, returns information about all of nodes known by the negotiator daemon.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_get_nodes(LL_get_nodes_info *ptr);
```

Parameters

ptr Specifies the pointer to the **LL_get_nodes_info** structure that was allocated by the caller. The **LL_get_nodes_info** members are:

int *version_num*
Represents the version number of the **LL_start_job_info** structure.

int *numNodes*
Represents the number of entries in the *NodeList* array.

**LL_node ****NodeList*
Represents the pointer to an array of **LL_node** structures. The **LL_node** structure is defined in **llapi.h**.

Description
The **LL_get_node_info** structure contains an array of **LL_job** structures indicating each node in the LoadLeveler system.

Return Values
This subroutines returns a value of zero when successful. Otherwise, it returns an integer value defined in the **llapi.h** file.

Error Values

-1 There is an error in the input parameter.

-2 The API cannot connect to the central manager.

-3 The API cannot allocate memory.

-4 A configuration error occurred.

Examples
Makefiles and examples which use this subroutine are located in the **samples/llsch** subdirectory of the release directory.

Related Information
Subroutines: **ll_free_jobs**, **ll_free_nodes**, **ll_get_jobs**.

ll_free_nodes Subroutine

Purpose
This subroutine, available to any user, frees storage that was allocated by **ll_get_nodes**.

Library
LoadLeveler API library **libllapi.a**

Syntax
```
#include "llapi.h"

int ll_nodes_jobs(LL_get_nodes_info *ptr);
```

Parameters
ptr Specifies the address of the **LL_get_nodes_info** structure to be freed.

Description
This subroutine frees the storage pointed to by the **LL_get_nodes_info** pointer.

Return Values
This subroutines returns a value of zero when successful. Otherwise, it returns an integer value defined in the **llapi.h** file.

Error Values
-8 The *version_num* member of the **LL_get_jobs_info** structure did not match the current version.

Examples
Makefiles and examples which use this subroutine are located in the **samples/llsch** subdirectory of the release directory.

Related Information
Subroutines: **ll_get_jobs**, **ll_free_nodes**, **ll_get_nodes**.

User Exits

This section discusses separate user exits for the following:

- Handling DCE security credentials
- Handling an AFS token
- Filtering a job script
- Overriding the default mail notification method

Handling DCE Security Credentials

You can write a pair of programs to override the default LoadLeveler DCE authentication method. To enable the programs, use the following keyword in your configuration file:

DCE_AUTHENTICATION_PAIR = *program1, program2*
Where *program1* and *program2* are LoadLeveler or installation supplied programs that are used to authenticate DCE security credentials. *program1* obtains a handle (an opaque credentials object), at the time the job is submitted, which is used to authenticate to DCE. *program2* is the path name of a LoadLeveler or an installation supplied program that uses the handle obtained by *program1* to authenticate to DCE before starting the job on the executing machine(s).

An example of a credentials object is a character string containing the DCE principle name and a password. *program1* writes the following to standard output:

- The length of the handle to follow
- The handle

If *program1* encounters errors, it writes error messages to standard error.

program2 receives the following as standard input:

- The length of the handle to follow
- The same handle written by *program1*

program2 writes the following to standard output:

- The length of the login context to follow
- An exportable DCE login context, which is the idl_byte array produced from the sec_login_export_context DCE API call. For more information, see the DCE Security Services API chapter in the Distributed Computing Environment for AIX Application Development Reference.
- A character string suitable for assigning to the KRB5CCNAME environment variable This string represents the location of the credentials cache established in order for *program2* to export the DCE login context.

If *program2* encounters errors, it writes error messages to standard error. The parent process, the LoadLeveler starter process, writes those messages to the starter log.

Usage Notes

If you are using DCE on AIX 4.3, you need the proper DCE credentials for the existing authentication method in order to run a command or function that uses **rshell** (**rsh**). Otherwise, the **rshell** command may fail. You can use the **lsauthent** command to determine the authentication method. If **lsauthent** indicates that DCE authentication is in use, you must log in to DCE wth the **dce_login** command to obtain the proper credentials.

LoadLeveler commands that run **rshell** include **llctl version** and **llctl start**.

For examples of programs that enable DCE security credentials, see the **/samples/lldce** subdirectory in the release directory.

Handling an AFS Token

You can write a program, run by the scheduler, to refresh an AFS token when a job is started. To invoke the program, use the following keyword in your configuration file:

AFS_GETNEWTOKEN = *myprog*
where *myprog* is a filter that receives the AFS authentication information on

standard input and writes the new information to standard output. The filter is run when the job is scheduled to run and can be used to refresh a token which expired when the job was queued.

Before running the program, LoadLeveler sets up standard input and standard output as pipes between the program and LoadLeveler. LoadLeveler also sets up the following environment variables:

LOADL_STEP_OWNER
The owner (UNIX user name) of the job

LOADL_STEP_COMMAND
The name of the command the user's job step invokes.

LOADL_STEP_CLASS
The class this job step will run.

LOADL_STEP_ID
The step identifier, generated by LoadLeveler.

LOADL_JOB_CPU_LIMIT
The number of CPU seconds the job is limited to.

LOADL_WALL_LIMIT
The number of wall clock seconds the job is limited to.

LoadLeveler writes the following current AFS credentials, in order, over the standard input pipe:
The **ktc_principal** structure indicating the service.
The **ktc_principal** structure indicating the client.
The **ktc_token** structure containing the credentials.

The ktc_principal structure is defined in the AFS header file **afs_rxkad.h**. The ktc_token structure is defined in the AFS header file **afs_auth.h**.

LoadLeveler expects to read these same structures in the same order from the standard output pipe, except these should be refreshed credentials produced by the user exit.

The user exit can modify the passed credentials (to extend their lifetime) and pass them back, or it can obtain new credentials. LoadLeveler takes whatever is returned and uses it to authenticate the user prior to starting the user's job.

Filtering a Job Script

You can write a program to filter a job script when the job is submitted. This program can, for example, modify defaults or perform site specific verification of parameters. To invoke the program, specify the following keyword in your configuration file:

SUBMIT_FILTER = *myprog*
where *myprog* is called with the job file as the standard input. The standard output is submitted to LoadLeveler. If the program returns with a non-zero exit code, the job submission is cancelled.

The following environment variables are set when the program is invoked:
LOADL_ACTIVE
LoadLeveler version
LOADL_STEP_COMMAND
Job command file name

LOADL_STEP_ID
The job identifier, generated by LoadLeveler
LOADL_STEP_OWNER
The owner (UNIX user name) of the job

Using Your Own Mail Program

You can write a program to override the LoadLeveler default mail notification method. You can use this program to, for example, display your own messages to users when a job completes, or to automate tasks such as sending error messages to a network manager.

The syntax for the program is the same as it is for standard UNIX mail programs; the command is called with a list of users as arguments, and the mail message is taken from standard input. This syntax is as follows:

MAIL = *program*
where *program* specifies the path name of a local program you want to use.

Writing Prolog and Epilog Programs

An administrator can write *prolog* and *epilog* user exits that can run before and after a LoadLeveler job runs, respectively.

Prolog and epilog programs fall into two categories: those that run as the LoadLeveler user ID, and those that run in a user's environment.

To specify prolog and epilog programs, specify the following keywords in the configuration file:

JOB_PROLOG = *pathname*
where *pathname* is the full path name of the prolog program. This program runs under the LoadLeveler user ID.

JOB_EPILOG = *pathname*
where *pathname* is the full path name of the epilog program. This program runs under the LoadLeveler user ID.

JOB_USER_PROLOG = *pathname*
where *pathname* is the full path name of the user prolog program. This program runs under the user's environment.

JOB_USER_EPILOG = *pathname*
where *pathname* is the full path name of the user epilog program. This program runs under the user's environment.

A user environment prolog or epilog runs with AFS and/or DCE authentification (if either is installed and enabled). For security reasons, you must code these programs on the machines where the job runs *and* on the machine that schedules the job. If you do not define a value for these keywords, the user enviroment prolog and epilog settings on the executing machine are ignored.

The user environment prolog and epilog can set environment variables for the job by sending information to standard output in the following format:

```
env id = value
```

Where:

id Is the name of the environment variable

value Is the value (setting) of the environment variable

For example, the user environment prolog below sets the environment variable **STAGE_HOST** for the job:

```
#!/bin/sh
echo env STAGE_HOST=shd22
```

Prolog Programs

The prolog program is invoked by the starter process. Once the starter process invokes the prolog program, the program obtains information about the job from environment variables.

Syntax:

prolog_program

Where *prolog_program* is the name of the prolog program as defined in the JOB_PROLOG keyword.

No arguments are passed to the program, but several environment variables are set. For more information on these environment variables, see “Run-time Environment Variables” on page 57.

The real and effective user ID of the prolog process is the LoadLeveler user ID. If the prolog program requires root authority, the administrator must write a secure C or perl program to perform the desired actions. You should *not* use shell scripts with set uid permissions, since these scripts may make your system susceptible to security problems.

Return Code Values:

0 The job will begin.

If the prolog program is killed, the job does not begin and a message is written to the starter log.

Sample Prolog Programs:

Sample of a Prolog Program for Korn Shell:

```
#!/bin/ksh
#
# Set up environment
set -a
. /etc/environment
. /.profile
export PATH="$PATH:/loctools/lladmin/bin"
export LOG="/tmp/$LOADL_STEP_OWNER.$LOADL_JOB_ID.prolog"
#
# Do set up based upon job step class
#
case $LOADL_STEP_CLASS in
    # A OSL job is about to run, make sure the osl filesystem is
    # mounted. If status is negative then filesystem cannot be
    # mounted and the job step should not run.
    "OSL")
      mount_osl_files >> $LOG
    if [ status = 0 ]
        then EXIT_CODE=1
      else
        EXIT_CODE=0
      fi
```

```
      ;;
# A simulation job is about to run, simulation data has to
# be made available to the job. The status from copy script must
# be zero or job step cannot run.
"sim")
      copy_sim_data >> $LOG
if [ status = 0 ]
        then EXIT_CODE=0
      else
        EXIT_CODE=1
      fi
      ;;
# All other job will require free space in /tmp, make sure
# enough space is available.
*)
      check_tmp >> $LOG
      EXIT_CODE=$?
      ;;
esac
# The job step will run only if EXIT_CODE == 0
exit $EXIT_CODE
```

Sample of a Prolog Program for C Shell:

```
#!/bin/csh
#
# Set up environment
source /u/loadl/.login
#
setenv PATH  "${PATH}:/loctools/lladmin/bin"
setenv LOG "/tmp/${LOADL_STEP_OWNER}.${LOADL_JOB_ID}.prolog"
#
# Do set up based upon job step class
#
switch ($LOADL_STEP_CLASS)
    # A OSL job is about to run, make sure the osl filesystem is
    # mounted. If status is negative then filesystem cannot be
    # mounted and the job step should not run.
    case "OSL":
      mount_osl_files >> $LOG
      if ($status < 0 ) then
        set EXIT_CODE = 1
      else
        set EXIT_CODE = 0
      endif
      breaksw
# A simulation job is about to run, simulation data has to
# be made available to the job. The status from copy script must
# be zero or job step cannot run.
case "sim":
    copy_sim_data >> $LOG
    if ($status == 0 ) then
      set EXIT_CODE = 0
    else
      set EXIT_CODE = 1
    endif
    breaksw
# All other job will require free space in /tmp, make sure
# enough space is available.
default:
    check_tmp >> $LOG
    set EXIT_CODE = $status
    breaksw
endsw

# The job step will run only if EXIT_CODE == 0
exit $EXIT_CODE
```

Epilog Programs

The installation defined epilog program is invoked after a job step has completed. The purpose of the epilog program is to perform any required clean up such as unmounting file systems, removing files, and copying results. The exit status of both the prolog program and the job step is set in environment variables.

Syntax:

epilog_program

Where *epilog_program* is the name of the epilog program as defined in the JOB_EPILOG keyword.

No arguments are passed to the program but several environment variables are set. These environment variables are described in “Run-time Environment Variables” on page 57. In addition, the following environment variables are set for the epilog programs:

LOADL_PROLOG_EXIT_CODE
: The exit code from the prolog program. This environment variable is set only if a prolog program is configured to run.

LOADL_USER_PROLOG_EXIT_CODE
: The exit code from the user prolog program. This environment variable is set only if a user prolog program is configured to run.

LOADL_JOB_STEP_EXIT_CODE
: The exit code from the job step.

Note: To interpret the exit status of the prolog program and the job step, convert the string to an integer and use the structures found in the **sys/wait.h** file.

Sample Epilog Programs:

Sample of an Epilog Program for Korn Shell:

```
#!/bin/ksh
#
# Set up environment
set -a
. /etc/environment
. /.profile
export PATH="$PATH:/loctools/lladmin/bin"
export LOG="/tmp/$LOADL_STEP_OWNER.$LOADL_JOB_ID.epilog"
#
if [ [ -z $LOADL_PROLOG_EXIT_CODE ] ]
then
echo "Prolog did not run" >> $LOG
else
echo "Prolog exit code = $LOADL_PROLOG_EXIT_CODE" >> $LOG
fi
#
if [ [ -z $LOADL_USER_PROLOG_EXIT_CODE ] ]
  then
   echo "User environment prolog did not run" >> $LOG
  else
   echo "User environment exit code = $LOADL_USER_PROLOG_EXIT_CODE" >> $LOG
fi
#
if [ [ -z $LOADL_JOB_STEP_EXIT_CODE ] ]
  then
   echo "Job step did not run" >> $LOG
  else
   echo "Job step exit code = $LOADL_JOB_STEP_EXIT_CODE" >> $LOG
```

```
fi
#
#
# Do clean up based upon job step class
#
case $LOADL_STEP_CLASS in
  # A OSL job just ran, unmount the filesystem.
  "OSL")
    umount_osl_files >> $LOG
    ;;
  # A simulation job just ran, remove input files.
  # Copy results if simulation was successful (second argument
  # contains exit status from job step).
  "sim")
    rm_sim_data >> $LOG
    if [ $2 = 0 ]
      then copy_sim_results >> $LOG
    fi
    ;;
# Clean up /tmp
*)
  clean_tmp >> $LOG
  ;;
esac
```

Sample of an Epilog Program for C Shell:

```
#!/bin/csh
#
# Set up environment
source /u/loadl/.login
#
setenv PATH  "${PATH}:/loctools/lladmin/bin"
setenv LOG "/tmp/${LOADL_STEP_OWNER}.${LOADL_JOB_ID}.prolog"
#
if ( ${?LOADL_PROLOG_EXIT_CODE} ) then
echo "Prolog exit code = $LOADL_PROLOG_EXIT_CODE" >> $LOG
else
echo "Prolog did not run" >> $LOG
endif
#
if ( ${?LOADL_USER_PROLOG_EXIT_CODE} ) then
    echo "User environment exit code = $LOADL_USER_PROLOG_EXIT_CODE" >> $LOG
  else
    echo "User environment prolog did not run" >> $LOG
endif
#
if ( ${?LOADL_JOB_STEP_EXIT_CODE} ) then
    echo "Job step exit code = $LOADL_JOB_STEP_EXIT_CODE" >> $LOG
  else
    echo "Job step did not run" >> $LOG
endif
#
# Do clean up based upon job step class
#
switch ($LOADL_STEP_CLASS)
  # A OSL job just ran, unmount the filesystem.
  case "OSL":
    umount_osl_files >> $LOG
    breaksw
# A simulation job just ran, remove input files.
# Copy results if simulation was successful (second argument
# contains exit status from job step).
case "sim":
  rm_sim_data >> $LOG
  if ($argv{2} == 0 ) then
    copy_sim_results >> $LOG
```

```
  endif
  breaksw
# Clean up /tmp
default:
  clean_tmp >> $LOG
  breaksw
endsw
```

Part 7. Appendixes

Appendix A. Troubleshooting

Troubleshooting LoadLeveler

This chapter is divided into the following sections:

- "Frequently Asked Questions", which contains answers to questions frequently asked by LoadLeveler customers. This section focuses on answers that may help you get out of problem situations. The questions and answers are organized into the following categories:
  - **Jobs submitted to LoadLeveler do not run.** See "Why Won't My Job Run?" for more information.
  - **One or more of your machines goes down.** See "What Happens to Running Jobs When a Machine Goes Down?" on page 308 for more information.
  - **The central manager is not operating.** See "What Happens if the Central Manager Isn't Operating?" on page 309 for more information.
  - **Miscellaneous questions.** See "Other Questions" on page 311 for more information.
- "Helpful Hints" on page 312, which contains tips on running LoadLeveler, including some productivity aids.
- "Getting Help from IBM" on page 316, which tells you how to contact IBM for assistance.

It is helpful to create error logs when you are diagnosing a problem. See to "Step 12: Record and Control Log Files" on page 113 for information on setting up error logs.

Frequently Asked Questions

This section contains answers to questions frequently asked by LoadLeveler customers.

Why Won't My Job Run?

If you submitted your job and it is in the LoadLeveler queue but has not run, issue **llq -s** first to help diagnose the problem. If you need more help diagnosing the problem, refer to the following table:

Why Your Job May Not Be Running:	Possible Soulution
Job requires specific machine, operating system, or other resource.	• Does the resource exist in the LoadLeveler cluster? If yes, wait until it becomes available. Check the GUI to compare the job requirements to the machine details, especially **Arch**, **OpSys**, and **Class**. Ensure that the spelling and capitalization matches.
Job requires specific job class	• Is the class defined in the administration file? Use **llclass** to determine this. If yes, • Is there a machine in the cluster that supports that class? If yes, you need to wait until the machine becomes available to run your job.
The maximum number of jobs are already running on all the eligible machines	Wait until one of the machines finishes a job before scheduling your job.

Why Your Job May Not Be Running:	Possible Soulution

The start expression evaluates to false.	Examine the configuration files (both **LoadL_config** and **LoadL_config.local**) to determine the **START** control function expression used by LoadLeveler to start a job. As a problem determination measure, set the START and SUSPEND values, as shown in this example:
	`START: T`
	`SUSPEND: F`

The priority of your job is lower than the priority of other jobs.	You cannot affect the system priority given to this job by the negotiator daemon but you can try to change your user priority to move this job ahead of other jobs you previously submitted using the **llprio** command or the GUI.
The information the central manager has about machines and jobs may not be current.	Wait a few minutes for the central manager to be updated and then the job may be dispatched. This time limit (a few minutes) depends upon the polling frequency and polls per update set in the **LoadL_config** file. The default polling frequency is five minutes.
You do not have the same user ID on all the machines in the cluster.	To run jobs on any machine in the cluster, you have to have the same user ID and the same uid number on every machine in the pool. If you do not have a userid on one machine, your jobs will not be scheduled to that machine.

You can use the **llq** command to query the status of your job or the **llstatus** command to query the status of machines in the cluster. Refer to “Chapter 9. LoadLeveler Commands” on page 167 for information on these commands.

Why Won’t My Parallel Job Run?

If you submitted your parallel job and it is in the LoadLeveler queue but has not run, issue **llq -s** first to help diagnose the problem. If issuing this command does not help, refer to the previous table and to the following table for more information:

Why Your Job May Not Be Running	Possible Solution
The minimum number of processors requested by your job is not available.	Sufficient resources must be available. Specifying a smaller number of processors may help if your job can run with fewer resources.
The pool in your **requirements** statement specifies a pool which is invalid or not available.	The specified pool must be valid and available.
The adapter specified in the **requirements** statement or the **network** statement identifies an adapter which is invalid or not available.	The specified adapter must be valid and available.
PVM3 is not installed	PVM3 must be installed on any machine you wish to use for pvm. The PVM3 system itself is not supplied with LoadLeveler.
You are already running a PVM3 job on one of the LoadLeveler machines.	PVM3 restrictions prevent a user from running more than one pvm daemon per user per machine. If you want to run pvm3 jobs on LoadLeveler, you must not run any pvm3 jobs outside of LoadLeveler control on any machine being managed by LoadLeveler.
The **parallel_path** keyword in your job command file is incorrect.	Use **parallel_path** to inform LoadLeveler where binaries that run your pvm tasks are for the pvm_spawn() command. If this is incorrect, the job may not run.
The **pvm_root** keyword in the administration file is incorrect.	This keyword corresponds to the pvm **ep** keyword and is required to tell LoadLeveler where the pvm system is installed.

Why Your Job May Not Be Running	Possible Solution
The file **/tmp/pvmd.***userid* exists on some LoadLeveler machine but no PVM jobs are running.	If PVM3 exits unexpectedly, it will not properly clean up after itself. Although LoadLeveler attempts to clean up after pvm, some situations are ambiguous and you may have to remove this file yourself. Check all the systems specified as being capable of running PVM3, and remove this file if it exists.

Common Set Up Problems with Parallel Jobs: This section presents a list of common problems found in setting up parallel jobs:

- If jobs appear to remain in a Pending or Starting state: check that the nameserver is consistent. Compare results of **host** *machine_name* and **host** *IP_address*
- For POE:
  - Specify the POE partition manager as the executable. Do *not* specify the parallel job as the executable.
  - Pass the parallel job as an argument to POE.
  - The parallel job must exist and must be specified as a full path name.
  - If the job runs in user space, specify the flag **-euilib us**.
  - Specify the correct adapter (when needed).
  - Specify a POE job only once in the job command file.
  - Compile only with the supported level of POE.
  - Specify only **parallel** as the *job_type*.
- For PVM:
  - Specify the parallel job as the executable. Do *not* specify PVM as the executable.
  - Compile only with the supported level of PVM.
  - Specify only **pvm3** as the *job_type*.

PVM Problem Determination: If LoadLeveler is to manage PVM jobs on a machine for a user, that user should not attempt to run PVM jobs on that machine outside of LoadLeveler control. Because of PVM restrictions, only a single PVM daemon per user per machine is permitted. If a user tries to run PVM jobs without using LoadLeveler and LoadLeveler later attempts to start a job for that user on the same machine, LoadLeveler may not be able to start PVM for the job. This will cause the LoadLeveler job to be cancelled.

If a PVM job submitted through LoadLeveler is rejected, it is probably because PVM was not correctly terminated the last time it ran on the rejecting machine. LoadLeveler attempts to handle this by making sure that it cleans up PVM jobs when they complete, but remember that you may need to clean up after the job yourself. If a machine refuses to start a PVM job, check the following:

- See if there is a process with the name **pvmd** running on the machine in question under the id of the user whose job will not start. Stop the process by issuing:

  ```
  ps -ef | grep pvmd
  kill -TERM pid
  ```

  Do not use either of the following variations to stop the daemon because this will prevent **pvmd** from cleaning up and jobs will still not start:

  ```
  kill -9 pid
  kill -KILL pid
  ```

- If there is no **pvmd** process running, see if there is a file called **/tmp/pvmd.** *userid*, where *userid* is the ID of the user whose job will not start. If the file exists, remove it.

Why Won't My Submit-Only Job Run?

If a job you submitted from a *submit-only* machine does not run, verify that you have defined the following statements in the machine stanza of the administration file of the submit-only machine:

```
submit_only = true
schedd_host = false
central_manager = false
```

Why Does a Job Stay in the Pending (or Starting) State?

If a job appears to stay in the Pending or Starting state, it is possible the job is continually being dispatched and rejected. Check the setting of the **MAX_JOB_REJECT** keyword. If it is set to the default, -1, the job will be rejected an unlimited number of times. Try resetting this keyword to some finite number. Also, check the setting of the **ACTION_ON_MAX_REJECT** keyword. These keywords are described in "Step 17: Specify Additional Configuration File Keywords" on page 129.

What Happens to Running Jobs When a Machine Goes Down?

Both the startd daemon and the schedd daemon maintain persistent states of all jobs. Both daemons use a specific protocol to ensure that the state of all jobs is consistent across LoadLeveler. In the event of a failure, the state can be recovered. Neither the schedd nor the startd daemon discard the job state information until it is passed onto and accepted by another daemon in the process.

If	Then
The network goes down but the machines are still running	If the network goes down but the machines are still running, when LoadLeveler is restarted, it looks for all jobs that were marked running when it went down. On the machine where the job is running, the startd daemon searches for the job and if it can verify that the job is still running, it continues to manage the job through completion. On the machine where schedd is running, schedd queues a transaction to the startd to re-establish the state of the job. This transaction stays queued until the state is established. Until that time, LoadLeveler assumes the state is the same as when the system went down.
The network partitions or goes down.	All transactions are left queued until the recipient has acknowledged them. Critical transactions such as those between the schedd and startd are recorded on disk. This ensures complete delivery of messages and prevents incorrect decisions based on incomplete state information.
The machine with startd goes down.	Because job state is maintained on disk in startd, when LoadLeveler is restarted it can forward correct status to the rest of LoadLeveler. In the case of total machine failure, this is usually ″JOB VACATED″, which causes the job to be restarted elsewhere. In the case that only LoadLeveler failed, it is often possible to ″find″ the job if it is still running and resume management of it. In this case LoadLeveler sends JOB RUNNING to the schedd and central manager, thereby permitting the job to run to completion.
The central manager machine goes down.	All machines in the cluster send current status to the central manager on a regular basis. When the central manager restarts, it queries each machine that checks in, requesting the entire queue from each machine. Over the period of a few minutes the central manager restores itself to the state it was in before the failure. Each schedd is responsible for maintaining the correct state of each job as it progressed while the central manager is down. Therefore, it is guaranteed that the central manager will correctly rebuild itself. All jobs started when the central manager was down will continue to run and complete normally with no loss of information. Users may continue to submit jobs. These new jobs will be forwarded correctly when the central manager is restarted.

If	Then
The schedd machine goes down	When schedd starts up again, it reads the queue of jobs and for every job which was in some sort of active state (i.e. PENDING, STARTING, RUNNING), it queries the machine where it is marked active. The running machine is required to return current status of the job. If the job completed while schedd was down, JOB COMPLETE is returned with exit status and accounting information. If the job is running, JOB RUNNING is returned. If the job was vacated, JOB VACATED is returned. Because these messages are left queued until delivery is confirmed, no job will be lost or incorrectly dispatched due to schedd failure. During the time the schedd is down, the central manager will not be able to start new jobs that were submitted to that schedd. To recover the resources allocated to jobs scheduled by a schedd machine, see “How Do I Recover Resources Allocated by a schedd Machine?” on page 311.
The **llsubmit** machine goes down	schedd gets its own copy of the executable so it does not matter if the **llsubmit** machine goes down.

Why Does llstatus Indicate that a Machine is Down when llq Indicates a Job is Running on The Machine?: If a machine fails while a job is running on the machine, the central manager does not change the status of any job on the machine. When the machine comes back up the central manager will be updated.

What Happens if the Central Manager Isn’t Operating?

In one of your machine stanzas specified in the administration file, you specified a machine to serve as the central manager. It is possible for some problem to cause this central manager to become unusable such as network communication or software or hardware failures. In such cases, the other machines in the LoadLeveler cluster believe that the central manager machine is no longer operating. If you assigned one or more alternate central managers in the machine stanza, a new central manager will take control. The alternate central manager is chosen based upon the order in which its respective machine stanza appears in the administration file.

Once an alternate central manager takes control, it starts up its negotiator daemon and notifies all of the other machines in the LoadLeveler cluster that a new central manager has been selected. The following diagram illustrates how a machine can become the alternate central manager:

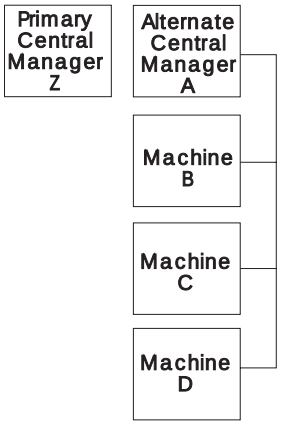

Figure 36. When the Primary Central Manager is Unavailable

The diagram illustrates that Machine Z is the primary central manager but Machine A took control of the LoadLeveler cluster by becoming the alternate central manager. Machine A remains in control as the alternate central manager until either:

- The primary central manager, Machine Z, resumes operation. In this case, Machine Z notifies Machine A that it is operating again and, therefore, Machine A terminates its negotiator daemon.
- Machine A also loses contact with the remaining machines in the pool. In this case, another machine authorized to serve as an alternate central manager takes control. Note that Machine A may remain as its own central manager.

The following diagram illustrates how multiple central managers can function within the same LoadLeveler pool:
In this diagram, the primary central manager is serving Machines A and B. Due to

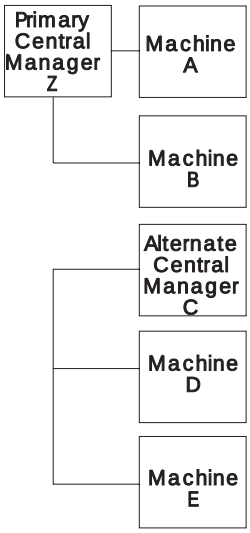

Figure 37. Multiple Central Managers

some network failure, Machines C, D, and E have lost contact with the primary central manager machine and, therefore, Machine C which is authorized to serve as an alternate central manager, assumes that role. Machine C remains as the alternate central manager until either:

- The primary central manager is able to contact Machines C, D, and E. In this case, the primary central manager notifies the alternate central managers that it is operating again and, therefore, Machine C terminates its negotiator daemon.

The negotiator daemon running on the primary central manager machine is refreshed to discard any old job status information and to pick up the new job status information from the newly re-joined machines.

- Machine C loses contact with Machines D and E. In this case, if machine D or E is authorized to act as an alternate central manager, it assumes that role. Otherwise, there will be no central manager serving these machines. Note that Machine C remains as its own central manager.

While LoadLeveler can handle this situation of two concurrent central managers without any loss of integrity, some installations may find administering it somewhat confusing. To avoid any confusion, you should specify all primary and alternate central managers on the same LAN segment.

For information on selecting alternate central managers, refer to “Step 1: Specify Machine Stanzas” on page 75.

How Do I Recover Resources Allocated by a schedd Machine?

If a node running the schedd daemon fails, resources allocated to jobs scheduled by this schedd cannot be freed up until you restart the schedd. Administrators must do the following to enable the recovery of schedd resources:

1. Recognize that a node running the schedd daemon is down and will be down long enough such that it is necessary for you to recover the schedd resources.
2. Add the statement **schedd_fenced=true** to the machine stanza of the failed node. This statement specifies that the central manager ignores connections from the schedd daemon running on this machine, and prevents conflicts from arising when a schedd machine is restarted while a purge (see below) is taking place.
3. Reconfigure the central manager node so that it recognizes the “fenced” node. From the central manager machine issue **llctl reconfig**.
4. Issue **llctl -h** *host* **purgeschedd** to purge all jobs scheduled by the schedd on the failed node.
5. Remove all files in the LoadLeveler spool directory of the failed node. Once the failed node is working again, you can remove the **schedd_fenced=true** statement.

Other Questions

Why do I have to setuid = 0?: The master daemon starts the startd daemon and the startd daemon starts the starter process. The starter process runs the job. The job needs to be run by the userid of the submitter. You either have to have a separate master daemon running for every ID on the system or the master daemon has to be able to **su** to every userid and the only user ID that can **su** any other userid is **root**.

Why Doesn’t LoadLeveler Execute my .profile or .login Script?: When you submit a batch job to LoadLeveler, the operating system will execute your **.profile** script before executing the batch job if your login shell is the Korn shell. On the other hand, if your login shell is the Bourne shell, on most operating systems (including AIX), the **.profile** script is not executed. Similarly, if your login shell is the C shell then AIX will execute your **.login** script before executing your LoadLeveler batch job but some other variants of UNIX may not invoke this script.

The reason for this discrepancy is due to the interactions of the shells and the operating system. To understand the nature of the problem, examine the following C program that attempts to open a login Korn shell and execute the ″ls″ command:

```
#include <stdio.h>
main()
{
execl("/bin/ksh","-","-c","ls",NULL);
}
```

UNIX documentations in general (SunOS, HP-UX, AIX, IRIX) give the impression that if the second argument is ″-″ then you get a login shell regardless of whether the first argument is /bin/ksh or /bin/csh or /bin/sh. In practice, this is not the case. Whether you get a login shell or not is implementation dependent and varies depending upon the UNIX version you are using. On AIX you get a login shell for /bin/ksh and /bin/csh but not the Bourne shell.

If your login shell is the Bourne shell and you would like the operating system to execute your **.profile** script before starting your batch job, add the following statement to your job command file:

```
# @ shell = /bin/ksh
```

LoadLeveler will open a login Korn shell to start your batch job which may be a shell script of any type (Bourne shell, C shell, or Korn shell) or just a simple executable.

What Happens When a mksysb is Created When LoadLeveler is Running Jobs?: When you create a mksysb (an image of the currently installed operating system) at a time when LoadLeveler is running jobs, the state of the jobs is saved as part of the mksysb. When the mksysb is restored on a node, those jobs will appear to be on the node, in the same state as when they were saved, even though the jobs are not actually there. To delete these phantom jobs, you must remove all files from the LoadLeveler **spool** and **execute** directories and then restart LoadLeveler.

Helpful Hints

This section contains tips on running LoadLeveler, including some productivity aids.

Scaling Considerations

If you are running LoadLeveler on a large number of nodes (128 or more), network traffic between LoadLeveler daemons can become excessive to the point of overwhelming a receiving daemon. To reduce network traffic, consider the following daemon, keyword, and command recommendations for large installations.

- Set the **POLLS_PER_UPDATE*POLLING_FREQUENCY** interval to five minutes or more. This limits the volume of machine updates the startd daemons send to the negotiator. For example, set **POLLS_PER_UPDATE** to 10 and set **POLLING_FREQUENCY** to 30 seconds.
- If your installation's mix of jobs includes a high percentage of parallel jobs requiring many nodes, specify **schedd_host=yes** in the machine stanza of each schedd machine. The schedd daemons must communicate with hundreds of startd daemons every time a job runs. You can distribute this communication by activating many schedd daemons. You should activate as many schedd daemons as there are jobs likely to be running at any one time. When you do this, each schedd handles the dispatching of one parallel job.
- If your installation allows jobs to be submitted from machines running the schedd daemon, you should consider avoiding "schedd affinity" by specifying **SCHEDD_SUBMIT_AFFINITY=FALSE** in the LoadLeveler configuration file. By default, the **llsubmit** command submits a job to the machine where the command was invoked provided the schedd daemon is running on the machine. (This is called schedd affinity.)

- You can decrease the amount of time the negotiator daemon spends running negotiation loops by increasing the **NEGOTIATOR_INTERVAL** and the **NEGOTIATOR_CYCLE_DELAY**. For example, set **NEGOTIATOR_INTERVAL** to 600, and set **NEGOTIATOR_CYCLE_DELAY** to 30.
- Make sure the machine update interval is not too short by setting the **MACHINE_UPDATE_INTERVAL** to a value larger than three times the polling interval (**POLLS_PER_UPDATE*POLLING_FREQUENCY**). This prevents the negotiator from prematurely marking a machine as “down” or prematurely cancelling jobs.
- In a large LoadLeveler cluster, issuing the **llctl** command with the **-g** can take minutes to complete. To speed this up, set up a working collective containing the machines in the cluster and use the PSSP **dsh** command; for example, **dsh llctl -g reconfig**. This command also allows you to limit your operation to a subset of machines by defining other working collectives.

Hints for Running Jobs

Determining When Your Job Started and Stopped: By reading the notification mail you receive after submitting a job, you can determine the time the job was submitted, started, and stopped. Suppose you submit a job and receive the following mail when the job finishes:

```
Submitted at: Sun Apr 30 11:40:41 1996
Started   at: Sun Apr 30 11:45:00 1996
Exited    at: Sun Apr 30 12:49:10 1996

Real Time:   0 01:08:29
Job Step User Time:   0 00:30:15
Job Step System Time:   0 00:12:55
Total Job Step Time:   0 00:43:10

Starter User Time:   0 00:00:00
Starter System Time:   0 00:00:00
Total Starter Time:   0 00:00:00
```

This mail tells you the following:

Submitted at
: The time you issued the **llsubmit** command or the time you submitted the job with the graphical user interface.

Started at
: The time the starter process executed the job.

Exited at
: The actual time your job completed.

Real Time
: The wall clock time from submit to completion.

Job Step User Time
: The CPU time the job consumed executing in user space.

Job Step System Time
: The CPU time the system (AIX) consumed on behalf of the job.

Total Job Step Time
: The sum of the two fields above.

Starter User Time
: The CPU time consumed by the LoadLeveler starter process for this job,

executing in user space. Time consumed by the starter process is the only LoadLeveler overhead which can be directly attributed to a user's job.

Starter System Time
The CPU time the system (AIX) consumed on behalf of the LoadLeveler starter process running for this job.

Total Starter Time
The sum of the two fields above.

You can also get the starting time by issing **llsummary -l -x** and then issuing **awk /Date|Event/** against the resulting file. For this to work, you must have **ACCT = A_ON A_DETAIL** set in the **LoadL_config** file.

Running Jobs at a Specific Time of Day: Using a machine's local configuration file, you can set up the machine to run jobs at a certain time of day (sometimes called an *execution window*). The following coding in the local configuration file runs jobs between 5:00 PM and 8:00AM daily, and suspends jobs the rest of the day:

```
START: (tm_day >= 1700) || (tm_day <= 0800)
SUSPEND: (tm_day > 0800)  && (tm_day < 1700)
CONTINUE: (tm_day >= 1700) || (tm_day <= 0800)
```

Controlling the Mix of Idle and Running Jobs: Three keywords determine the mix of idle and running jobs for a user. By a running job, we mean a job that is in one of the following states: Running, Pending, or Starting. These keywords, which are described in detail in "Step 2: Specify User Stanzas" on page 81, are:

maxqueued
Controls the number of jobs in any of these states: Idle, Running, Pending, or Starting.

maxjobs
Controls the number of jobs in any of these states: Running, Pending, or Starting; thus it controls a subset of what **maxqueued** controls. **maxjobs** effectively controls the number of jobs in the Running state, since Pending and Starting are usually temporary states.

maxidle
Controls the number of jobs in any of these states: Idle, Pending, or Starting; thus it controls a subset of what **maxqueued** controls. **maxidle** effectively controls the number of jobs in the Idle state, since Pending and Starting are usually temporary states.

What Happens When You Submit a Job: For a user's job to be allowed into the job queue, the total of other jobs (in the Idle, Pending, Starting and Running states) for that user must be less than the **maxqueued** value for that user. Also, the total idle jobs (those in the Idle, Pending, and Starting states) must be less than the **maxidle** value for the user. If either of these constraints are at the maximum, the job is placed in the Not Queued state until one of the other jobs changes state. If the user is at the **maxqueued** limit, a job must complete, be cancelled, or be held before the new job can enter the queue. If the user is at the **maxidle** limit, a job must start running, be cancelled, or be held before the new job can enter the queue.

Once a job is in the queue, the job is not taken out of queue unless the user places a hold on the job, the job completes, or the job is cancelled. (An exception to this, when you are running the default LoadLeveler scheduler, is parallel jobs which do

not accumulate sufficient machines in a given time period. These jobs are moved to the Deferred state, meaning they must vie for the queue when their Deferred period expires.)

Once a job is in the queue, the job will run unless the **maxjobs** limit for the user is at a maximum.

Note the following restrictions for using these keywords:

- If **maxqueued** is greater than (**maxjobs** + **maxidle**), the **maxqueued** value will never be reached.
- If either **maxjobs** or **maxidle** is greater than **maxqueued**, then **maxqueued** will be the only restriction in effect, since **maxjobs** and **maxidle** will never be reached.

Sending Output from Several Job Steps to One Output File: You can use dependencies in your job command file to send the output from many job steps to the same output file. For example:

```
# @ step_name = step1
# @ executable = ssba.job
# @ output = ssba.tmp
# @ ...
# @ queue
#
# @ step_name = append1
# @ dependency = (step1 != CC_REMOVED)
# @ executable = append.ksh
# @ output = /dev/null
# @ queue
# @
# @ step_name = step2
# @ dependency = (append1 == 0)
# @ executable = ssba.job
# @ output = ssba.tmp
# @ ...
# @ queue
# @
# @ step_name = append2
# @ dependency = (step2 != CC_REMOVED)
# @ executable = append.ksh
# @ output = /dev/null
# @ queue
#
# ...
```

Then, the file **append.ksh** could contain the line **cat ssba.tmp >> ssba.log**. All your output will reside in **ssba.log**. (Your dependecies can look for different return values, depending on what you need to accomplish.)

You can achieve the same result from within **ssba.job** by appending your output to an output file rather than writing it to **stdout**. Then your output statement for each step would be **/dev/null** and you wouldn't need the append steps.

Hints for Using Machines

Setting Up a Single Machine To Have Multiple Job Classes: You can define a machine to have multiple job classes which are active at different times. For example, suppose you want a machine to run jobs of Class A any time, and you want the same machine to run Class B jobs between 6 p.m. and 8 a.m.

You can combine the **Class** keyword with a user-defined macro (called **Off_shift** in this example).

For example:

```
Off_Shift = ((tm_hour >= 18) || (tm_hour < 8))
```

Then define your START statement:

```
START : (Class == "A") || ((Class == "B") && $(Off_Shift))
```

Make sure you have the parenthesis around the **Off_Shift** macro, since the logical OR has a lower precedence than the logical AND in the START statement.

Also, to take weekends into account, code the following statements. Remember that Saturday is day 6 and Sunday is day 0.

```
Off_Shift = ((tm_wday == 6) || (tm_wday == 0) || (tm_hour >=18) \
|| (tm_hour < 8))

Prime_Shift = ((tm_wday != 6) && (tm_wday != 0) && (tm_hour >= 8) \
&& (tm_hour < 18))
```

Reporting the Load Average on Machines: You can use the **/usr/bin/rup** command to report the load average on a machine. The **rup** *machine_name* command gives you a report that looks similar to the following:

```
localhost    up 23 days, 10:25,    load average: 1.72, 1.05, 1.17
```

You can use this command to report the load average of your local machine or of remote machines. Another command, **/usr/bin/uptime**, returns the load average information for only your local host.

History Files and schedd

The **schedd** daemon writes to the spool/history file only when a job is completed or removed. Therefore, you can delete the history file and restart **schedd** even when some jobs are scheduled to run on other hosts.

However, you should clean up the **spool/job_queue.dir** and **spool/job_queue.pag** files only when no jobs are being scheduled on the machine.

You should not delete these files if there are any jobs in the job queue that are being scheduled from this machine (for example, jobs with names such as *thismachine.clusterno.jobno*).

Getting Help from IBM

Should you require help from IBM in resolving a LoadLeveler problem, you can get assistance by calling IBM Support. Before you call, be sure you have the following information:

1. Your access code (customer number).
2. The LoadLeveler product number (5765-D61).
3. The name and version of the operating system you are using.
4. A telephone number where you can be reached.

In addition, issue the following command:

```
llctl version
```

This command will provide you with code level information. Provide this information to the IBM representative.

The number for IBM support in the United States is 1-800-IBM-4YOU (426-4968).

The Facsimile number is 800-2IBM-FAX (2426-329).

Appendix B. Customer Case Studies

This chapter gives you an overview, including configuration information, of some LoadLeveler customers. These profiles are meant to highlight how customers in different industries use LoadLeveler.

Note that all of these configurations apply to Version 1 Release 3 of the default LoadLeveler scheduler unless otherwise noted.

Customer 1: Technical Computing at the Cornell Theory Center

The Cornell Theory Center (CTC) of Cornell University provides a high-performance computing environment to advance and facilitate research and education.

System Configuration

The CTC runs a 160-node SP with 16 wide nodes and 144 thin nodes. The SP nodes include two interactive nodes and two submit-only nodes. The majority of the other SP nodes run batch jobs. The LoadLeveler central manager runs on a workstation outside of the SP. Also, two other non-SP workstations act as schedd hosts.

LoadLeveler Configuration

The CTC runs parallel jobs by disabling the default LoadLeveler scheduler **SCHEDULER_API=YES**) and running an external scheduler. The CTC has developed this scheduler to meet the needs of its users.

The following figures represent sections of the CTC's **LoadL_admin** file. Note that not all nodes are shown here.

```
########################################################################
# DEFAULTS FOR MACHINE, CLASS, USER, AND GROUP STANZAS:
# Remove initial # (comment), and edit to suit.
########################################################################
default:        type = machine
                central_manager = false  # default not central manager
                schedd_host = false      # default not a public scheduler
                submit_only = false      # default not a submit-only machine
                pvm_root = /usr/local/app/pvm3  # default pvm3 directory
                rm_host = true           # default is parallel SP2 node
#               speed = 1                # default machine speed
#               cpu_speed_scale = false  # scale cpu limits by speed

default:        type = class             # default class stanza
#               priority = 0             # default ClassSysprio
#               max_processors = -1      # default max processors for class (no

default:        type = user              # default user stanza
#               priority = 0             # default UserSysprio
                default_class = DSI      # default class
                default_group = No_Group # default group = No_Group (not
                                         # optional)
#               maxjobs = -1             # default maximum jobs user is allowed
                                         # to run simultaneously (no limit)
#               maxqueued = -1           # default maximum jobs user is allowed
                                         # on system queue (no limit).  does not
                                         # limit jobs submitted.

default:        type = group             # default group stanza
#               priority = 0             # default GroupSysprio
#               maxjobs = -1             # default maximum jobs group is allowed
```

```
                                        # to run simultaneously (no limit)
#               maxqueued = -1          # default maximum jobs group is allowed
                                        # on system queue (no limit).  does not
                                        # limit jobs submitted.
##########################################################################
# MACHINE STANZAS:
# These are the machine stanzas; the first machine is defined as
# the central manager.  mach1:, mach2:, etc. are machine name labels -
# revise these placeholder labels with the names of the machines in the
# pool, and specify any schedd_host and submit_only keywords and values
# (true or false), if required.
##########################################################################

# spscheduler is a 43P running EASY-LL and the Central Manager
spscheduler.tc.cornell.edu:   type = machine
                              central_manager = true
                              rm_host =false

# ctc1 and ctc2 are two 43P's running as dedicated SchedDs
ctc1.tc.cornell.edu: type = machine
                           schedd_host = true

ctc2.tc.cornell.edu: type = machine
                           schedd_host = true

# Submit only node for Sweb server
arms.tc.cornell.edu:  type = machine
                     submit_only = true

#
#   Nodes of the SP2
#
# Rack 1
#
# PIOFS name server, HiPPi router, Switch & JMD primary
#r01n01.tc.cornell.edu:   type = machine
#                         alias = r01n01-css
# r01n02 & r01n05 are interactive nodes
r01n03.tc.cornell.edu:   type = machine
                         alias = r01n03-css
                         submit_only = true
r01n05.tc.cornell.edu:   type = machine
                         alias = r01n05-css
                         submit_only = true
r01n07.tc.cornell.edu:   type = machine
                         alias = r01n07-css
r01n09.tc.cornell.edu:   type = machine
                         alias = r01n09-css
r01n11.tc.cornell.edu:   type = machine
                         alias = r01n11-css
r01n13.tc.cornell.edu:   type = machine
                         alias = r01n13-css
r01n15.tc.cornell.edu:   type = machine
                         alias = r01n15-css
#
# Rack 2
#
# HPSS/PIOFS backup
#r02n01.tc.cornell.edu:   type = machine
#                         alias = r02n01-css
# r02n03, r02n05, r02n07, r02n09 are splong nodes
r02n03.tc.cornell.edu:   type = machine
                         alias = r02n03-css
                         submit_only = true
r02n05.tc.cornell.edu:   type = machine
                         alias = r02n05-css
                         submit_only = true
r02n07.tc.cornell.edu:   type = machine
```

```
                        alias = r02n07-css
                        submit_only = true
r02n09.tc.cornell.edu:  type = machine
                        alias = r02n09-css
                        submit_only = true
# VIS node
#r02n11.tc.cornell.edu:  type = machine
#                        alias = r02n11-css
r02n13.tc.cornell.edu:  type = machine
                        alias = r02n13-css
r02n15.tc.cornell.edu:  type = machine
                        alias = r02n15-css

#
# Rack 3
#
r03n01.tc.cornell.edu:  type = machine
                        alias = r03n01-css
r03n02.tc.cornell.edu:  type = machine
                        alias = r03n02-css
r03n03.tc.cornell.edu:  type = machine
                        alias = r03n03-css
r03n04.tc.cornell.edu:  type = machine
                        alias = r03n04-css
r03n05.tc.cornell.edu:  type = machine
                        alias = r03n05-css
r03n06.tc.cornell.edu:  type = machine
                        alias = r03n06-css
r03n07.tc.cornell.edu:  type = machine
                        alias = r03n07-css
r03n08.tc.cornell.edu:  type = machine
                        alias = r03n08-css
r03n09.tc.cornell.edu:  type = machine
                        alias = r03n09-css
r03n10.tc.cornell.edu:  type = machine
                        alias = r03n10-css
r03n11.tc.cornell.edu:  type = machine
                        alias = r03n11-css
r03n12.tc.cornell.edu:  type = machine
                        alias = r03n12-css
r03n13.tc.cornell.edu:  type = machine
                        alias = r03n13-css
r03n14.tc.cornell.edu:  type = machine
                        alias = r03n14-css
r03n15.tc.cornell.edu:  type = machine
                        alias = r03n15-css
# ATM/FDDI routing node
#r03n16.tc.cornell.edu:  type = machine
#                        alias = r03n16-css


#
# Rack 4
#
r04n01.tc.cornell.edu:  type = machine
                        alias = r04n01-css
r04n02.tc.cornell.edu:  type = machine
                        alias = r04n02-css
r04n03.tc.cornell.edu:  type = machine
                        alias = r04n03-css
r04n04.tc.cornell.edu:  type = machine
                        alias = r04n04-css
r04n05.tc.cornell.edu:  type = machine
                        alias = r04n05-css
r04n06.tc.cornell.edu:  type = machine
                        alias = r04n06-css
r04n07.tc.cornell.edu:  type = machine
```

```
                          alias = r04n07-css
r04n08.tc.cornell.edu:    type = machine
                          alias = r04n08-css
r04n09.tc.cornell.edu:    type = machine
                          alias = r04n09-css
r04n10.tc.cornell.edu:    type = machine
                          alias = r04n10-css
r04n11.tc.cornell.edu:    type = machine
                          alias = r04n11-css
# r04n12 - r14n16 HPSS nodes
#r04n12.tc.cornell.edu:    type = machine
#                          alias = r04n12-css
#r04n13.tc.cornell.edu:    type = machine
#                          alias = r04n13-css
#r04n14.tc.cornell.edu:    type = machine
#                          alias = r04n14-css
#r04n15.tc.cornell.edu:    type = machine
#                          alias = r04n15-css
#r04n16.tc.cornell.edu:    type = machine
#                          alias = r04n16-css
#
############################################################################
# CLASS STANZAS: (optional)
# These are sample class stanzas; small, medium, large, and nqs are sample
# labels for job classes - revise these labels and specify attributes
# to each class.
############################################################################
DSI:       type = class

piofs:     type = class
############################################################################
```

The following represents the CTC's **LoadL_config** file:

```
#
# Machine Description
#
ARCH = R6000

#
#  Specify LoadLeveler Administrators here:
#
LOADL_ADMIN = loadl admin1 admin2 admin3 admin4

#
# Default to starting LoadLeveler daemons when requested
#
START_DAEMONS = TRUE

#
# Machine authentication
#
# If TRUE, only connections from machines in the ADMIN_LIST are accepted.
# If FALSE, connections from any machine are accepted.  Default if not
# specified is FALSE.
#
MACHINE_AUTHENTICATE = FALSE

#
# Specify which daemons run on each node
#
SCHEDD_RUNS_HERE = False
STARTD_RUNS_HERE = True
```

```
#
# Specify information for backup central manager
#
# CENTRAL_MANAGER_HEARTBEAT_INTERVAL = 300
# CENTRAL_MANAGER_TIMEOUT = 6
#
# Specify pathnames
#
RELEASEDIR = /usr/lpp/LoadL/nfs
LOCAL_CONFIG = $(tilde)/local/configs/LoadL_config.$(host)
ADMIN_FILE = $(tilde)/LoadL_admin
LOG = /var/loadl/log
SPOOL = /var/loadl/spool
EXECUTE = /var/loadl/execute
HISTORY = $(SPOOL)/history
BIN = $(RELEASEDIR)/bin
LIB = $(RELEASEDIR)/lib
ETC = $(RELEASEDIR)/etc
#
# Specify port numbers
#
COLLECTOR_STREAM_PORT  = 9612
MASTER_STREAM_PORT     = 9616
NEGOTIATOR_STREAM_PORT = 9614
SCHEDD_STREAM_PORT     = 9605
STARTD_STREAM_PORT     = 9611
COLLECTOR_DGRAM_PORT   = 9613
STARTD_DGRAM_PORT      = 9615
MASTER_DGRAM_PORT      = 9617
SCHEDULER_API          = YES
SCHEDULER_PORT         = 9624

#
# Specify accounting controls
#
ACCT    = A_ON
ACCT_VALIDATION  = $(BIN)/llacctval
GLOBAL_HISTORY  = $(SPOOL)

#
# Specify prolog and epilog path names
#
JOB_PROLOG = $(ETC)/llprolog
JOB_EPILOG = $(ETC)/llepilog
JOB_USER_PROLOG = $(ETC)/ll_user_prolog
JOB_USER_EPILOG = $(ETC)/ll_user_epilog
#
#
# Refresh AFS token program.
#
AFS_GETNEWTOKEN = $(ETC)/tokenreviveclient
#
# Customized mail delivery program.
#
# MAIL =

#
# Customized submit (job command file) filter program.
#
# SUBMIT_FILTER =

#
# Specify checkpointing intervals
#
MIN_CKPT_INTERVAL    = 900
MAX_CKPT_INTERVAL    = 7200
```

```
# LoadL_KeyboardD Macros
#
KBDD                = $(BIN)/LoadL_kbdd
KBDD_LOG            = $(LOG)/KbdLog
MAX_KBDD_LOG        = 64000
KBDD_DEBUG          =

#
# Specify whether to start the keyboard daemon
#

X_RUNS_HERE   = False

#
# Specify whether to use X server XGetIdleTime() protocol extension
#

USE_X_IDLE_EXTENSION = False

#
#  LoadL_StartD Macros
#
STARTD   = $(BIN)/LoadL_startd
STARTD_LOG  = $(LOG)/StartLog
MAX_STARTD_LOG  = 5000000
#STARTD_DEBUG  = D_STARTD D_FULLDEBUG D_THREAD
STARTD_DEBUG  = D_FULLDEBUG
POLLING_FREQUENCY = 10
POLLS_PER_UPDATE = 24
JOB_LIMIT_POLICY = 240
JOB_ACCT_Q_POLICY = 3600

#
#  LoadL_SchedD Macros
#
SCHEDD   = $(BIN)/LoadL_schedd
SCHEDD_LOG  = $(LOG)/SchedLog
MAX_SCHEDD_LOG  = 5000000
SCHEDD_DEBUG  = D_SCHEDD
SCHEDD_INTERVAL  = 180

CLIENT_TIMEOUT  = 300
#
# Negotiator Macros
#
NEGOTIATOR  = $(BIN)/LoadL_negotiator
NEGOTIATOR_DEBUG  = D_FULLDEBUG D_ALWAYS D_NEGOTIATE
NEGOTIATOR_LOG = $(LOG)/NegotiatorLog
MAX_NEGOTIATOR_LOG = 5000000
NEGOTIATOR_INTERVAL = 60
MACHINE_UPDATE_INTERVAL = 600
NEGOTIATOR_PARALLEL_DEFER = 1800
NEGOTIATOR_PARALLEL_HOLD = 300
NEGOTIATOR_REDRIVE_PENDING = 1800
NEGOTIATOR_RESCAN_QUEUE  = 180
NEGOTIATOR_REMOVE_COMPLETED = 0

#
# Sets the interval between recalculation of the SYSPRIO values
# for all the jobs in the queue
#
NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL = 0

#
# Starter Macros
#
```

```
STARTER = $(BIN)/LoadL_starter
STARTER_DEBUG = D_FULLDEBUG
STARTER_LOG = $(LOG)/StarterLog
MAX_STARTER_LOG = 500000

#
# LoadL_Master Macros
#
MASTER   = $(BIN)/LoadL_master
MASTER_LOG  = $(LOG)/MasterLog
MASTER_DEBUG  = D_FULLDEBUG
MAX_MASTER_LOG  = 64000
RESTARTS_PER_HOUR = 12
PUBLISH_OBITUARIES = TRUE
OBITUARY_LOG_LENGTH = 25

#
# Specify whether log files are truncated when opened
#
TRUNC_MASTER_LOG_ON_OPEN     = False
TRUNC_STARTD_LOG_ON_OPEN     = False
TRUNC_SCHEDD_LOG_ON_OPEN     = False
TRUNC_KBDD_LOG_ON_OPEN       = False
TRUNC_STARTER_LOG_ON_OPEN    = False
TRUNC_COLLECTOR_LOG_ON_OPEN  = False
TRUNC_NEGOTIATOR_LOG_ON_OPEN = False
#       NQS Directory
#
#
# For users of NQS resources:
# Specify the directory containing qsub, qstat, qdel
#
# NQS_DIR   = /usr/bin

#
# Specify Custom metric keywords
#
# CUSTOM_METRIC  =
# CUSTOM_METRIC_COMMAND = $(ETC)/sw_chip_number
#
# Machine control expressions and macros
#

OpSys :  $(OPSYS)
Arch  :  $(ARCH)
Machine :  $(HOST).$(DOMAIN)

#
# Expressions used to control starting and stopping of foreign jobs
#
MINUTE  = 60
HOUR  = (60 * $(MINUTE))
StateTimer = (CurrentTime - EnteredCurrentState)

BackgroundLoad  = 0.7
HighLoad  = 1.5
StartIdleTime  = 15 * $(MINUTE)
ContinueIdleTime = 5 * $(MINUTE)
MaxSuspendTime  = 10 * $(MINUTE)
MaxVacateTime  = 10 * $(MINUTE)

KeyboardBusy= KeyboardIdle < $(POLLING_FREQUENCY)
CPU_Idle = LoadAvg <= $(BackgroundLoad)
CPU_Busy = LoadAvg >= $(HighLoad)
# START  : $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
# SUSPEND : $(CPU_Busy) || $(KeyboardBusy)
# CONTINUE : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)
```

```
# VACATE : $(StateTimer) > $(MaxSuspendTime)
# KILL  : $(StateTimer) > $(MaxVacateTime)

START  : T
SUSPEND  : F
CONTINUE : T
VACATE  : F
KILL  : F
#
# Expressions used to prioritize job queue
#
# Values which can be part of the SYSPRIO expression are:
#
# QDate    Job submission time
# UserPrio   User priority
# UserSysprio   System priority value based on userid (from the user
#     list file with default of 0)
# ClassSysprio   System priority value based on job class (from the class
#     list file with default of 0)
# UserRunningProcs  Number of jobs running for the user
# GroupRunningProcs Number of jobs running for the group
#
# The following expression is an example.
#
#SYSPRIO: (ClassSysprio * 100) + (UserSysprio * 10) + (GroupSysprio * 1)- (QDate
)
#
# The following (default) expression for SYSPRIO creates a FIFO job queue.
#
SYSPRIO: (ClassSysprio * 100) - (QDate)
#
# Expressions used to prioritize machines
#
# The following example orders machines by the load average
# normalized for machine speed:
#
#MACHPRIO: 0 - (1000 * (LoadAvg / (Cpus * Speed)))
#
# The following (default) expression for MACHPRIO orders
# machines by load average.
#
#MACHPRIO: 0 - (LoadAvg) + (MasterMachPriority * 10000)
#       The following  expression for MACHPRIO orders
#       machines by increasing ammount of memory and
#       decreasing node number.
#
MACHPRIO: 0 - (100 * Memory) + CustomMetric + (MasterMachPriority * 10000)

#
# The MAX_JOB_REJECT value determines how many times a job can be
# rejected before it is canceled or put on hold.  The default value
# is -1, which indicates no limit to the number of times a job can be
# rejected.

#
MAX_JOB_REJECT = 0
#
# When ACTION_ON_MAX_REJECT is HOLD, jobs will be put on user hold
# when the number of rejects reaches the MAX_JOB_REJECT value.  When
# ACTION_ON_MAX_REJECT is CANCEL, jobs will be canceled when the
# number of rejects reaches the MAX_JOB_REJECT value. The default
# value is HOLD.
#
ACTION_ON_MAX_REJECT = CANCEL
```

Customer 2: Circuit Simulation

This customer performs CPU-intensive work in the area of circuit simulation using Electronic Design Automation (EDA).

System Configuration

The customer has 752 batch servers; 209 are dedicated to run LoadLeveler jobs 24 hours a day (the central manager is excluded). The rest are used by LoadLeveler when they are not in use by their respective owners.

The LoadLeveler administrators control all the 173 dedicated machines. That means that users cannot get onto these systems without submitting a LoadLeveler job. 117 of the dedicated machines are public schedulers. The user machines are submit-only machines, and users do not have access to their root password. If a user needs root access to his or her machine, he or she is allowed alternate root access only; he or she cannot get global root access to all the machines on site. (Site administrators use a common global root password.)

This site runs over 31,000 jobs per week and about 2,800 CPU days of resource utilization. The central manager is a RISC/System 6000 model 370 with 128MB of RAM. The batch machines are generally 80 percent busy. The central manager is about 35 percent to 70 percent busy. The central manager does not run any jobs, it just manages. All of the LoadLeveler machines run one job at a time. (That is, **MAX_STARTERS=1**.)

This customer sees some machines in a down state occassionally. The administrator feels the CPU on these machines are too busy to get a time slice to report its state to the central manager. However, this down state does not cause any problem for this customer.

117 public schedulers are subset of our 173 dedicated machines and are listed in the admin file.

LoadLeveler Configuration

The following figures represent sections of this customer's **LoadL_admin** file for dedicated machines. Notice the default stanza. Also, every machine in the LoadLeveler cluster is listed in this file.

```
#================================================================================#
# type = machine default stanza
#================================================================================#

default:  type = machine             # defaults for machine stanzas
central_manager = false    #  no central manager on machine
schedd_host = true         #  public schedd on machine
#================================================================================#
# Central Manager
#================================================================================#

mips1:    type = machine          # PRIMARY server - MANAGER   370 128M 3.2.5
central_manager = true  #  runs negotiator
#================================================================================#
#                                Primary Servers
#================================================================================#

beast100: type = machine
# PRIMARY C=a/b/o/s2/t2           . . 550    128M 3.2.5
beast101: type = machine
# PRIMARY C=a/b/b1/b4/c/o/r/s/t   F . 550    128M 3.2.5
```

```
beast102: type = machine
# PRIMARY C=a                    F . 550    128M 3.2.5
beast103: type = machine
# PRIMARY C=a                    . . 550    128M 3.2.5
```

Later in the **Loadl_admin file**, user machines are defined. Notice the default stanza.

```
#============================================================================#

default:  type = machine             # defaults for machine stanzas
central_manager = false    #  no central manager on machine
schedd_host = false        #  no public schedd on machine
#============================================================================#

agni:      type = machine
# SECONDARY server - rmkohn           550    64M 3.2.5
akama:     type = machine
# SECONDARY server - poulter          365    64M 3.2.5
alaska:    type = machine
# SECONDARY server - jcahill          340    64M 3.2.5
alcor:     type = machine
# SECONDARY server - drolson          340    64M 3.2.5
```

The following represents a local configuration file for a dedicated, public scheduler machine:

```
#                          PRIMARY LoadL SERVER ==> mips27
#
# this loadl.config.local is tuned for a machine that is part of a compute
# farm.  Interactive users are discouraged.
#
# Run up to one jobs at a time.
#
# Always start a job if there is a class available.
#
# Never suspend a job.
#
# Since jobs never get suspended they never get vacated or killed.
#

SCHEDD_RUNS_HERE    = True
STARTD_RUNS_HERE    = True

Class = { "a" "b" "b1" "b4" "c" "k" "r" "s" "t" }
Feature = { "PRI" }

MAX_STARTERS = 1

POLLING_FREQUENCY       = 30
POLLS_PER_UPDATE        = 15

START           : T
SUSPEND         : F

START_DAEMONS = True
X_RUNS_HERE   = False
```

The following represents a local configuration file for a user's machine.

```
#                       SECONDARY SERVER ==> common
#
# This loadl_config.local is tuned to be "nice" to a workstation owner
# who permits loadl jobs on his system but wants good response whenever
# he is doing his own work.
#
```

```
# Run only one LoadLeveler job at a time.
#
# Check the keyboard for activity every five seconds.
#
#
# Suspend a job if the load average exceeds 1.4
#
# Continue a job when keyboard again goes idle for 10 minutes and the load
# average is <.5

SCHEDD_RUNS_HERE  = False
STARTD_RUNS_HERE  = True

Class = { "a" "b" "b1" "b4" "c" "o" "r" "s" "t" }
MAX_STARTERS = 1

START           : $(FirstShift_KB9999) && $(StartS1) || ($(Off_Shift) ||
$(Week_End)) && $(Mach_Idle_S)
SUSPEND         : $(CPU_Busy) || $(KeyboardBusy)
CONTINUE        : $(Mach_Idle_C)
VACATE          : ((Class == "a") && $(Vacate_A)) || ($(Vacate_ClassesB)
&& $(Vacate_B)) || $(Vacate_X)
KILL            : $(Kill_Job)

START_DAEMONS = True
X_RUNS_HERE   = True
```

Customer 3: High-Energy Physics

This scientific customer provides experimental facilities for physicists from its 17 member states and for visiting scientists from throughout the world. The computing requirements of these users vary from mail and text processing to heavy batch and parallel processing.

System Configuration

Their processor is an SP2 using RISC System/6000 nodes linked by an internal high-speed network with a centrally managed software environment. The nodes are functionally divided into four groups of 16 each for different types of work: interactive logins, sequential job batch processing, parallel job batch processing and data, and tape and network services.

This customer uses AFS heavily. It provides the single system image for users' home directories and the files common to their experiments. Many software products are served directly out of AFS using symbolic links.

LoadLeveler provides this customer with the following facilities:

- Interactive load balancing of users across nodes on the SP2 and other UNIX services on site
- Batch services for serial compute jobs
- Scheduling for parallel applications

LoadLeveler Batch Configuration

The batch configuration is designed to maximize short job turnaround while allowing the heavy CPU jobs to get good usage of the resources available.

The basic configuration uses a range of classes – `short`, `medium`, `long` and `verylong` – with a range of maximum job CPU times of from five minutes to six days. An

additional class, `night`, provides off-peak and weekend computing time on the interactive areas of the SP2 during periods of low demand. Access to this class is limited to specific users.

Users in different experiments are defined in LoadLeveler groups which provide associated queue priorities. This allows groups with a large computing budget to be given higher priorities. An automated procedure calculates each group's resource utilization over the last month and adjusts their priorities accordingly. This ensures a fair allocation of CPU time among the groups.

LoadLeveler Interactive Configuration

This customer uses the Interactive Session Support facility to provide a name servier which returns the least loaded node according to a site defined metric. This allows a user to be given the least loaded operational node when he or she logs in.

This metric is based on the number of logged in users, with some weight given to those using Xstations. Every few minutes, the system is scanned to evaluate the following:

*Xterminals*3 + *Telnet*2 + *Process*

Where:

- *Xterminals* is the number of users logged in from an Xstation
- *Telnet* is the number logged in via **telnet** or **rlogin**
- *Process* is the number of users who have processes running.

This metric tries to balance users across the system while providing some factor for their likely future utilization. A metric based on the CPU load average is too dependent on the current load to provide good balancing.

The metric can also be set to return a low priority if the file **/etc/iss.nologin** exists. This allows the administrator to drain the interactive use of a node if there is scheduled system maintenance. When the maintenance is completed, the file can be removed and the metric will return the correct value for the node. Users will therefore see an improved availability, since they will not be given a node that is about to shutdown.

Processor Configuration

The processors are configured as follows:

- **parallel** nodes support a mixture of short, medium, long, and verylong classes.
- **batch** nodes support the same class mix as parallel. Additional paging space is available on these nodes to provide multiple jobs running per node.
- **interactive** nodes support the night class only. The night class only allows jobs to start after 6 PM and before midnight during the week and anytime on weekends. A maximum CPU time of 8 hours ensures that the jobs are finished when the prime shift starts. This is configured using LoadLeveler's START expression:

```
Is_Weekend          = (tm_wday==0  || tm_wday==6)
Is_Start_Night_Time = (tm_hour>18)

START: $(Is_Start_Night_Time) || $(Is_Weekend)
```

Customer 4: Computer Chip Design

This customer uses EDA to perform work in the area of computer chip design.

System Configuration

The customer has seven clusters of RISC/System 6000 machines. The largest cluster has 530 machines; the smallest cluster has 87 machines. The total number of machines at this installation is over 1200.

Interactive Configuration

This customer has defined two configuration files for interactive work: one for standard workstations and one for large interactive servers. These files are meant to be tailored to machines of differing processing power.

Standard Workstation Configuration

```
#============================================================================#
# Description: LoadL_config.local for Standard Workstations (<370 Class)
#============================================================================#
# Need 2x Paging Space to Real Memory ( minimum ) For Worst Case Of One
# Suspended and One Foreground Running Job.
#    *) All Jobs (btv,lp) Suspend on LoadAvg or Keyboard/Mouse Movement.
#============================================================================#
# Class defines the permissable classes, MAX_STARTERS defines the max
# total jobs to be permitted.
#============================================================================#
Class        = { "btv" "lp" }
MAX_STARTERS = 1
#============================================================================#
# The next definitions are used in the expressions below to regulate the
# conditions under which jobs get started, suspended, and evicted.
#     All times are specified in units of seconds.
#============================================================================#
BackgroundLoad   = 0.8
HighLoad         = 1.6
StartIdleTime    = 900
ContinueIdleTime = 900
#============================================================================#
# LoadAvg is an internal variable whose value is the (Berkeley) load average
# of the machine.
#
#    CPU_Idle - No LoadL job running, or One job just finishing.
#    CPU_Busy - One LoadL job running, second job ( Foreground or Batch )
#               starting up.
#    CPU_Max  - Two LoadL jobs running.
#============================================================================#
CPU_Idle = (LoadAvg <= $(BackgroundLoad))
CPU_Busy = (LoadAvg >= $(HighLoad))

#============================================================================#
# This defines a boolean "KeyboardBusy" whose value is TRUE if the keyboard
# or mouse has been used since loadl last checked.  Thus if POLLING_FREQUENCY
# is 5 seconds, KeyboardBusy is TRUE if anybody has used the kbd or mouse in
# the last 5 seconds.
#============================================================================#
KeyboardBusy = KeyboardIdle < $(POLLING_FREQUENCY)

#============================================================================#
# This statement indicates when a job should be started on this machine
#============================================================================#
Weekend   = ( (tm_wday >=  6) || (tm_wday <  1) )
Day       = ( (tm_hour >=  7) && (tm_hour < 18) )
Night     = ( (tm_hour >= 18) || (tm_hour <  4) )
Inactive  = ( (KeyboardIdle > $(StartIdleTime)) && $(CPU_Idle) )

HP        = ( (Class == "btv") )
LP        = ( ($(Weekend) || $(Night)) )
```

```
START     : ( ($(HP) || $(LP)) && $(Inactive) )

#============================================================================#
# The SUSPEND statement here says that a job should be suspended but not
# killed if:
#                LoadAvg >= 1.6  Or  KeyboardIdle < 5
#============================================================================#
SUSPEND  : ( $(CPU_Busy) || $(KeyboardBusy) )

#============================================================================#
# This CONTINUE statement indicates that a suspended job should be continued
# if the cpu goes idle and the keyboard/mouse has not been used for the last
# 15 minutes.
#============================================================================#
CONTINUE : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)

#============================================================================#
# Jobs in the SUSPEND state are never killed, after 60 minutes they are
# relocated to a different machine if possible.
#============================================================================#
MaxSuspendTime   = 60 * $(MINUTE)
VACATE    : $(StateTimer) > $(MaxSuspendTime)
KILL      : F
#============================================================================#
# If you set START_DAEMONS to False loadl can never start on this machine.
# For example you may want to stop loadl for a couple days for maintenance
# and make sure no procedure automatically restarts it.
#============================================================================#
START_DAEMONS = True

#============================================================================#
# Set the maximum size each of the logs can reach before wrapping.
#============================================================================#
MAX_SCHEDD_LOG    = 128000
MAX_COLLECTOR_LOG = 128000
MAX_STARTD_LOG    = 128000
MAX_SHADOW_LOG    = 128000
MAX_KBDD_LOG      = 128000
```

Large Interactive Server Configuration

```
#============================================================================#
# Description: LoadL_config.local for Interactive Large Servers (580-590 Class)

#============================================================================#
# Need 3x Real Memory To Paging Space ( minimum ) For Worst Case Of Two
# Suspended and One Foreground Running Job.
#    *) All Jobs (btv,lp) Suspend on LoadAvg or Keyboard/Mouse Movement.
#    *) Real Memory >= 192meg.
#============================================================================#

#============================================================================#
# Class defines the permissable classes, MAX_STARTERS defines the max
# total jobs to be permitted.
#============================================================================#
Class        = { "btv" "lp" }
MAX_STARTERS = 2

#============================================================================#
# The next definitions are used in the expressions below to regulate the
# conditions under which jobs get started, suspended, and evicted.
#
#     All times are specified in units of seconds.
#============================================================================#
BackgroundLoad   = 0.8
LowLoad          = 1.0
```

```
HighLoad         = 1.6
MaxLoad          = 2.0
StartIdleTime    = 900
ContinueIdleTime = 900
#============================================================================#
# LoadAvg is an internal variable whose value is the (Berkeley) load average
# of the machine.
#
#    CPU_Idle - No LoadL job running, or One job just finishing.
#    CPU_Busy - One LoadL job running, second job ( Foreground or Batch )
#               starting up.
#    CPU_Max  - Two LoadL jobs running.
#============================================================================#
CPU_Idle = (LoadAvg <= $(BackgroundLoad))
CPU_Run  = (LoadAvg <= $(LowLoad))
CPU_Busy = (LoadAvg >= $(HighLoad))
CPU_Max  = (LoadAvg >= $(MaxLoad))

#============================================================================#
# This defines a boolean "KeyboardBusy" whose value is TRUE if the keyboard
# or mouse has been used since loadl last checked.  Thus if POLLING_FREQUENCY
# is 5 seconds, KeyboardBusy is TRUE if anybody has used the kbd or mouse in
# the last 5 seconds.
#============================================================================#
KeyboardBusy = KeyboardIdle < $(POLLING_FREQUENCY)
#============================================================================#
# This statement indicates when a job should be started on this machine
#============================================================================#
Weekend   = ( (tm_wday >=  6) || (tm_wday <  1) )
Day       = ( (tm_hour >=  7) && (tm_hour < 18) )
Night     = ( (tm_hour >= 18) || (tm_hour <  4) )
Inactive1 = ( (KeyboardIdle > $(StartIdleTime)) )
Inactive2 = ( (KeyboardIdle > $(ContinueIdleTime)) )

HP        = ( (Class == "btv") )
LP        = ( (Class == "lp") && $(CPU_Idle) )

START     : ( ($(HP) || $(LP)) && $(Inactive1) )

#============================================================================#
# The SUSPEND statement here says that a job should be suspended but not
# killed if:
#                KeyboardIdle < 5                Or
#                lp  Class  And  LoadAvg >= 1.6   Or
#                btv Class  And  LoadAvg >= 2.0
#============================================================================#
SUSPEND   : ( ( (Class == "lp")  && $(CPU_Busy) ) || \
( (Class == "btv") && $(CPU_Max)  ) || \
(  $(KeyboardBusy)                )    )

#============================================================================#
# This CONTINUE statement indicates that a suspended job should be continued
# if:
#               lp  Class  And  LoadAvg <= 0.8  And  KeyboardIdle > 15 min  Or
#               btv Class  And  LoadAvg <= 1.0  And  KeyboardIdle > 15 min
#============================================================================#
CONTINUE  : ( ( (Class == "lp")  && $(CPU_Idle) && $(Inactive2) ) || \
( (Class == "btv") && $(CPU_Run)  && $(Inactive2) )    )
#============================================================================#
# Jobs in the SUSPEND state are never killed, after 60 minutes they are
# relocated to a different box if possible.
#============================================================================#
MaxSuspendTime   = 60 * $(MINUTE)
VACATE    : $(StateTimer) > $(MaxSuspendTime)
KILL      : F
```

```
#============================================================================#
# If you set START_DAEMONS to False loadl can never start on this machine.
# For example you may want to stop loadl for a couple days for maintenance
# and make sure no procedure automatically restarts it.
#============================================================================#
START_DAEMONS = True

#============================================================================#
# Set the maximum size each of the logs can reach before wrapping.
#============================================================================#
MAX_SCHEDD_LOG    = 128000
MAX_COLLECTOR_LOG = 128000
MAX_STARTD_LOG    = 128000
MAX_SHADOW_LOG    = 128000
MAX_KBDD_LOG      = 128000
```

Batch Configuration

The following configuration file defines dedicated batch machines. Notice, however, that jobs in the `lp` class will suspend when a machine becomes too busy. So in this sense, the machines are not fully dedicated.

```
#============================================================================#
# Description: LoadL_config.local for Large Batch Servers ( 580 - 590 Class )
#============================================================================#
# Need 3x Real Memory To Paging Space ( minimum ) For Worst Case Of One
# Suspended and Two Foreground Running Job.
#    *) High Priority Jobs (btv) Never Suspend.
#    *) Job Suspension (lp) Based on LoadAvg Only.
#    *) Real Memory >= 192meg.
#============================================================================#

#============================================================================#
# Class defines the permissable classes, MAX_STARTERS defines the max
# total jobs to be permitted.
#============================================================================#
Class        = { "btv" "lp" }
MAX_STARTERS = 2

#============================================================================#
# The next definitions are used in the expressions below to regulate the
# conditions under which jobs get started, suspended, and evicted.
#
#     All times are specified in units of seconds.
#============================================================================#
BackgroundLoad   = 0.5
HighLoad         = 1.6
StartIdleTime    = 900
ContinueIdleTime = 900

#============================================================================#
# LoadAvg is an internal variable whose value is the (Berkeley) load average
# of the machine.
#
#    CPU_Idle - No LoadL job running, or One job just finishing.
#    CPU_Busy - One LoadL job running, second job ( Foreground or Batch )
#               starting up.
#    CPU_Max  - Two LoadL jobs running.
#============================================================================#
CPU_Idle = (LoadAvg <= $(BackgroundLoad))
CPU_Busy = (LoadAvg >= $(HighLoad))
#============================================================================#
# This defines a boolean "KeyboardBusy" whose value is TRUE if the keyboard
# or mouse has been used since loadl last checked.  Thus if POLLING_FREQUENCY
# is 5 seconds, KeyboardBusy is TRUE if anybody has used the kbd or mouse in
# the last 5 seconds.
#============================================================================#
```

```
KeyboardBusy = KeyboardIdle < $(POLLING_FREQUENCY)

#==========================================================================#
# This statement indicates when a job should be started on this machine
#==========================================================================#
HP        = ( (Class == "btv") )
LP        = ( (Class == "lp") && $(CPU_Idle) )

START     : ( $(HP) || $(LP) )

#==========================================================================#
# The SUSPEND statement here says that a "lp" job should be suspended but not
# killed if a high priority job starts up or a foreground job causes the
# Loadavg to be greater than CPU_Busy ( 1.6 ).
#==========================================================================#
SUSPEND   : (Class == "lp") && $(CPU_Busy)

#==========================================================================#
# This CONTINUE statement indicates that a suspended job should be continued
# if the cpu goes idle and the keyboard/mouse has not been used for the last
# 15 minutes.
#==========================================================================#
CONTINUE  : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)

#==========================================================================#
# Jobs in the SUSPEND state are never killed, after 60 minutes they are
# relocated to a different box if possible.
#==========================================================================#
MaxSuspendTime   = 60 * $(MINUTE)
VACATE    : $(StateTimer) > $(MaxSuspendTime)
KILL      : F

#==========================================================================#
# If you set START_DAEMONS to False loadl can never start on this machine.
# For example you may want to stop loadl for a couple days for maintenance
# and make sure no procedure automatically restarts it.
#==========================================================================#
START_DAEMONS = True

#==========================================================================#
# Set the maximum size each of the logs can reach before wrapping.
#==========================================================================#
MAX_SCHEDD_LOG    = 128000
MAX_COLLECTOR_LOG = 128000
MAX_STARTD_LOG    = 128000
MAX_SHADOW_LOG    = 128000
MAX_KBDD_LOG      = 128000
```

Configuration for a Machine That Schedules (But Doesn't Run) Jobs

The following statements define a machine that schedules jobs but does not run jobs. Notice that the schedd daemon is never forced to *not* run.

```
#
# This loadl local configuration file is set up to make a machine a
# submitter only.
#
# No jobs are allowed to run on this system.
#
MAX_STARTERS            = 0

START                   : F
#
# If you set START_DAEMONS to False loadl can never start on this machine.
```

```
# For example you may want to stop loadl for a couple days for maintenance
# and make sure no procedure automatically restarts it.
#
START_DAEMONS           = True
```

Glossary

This section contains some of the terms that are commonly used in the LoadLeveler books and in this book in particular.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (GC20-1699), *IBM DATABASE 2 Application Programming Guide for TSO Users* (SC26-4081), and *Internetworking With TCP/IP, Principles, Protocols, and Architecture* , by Douglas Comer, Copyright 1988 by Prentice Hall, Incorporated

A

AFS. Andrew File System.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

Authentication. The process of validating the identity of a user or server.

Authorization. The process of obtaining permission to perform specific actions.

B

Berkeley Load Average. The average number of processes on the operating system's ready to run queue.

C

C. A general purpose programming language. It was formalized by ANSI standards committee for the C language (X3J11) in 1984 and by Uniforum in 1983.

client. *(1) A function that requests services from a server, and makes them available to the user. *(2) An address space in MVS that is using TCP/IP services. *(3) A term used in an environment to identify a machine that uses the resources of the network.

cluster. (1) A group of processors interconnected through a high speed network that can be used for high performance computing. (2) A group of jobs submitted from the same job command file. (3)A set of machines with something in common between them. This commonality could be that they are all backed up by one machine or they are all in the LoadLeveler administration file.

D

daemon. A process, not associated with a particular user, that performs system-wide functions such as administration and control of networks, execution of time-dependent activities, line printer spooling, and so on.

datagram. A protocal known as the User Datagram Protocol (UDP). It is an internet standard protocol that allows an application program on one machine to send a datagram to an application program on another machine. UDP uses the Internet Protocol to deliver datagrams. Conceptually, the important difference between UDP and IP is that UDP messages include a protocol port number, allowing the sender to distinguish among multiple destinations (application programs) on the remote machines. In practice, UDP also includes a checksum over the data being sent.

DCE. Distributed Computing Environment.

default. An alternative value, attribute, or option that is assumed when none has been specified.

DFS. Distributed File System. A subset of the IBM Distributed Computing Environment.

H

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

M

menu. A display of a list of available functions for selection by the user.

Motif. The UNIX industry's standard user interface, originally developed by the Open Systems Foundation. Motif is based on the X-Window system and is a Presentation Manager look-alike. Motif is available for all IBM AIX workstations.

N

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

NFS. Network File System.

node. In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network.

NQS. Network Queueing System.

P

parameter. *(1) A variable that is given a constant value for a specified application and that may denote the application. *(2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. *(3) A name in a procedure that is used to refer to an argument that is passed to the procedure. *(4) A particular piece of information that a system or application program needs to process a request.

process. *(1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. *(2) Any operation or combination of operations on data. *(3) A function being performed or waiting to be performed. *(4) A program in operation. For example, a daemon is a system process that is always running on the system.

S

SDR. Abbreviation for System Data Repository. A repository of system information describing SP hardware and operating characteristics.

server. (1) A function that provides services for users. A machine may run client and server processes at the same time. (2) A machine that provides resources to the network. It provides a network service, such as disk storage and file transfer, or a program that uses such a service.

shell. The shell is the primary user interface for the UNIX operating system. It serves as command language interpreter, programming language, and allows foreground and background processing. There are three different implementations of the shell concept: Bourne, C and Korn.

stream. An internet standard transport level protocol that provides the reliable, full duplex, stream service on which many application protocols depend. TCP allows a process on one machine to send a stream of data to a process on another. It is connection-oriented in the sense that before transmitting data, participants must establish a connection. Software implementing TCP usually resides in the operating system and uses the IP protocol to transmit information across the Internet. It is possible to terminate (shut down) one direction of flow across a TCP connection, leaving a one-way (simplex) connection. The Internet protocol suite is often referred to as TCP/IP because TCP is one of the two most fundamental protocols.

System Administrator. The user who is responsible for setting up, modifying, and maintaining LoadLeveler.

U

user. Anyone who is using LoadLeveler.

W

working directory. All files without a fully qualified path name are relative to this directory.

workstation. *(1) A configuration of input/output equipment at which an operator works. *(2) A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull LoadLeveler V2R2 Using and Administering

Nº Reférence / Reference Nº : 86 A2 14EF 00

Daté / Dated : October 2000

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Technical Publications Ordering Form

Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

BULL CEDOC
ATTN / MME DUMOULIN
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Managers / Gestionnaires :
Mrs. / Mme : **C. DUMOULIN** +33 (0) 2 41 73 76 65
Mr. / M : **L. CHERUBIN** +33 (0) 2 41 73 63 96

FAX : +33 (0) 2 41 73 60 19
E–Mail / Courrier Electronique : srv.Cedoc@franp.bull.fr

Or visit our web sites at: / Ou visitez nos sites web à:

http://www.logistics.bull.net/cedoc

`http://www-frec.bull.com` `http://www.bull.com`

CEDOC Reference # Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	

[_ _] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente

NOM / NAME : ______________________________ Date : __________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

____

PHONE / TELEPHONE : ____________________ FAX : ____________________

E–MAIL : ______________________________

For Bull Subsidiaries / Pour les Filiales Bull :
Identification: ______________________________

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :
Customer Code / Code Client : ______________________________

For Bull Internal Customers / Pour les Clients Internes Bull :
Budgetary Section / Section Budgétaire : ______________________________

For Others / Pour les Autres :
Please ask your Bull representative. / Merci de demander à votre contact Bull.

PLACE BAR CODE IN LOWER LEFT CORNER

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 14EF 00

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

AIX
LoadLeveler V2R2
Using and
Administering

86 A2 14EF 00

AIX
LoadLeveler V2R2
Using and
Administering

86 A2 14EF 00

AIX
LoadLeveler V2R2
Using and
Administering

86 A2 14EF 00