Using Guide (Printable)

The information presented here is also available in individual pages in the Using section. This page is designed to enable you to easily export the information to a PDF or Word document.

To print a version of this document, log in to wikis.sun.com, click Tools, then select Export to PDF or Export to Word.

<table>
<tr><td align="center">Contents</td></tr>
</table>

- Monitoring Hosts
- How to Monitor Hosts From the Command Line
- How to Monitor Hosts With QMON
- Monitoring and Controlling Jobs
- How to Monitor Jobs From the Command Line
- How to Monitor Jobs With QMON
- How to Monitor Jobs by Email
- How to Monitor Jobs by Email With QMON
- How to Control Jobs From the Command Line
- How to Control Jobs With QMON
- Monitoring and Controlling Queues
- How to Monitor Queues With QMON
- How to Control Queues From the Command Line
- Automating Grid Engine Functions Through DRMAA
- Developing With the C Language Binding
- Developing With the Java Language Binding

Using Sun Grid Engine

Using Guide (Printable)

This section focuses on using Sun Grid Engine to perform tasks that distribute workload across your grid systems:

Topic	Description
Interacting With Sun Grid Engine as a User	Learn how you can use the command line interface, the graphical user interface (QMON), and the Distributed Resource Management Application API (DRMAA) to interact with the Sun Grid Engine system.
Displaying User Properties	Learn how to display user properties.
Displaying Host Properties	Learn how to display host properties.

Displaying Queue Properties	Learn how to display queue properties.
Submitting Jobs	Learn how to submit jobs.
Monitoring Hosts	Learn how to monitor and control hosts.
Monitoring and Controlling Jobs	Learn how to monitor and control jobs.
Monitoring and Controlling Queues	Learn how to monitor and control queues.
Using Job Checkpointing	Learn how to use job checkpointing as another method for monitoring jobs.
Using Job to Core Binding	Learn how to bind jobs to processor cores on the execution host.
Starting ARCo	Learn how to gather and view information about how effectively your workload distribution uses resources.

To print this section, see the Using Guide (Printable).

Interacting With Sun Grid Engine as a User

- Launching QMON From the Command Line
- Customizing QMON
- Using the Command-Line Interface

Launching QMON From the Command Line

To launch QMON from the command line, type the following command:

```
qmon
```

Customizing QMON

A specifically designed resource file largely defines the QMON look and feel. Reasonable defaults are compiled in $SGE_ROOT/qmon/Qmon. This file also includes a sample resource file. Refer to the comment lines in the sample Qmon file for detailed information on the possible customizations.

Users can configure the following personal preferences:

- Users can modify the Qmon file.
- The Qmon file can be moved to the home directory or to another location pointed to by the private XAPPLRESDIR search path.
- Users can include the necessary resource definitions in their private .Xdefaults or .Xresources files.
  A private Qmon resource file can also be installed using the xrdb command. The xrdb command can be used during operation. xrdb can also be used at startup of the X11 environment, for example, in a .xinitrc resource file.

You can also use the Job Customize and Queue Customize dialog boxes to customize QMON. These dialog boxes are shown in Customizing the Job Control Display and in Filtering Cluster Queues and Queue Instances. In both dialog boxes, users can use the Save button to store the filtering and display definitions to the .qmon_preferences file in their home directories. When QMON is restarted, this file is read, and QMON reactivates the previously defined behavior.

For information on what your administrator can configure, see Interacting With Sun Grid Engine as an Administrator.

Using the Command-Line Interface

As a user, you will find the following commands particularly useful:

- `qalter` – Modify a pending batch job.
- `qdel` – Delete a queue.
- `qhost` – Show the status of hosts, queues, and jobs.
- `qlogin` – Submit an interactive login session.
- `qrsh` – Submit an interactive `rsh` session.
- `qsub` – Submit Jobs
- `qstat` – Check the status of a job queue.
- `qtcsh` – Used as interactive command interpreter as well as for the processing of `tcsh` shell scripts.

For a complete list of ancillary programs, see Command Line Interface Ancillary Programs. For more information, see the man pages.

Displaying User Properties

- User Access Permissions
- Displaying User Access Permissions
- Displaying Managers, Operators, and Owners

For information on the different categories of Sun Grid Engine users, see Users and User Categories.

User Access Permissions

> **Note**
> The Grid Engine software automatically takes into account the access restrictions configured by the cluster administration. The following sections are important only if you want to query your personal access permission.

The administrator can restrict access to queues and other facilities, such as parallel environment interfaces. Access can also be restricted to certain users or user groups. For more information on how administrators configure access lists, see How to Configure User Access Lists.

Users who belong to ACLs that are listed in access-allowed-lists have permission to access the queue or the parallel environment interface. Users who are members of ACLs in access-denied-lists cannot access the resource in question.

ACLs are also used to define projects, to which assigned users can submit their jobs. The administrator can also restrict access to cluster resources on a per project basis. For more on projects, see Configuring Projects.

The User Configuration dialog box opens when you click the User Configuration button in the QMON Main Control window. This dialog box enables you to query for the ACLs to which you have access. For details, see Managing User Access.

You can display project access by clicking the Project Configuration icon in the QMON Main Control window. Details are described in Configuring Projects.

The ACLs consist of user account names and UNIX group names. The UNIX group names are identified by a prefixed @ sign. In this way, you can determine which ACLs your account belongs to.

> **Note**
> If you have permission to switch your primary UNIX group with the `newgrp` command, your access permissions might change. For details, see the `newgrp(1)` man page.

You can check for those queues or parallel environment interfaces to which you have access or to which your access is denied. Query the queue or parallel environment interface configuration, as described in Displaying Queue Properties and How to Configure Parallel Environments With QMON.

The access-allowed-lists are named `user_lists`. The access-denied-lists are named `xuser_lists`. If your user account or primary UNIX group is associated with an access-allowed-list, you are allowed to access the resource in question. If you are associated with an access-denied-list, you cannot access the queue or parallel environment interface. If both lists are empty, every user with a valid account can access the resource in question.

If you have access to a project, you are allowed to submit jobs that are subordinated to the project. You can submit such jobs from the

command line using the following command:

```
% qsub -P <project-name> <options>
```

The cluster configurations, host configurations, and queue configurations define project access in the same way as for ACLs. These configurations use the `project_lists` and `xproject_lists` parameters for this purpose.

Displaying User Access Permissions

Task	User Interface	Description
How to Display User Access Lists	CLI or QMON	Learn how to display user access lists.
How to Display a List of Defined Projects	CLI or QMON	Learn how to display a list of defined projects.

Displaying Managers, Operators, and Owners

> **Note**
> The superuser of an administration host is considered to be a manager by default.

Task	User Interface	Description
How to Display a List of Managers	CLI or QMON	Learn how to display a list of managers.
How to Display a List of Operators	CLI or QMON	Learn how to display a list of operators.
How to Display a List of Owners	CLI or QMON	Learn how to display a list of owners.

How to Display A List of Managers From the Command Line

To display a list of managers, type the following command:

```
qconf -sm
```

How to Display a List of Managers With QMON

1. Click on the User Configuration button on the QMON Main Control window.

2. Click on the Manager tab
   A list of currently-configured managers are displayed.

How to Display A List of Operators From the Command Line

To display a list of operators, type the following command:

```
qconf -so
```

How to Display a List of Operators With QMON

1. Click on the User Configuration button on the QMON Main Control window.

2. Click on the Operator tab.
    A list of currently-configured operators are displayed.

How to Display a List of Owners From the Command Line

To display a list of owners, type the following command:

```
qconf -sq {<cluster-queue> | <queue-instance> | <queue-domain>}
```

How to Display a List of Owners With QMON

1. Click on the User Configuration button on the QMON Main Control window.

2. *Click on the

How to Display User Access Lists From the Command Line

To display a list of currently configured ACLS, type the following command:

```
qconf -sul
```

To display a list of currently configured ACLS, type the following command:

```
qconf -su <acl-name> [,<...>]
```

How to Display User Access Lists With QMON

1. Click User Configuration on the QMON Main Control window.

2. Click the Userset tab.

This dialog box enables you to query for the ACLs to which you have access.

You can also see what projects to which you have access. For more on projects, see Configuring Projects.

How to Display a List of Defined Projects From the Command Line

To display a list of all defined projects, type the following command:

```
qconf -sprjl
```

To display a specific project configuration, type the following command:

```
qconf -sprj <project-name>
```

How to Display a List of Defined Projects With QMON

Displaying Host Properties

Clicking the `Host Configuration` button in the QMON Main Control window displays an overview of the functionality that is associated with the hosts in your cluster. You need to have manager privileges to apply any changes to the configuration.

The host configuration dialog boxes are described in Configuring Hosts. The following sections describe the commands used to retrieve host information from the command line.

Task	User Interface	Description
How To Display the Name of the Master Host	CLI	Learn how to display the name of the master host.
How to Display a List of Execution Hosts	CLI	Learn how to display a list of execution hosts.
How to Display a List of Administration Hosts	CLI	Learn how to display a list of administration hosts.
How to Display a List of Submit Hosts	CLI	Learn how to display a list of submit hosts.

How to Display the Name of the Master Host From the Command LIne

The location of the master host can migrate between the current master host and one of the shadow master hosts at any time. Therefore, the location of the master host should be transparent to the user.

To display the name of the master host, view `$SGE_ROOT/$SGE_CELL/common/act_qmaster file` in a text editor.

The name of the current master host is listed in the file.

⚙ How to Display a List of Execution Hosts From the Command Line

To display a complete list of the execution hosts in your cluster, type the following command:

```
qconf -sel
```

To display the configuration for a specific execution host, type the following command:

```
qconf -se <hostname>
```

To display status and load information about execution hosts, type the following command:

```
qhost
```

See the host_conf(5) man page for details on the information displayed using qconf. See the qhost(1) man page for details on its output and other options.

⚙ How to Display a List of Administration Host From the Command Line

To display a list of administration hosts, type the following command:

```
qconf -sh
```

⚙ How to Display a List of Submit Hosts From the Command Line

To display a list of submit hosts, type the following command:

```
qconf -ss
```

⚙ Displaying Queue Properties

To make the best use of the Grid Engine system at your site, you should be familiar with the queue structure. You should also be familiar with the properties of the queues that are configured for your Grid Engine system.

Displaying Queues and Queue Properties

Task	User Interface	Description
How to Display a List of Queues	CLI or QMON	Learn how to display a list of currently configured queues.

| How to Display Queue Properties | CLI or QMON | Learn how to display queue properties for a specified queue. |

How to Display a List of Queues From the Command Line

To display a list of queues from the command line, type the following command:

```
% qconf -sql
```

How to Display a List of Queues With QMON

1. Launch the QMON Main Control window.

2. Click the Queue Control button.
   The Cluster Queue Control dialog box appears. Queue Control dialog box provides a quick overview of the installed queues and their current status.

How to Display Queue Properties From the Command Line

To display queue properties from the command line, type the following command:

```
% qconf -sq {<queue> | <queue-instance> | <queue-domain>}
```

Interpreting Queue Property Information

You can find a detailed description of each queue property in the `queue_conf(5)` man page.
The following is a list of some of the more important parameters:

- `qname` – The queue name as requested.
- `hostlist` – A list of hosts and host groups associated with the queue.
- `processors` – The processors of a multiprocessor system to which the queue has access.

> ⛔ Caution
> Do not change this value unless you are certain that you need to change it.

- `qtype` – The type of job that can run in this queue. Currently, the type can be either batch or interactive.
- `slots` – The number of jobs that can be executed concurrently in that queue.
- `owner_list` – The owners of the queue. For more information, see Users and User Categories.
- `user_lists` – The user or group identifiers in the user access lists who can access the queue. For more information, see Displaying User Properties.
- `xuser_lists` – The user or group identifiers in the user access lists who cannot access the queue. For more information, see Displaying User Properties.
- `project_lists` – The jobs submitted with the project identifiers that can access the queue. For more information, see Configuring Projects.
- `xproject_lists` – The jobs submitted with the project identifiers that cannot access the queue. For more information, see Configuring Projects.
- `complex_values` – Assigns capacities as provided for this queue for certain complex resource attributes. For more information, see

Requestable Attributes.

⚙ How to Display Queue Properties With QMON

1. Launch the QMON Main Control window.

2. Click the Queue Control button.
   The Cluster Queue Control dialog box appears.

3. Select a queue, and then click Show Detached Settings.
   The Browser dialog box appears.

4. In the Browser dialog box, click Queue.

5. In the Cluster Queue dialog box, click the Queue Instances tab.

6. Select a queue instance.
   The Browser dialog box lists the queue properties for the selected queue instance.

⚙ Submitting Jobs

A job is a segment of work. Each job includes a description of what to do and a set of property definitions that describe how the job should be run.

The Sun Grid Engine system recognizes the following four basic classes of jobs:

- Batch Jobs – Single segments of work. Typically, a batch job is only executed once.
- Array Jobs – Groups of similar work segments that can all be run in parallel but are completely independent of one another. All of the workload segments of an array job, known as tasks, are identical except for the data sets on which they operate.
- Parallel Jobs – Jobs composed of cooperating tasks that must all be executed at the same time, often with requirements about how the tasks are distributed across the resources.
- Interactive Jobs – Jobs that provide the submitting user with an interactive login to an available resource in the compute cluster. Interactive jobs allow users to execute work on the compute cluster that is not easily submitted as a batch job.

Topic	Description
How Jobs Are Scheduled	Learn how jobs are scheduled using policies and queue selection.
Defining Resource Requirements	Learn how you can define resource requirements for the jobs that you submit.
Requestable Attributes	Learn how to define a requirement profile for the jobs that you submit.
Submitting Batch Jobs	Learn how to submit batch jobs.
Submitting Array Jobs	Learn how to submit array jobs.
Submitting Interactive Jobs	Learn how to submit interactive jobs.
Transparent Remote Execution	Learn about transparent remote execution.

Task	User Interface	Description
How to Submit a Simple Job	CLI or QMON	Learn how to submit a simple job.
How to Submit an Extended Job	CLI or QMON	Learn how to submit an extended job.
How to Submit an Advanced Job	CLI or QMON	Learn how to submit an advanced job.
How to Configure Job Dependencies	CLI	Learn how to configure job dependencies.

How Jobs Are Scheduled

The Sun Grid Engine system schedules jobs using the following process:

1. A scheduling run is triggered in one of the following ways:
    - At a fixed interval. The default is every 15 seconds.
    - By new job submissions or notification from an execution daemon that one or more jobs has finished executing.
    - By using `qconf -tsm`, which an administrator can use to trigger a scheduling run.
2. The scheduler assesses the needs of all pending jobs against available resources by considering the following:

    > ℹ️ If share-based scheduling is used, the calculation takes into account the usage that has already occurred for that user or project.

    - Administrator's specifications for jobs and queues
    - Each pending job's resource requirements (for example, CPU, memory, and I/O bandwidth)
    - Resource reservations that need to be made for future jobs
    - The cluster's current load
    - The host's relative performance
3. As a result of the scheduler's assessment, the Grid Engine system does the following tasks, as needed:
    - Dispatches new jobs
    - Suspends running jobs
    - Increases or decreases the resources allocated to running jobs
    - Maintains the status quo

Between scheduling actions, the Grid Engine system keeps information about significant events such as the following:

- Job submission
- Job completion
- Job cancellation
- An update of the cluster configuration
- Registration of a new machine in the cluster

Usage Policies

The Grid Engine software's policy management automatically controls the use of shared resources in the cluster to best achieve the goals of the administration. High priority jobs are dispatched preferentially and receive better access to resources.

The cluster administrator can define high-level usage policies. The following policies are available:

- Functional – Special treatment is given because of affiliation with a certain user group, project, and so forth.
- Share-based – Level of service depends on an assigned share entitlement, the corresponding shares of other users and user groups, the past usage of resources by all users, and the current presence of users in the system.
- Urgency – Preferential treatment is given to jobs that have greater urgency. A job's urgency is based on its resource requirements, how long the job must wait, and whether the job is submitted with a deadline requirement.
- Override – Manual intervention by the cluster administrator modifies the automated policy implementation.

The Grid Engine software can be set up to routinely use either a share-based policy, a functional policy, or both. These policies can be combined in any proportion, from giving zero weight to one policy and using only the second policy, to giving both policies equal weight. Administrators can temporarily override share-based scheduling and functional scheduling. An override can be applied to an individual job or to all jobs associated with a user, a department, or a project. For more information, see Managing Policies.

Along with the routine policies, jobs can be submitted with an initiation deadline. See the description of the deadline submission parameter under How to Submit an Advanced Job With QMON. Deadline jobs disturb routine scheduling.

Job Priorities

The Grid Engine software also lets users set individual job priorities. A user who submits several jobs can specify, for example, that job 3 is the

most important and that jobs 1 and 2 are equally important but less important than job 3.

Use one of the following options to set priorities:

- QMON Submit Job parameter Priority
- `qsub -p` option.

You can set a priority range of -1023 (lowest) to 1024 (highest). This priority tells the scheduler how to choose among users' jobs when several jobs are in the system simultaneously.

> ℹ Since users are not permitted to submit jobs with a priority higher than 0, which is the default, a best administrative practice is to set the default priority at a lower priority, i.e. `-100`. For more information, see the `sge_request(5)` man page.

Ticket Policies

The functional policy, the share-based policy, and the override policy are all implemented with tickets. Each ticket policy has a ticket pool from which tickets are allocated to jobs that are entering the Grid Engine system. Each routine ticket policy that is in force allocates some tickets to each new job. The ticket policy can reallocate tickets to the executing job at each scheduling interval.

Tickets weight the three ticket policies. For example, if no tickets are allocated to the functional policy, then that policy is not used. If an equal number of tickets are assigned to the functional ticket pool and to the share-based ticket pool, then both policies have equal weight in determining a job's importance.

The following are criteria that each ticket policy uses to allocate tickets:

- Grid Engine managers allocate tickets to the routine ticket policies at system configuration. Managers and operators can change ticket allocations at any time. Additional tickets can be injected into the system temporarily to indicate an override. Ticket policies can be combined when tickets are allocated to multiple ticket policies, a job gets a portion of its tickets from each ticket policy.
- The Grid Engine system grants tickets to jobs that are entering the system to indicate their importance under each ticket policy. Each running job can gain tickets, for example, from an override; lose tickets, for example, because the job is getting more than its fair share of resources; or keep the same number of tickets at each scheduling interval. The number of tickets that a job holds represents the resource share that the Grid Engine system tries to grant that job during each scheduling interval.

You can display the number of tickets a job holds with QMON or using `qstat -ext`. See How to Monitor and Control Jobs With QMON. The `qstat` command also displays the priority value assigned to a job, for example, using `qsub -p`. See the qstat(1) man page for more details.

Queue Selection

Jobs that are submitted to a named queue go directly to the named queue, regardless of whether the jobs can be started or need to be spooled. Jobs that are not submitted to a named queue that cannot be started immediately are put into a spool. The `sge_qmaster` then tries to reschedule the jobs until a suitable queue becomes available, allowing the jobs to be dispatched. Therefore, viewing the queues of the Grid Engine system as computer science batch queues is valid only for jobs requested by name. Jobs submitted with nonspecific requests use the spooling mechanism of `sge_qmaster` for queueing, thus using a more abstract and flexible queuing concept.

If a job is scheduled and multiple free queues meet its resource requests, the job is usually dispatched to a suitable queue belonging to the least loaded host. By setting the scheduler configuration entry `queue_sort_method` to `seq_no`, the cluster administration can change this load-dependent scheme into a fixed order algorithm. The queue configuration entry `seq_no` defines a precedence among the queues, assigning the highest priority to the queue with the lowest sequence number.

⚙ Defining Resource Requirements

In the examples so far, the submit options do not express any resource requirements for the hosts on which the jobs are to be executed. The Grid Engine system assumes that such jobs can be run on any host. In practice, however, most jobs require that certain prerequisites be met on the executing host in order for the job to finish successfully. These prerequisites include:

- Enough available memory
- Installation of required software

- Certain operating system architecture

Also, the cluster administrator usually imposes restrictions on the use of the machines in the cluster. For example, the CPU time that can be consumed by the jobs is often restricted.

The Grid Engine system provides users with the means to find suitable hosts for their jobs without precise knowledge of the cluster`s equipment and its usage policies. Users specify the requirement of their jobs and let the Grid Engine system manage the task of finding a suitable and lightly loaded host.

You specify resource requirements through requestable attributes, which are described in Requestable Attributes. QMON provides a convenient way to specify the requirements of a job. The Requested Resources dialog box displays only those attributes in the Available Resource list that are currently eligible. Click Request Resources in the Submit Job dialog box to open the Requested Resources dialog box.

When you double-click an attribute, the attribute is added to the Hard or Soft Resources list of the job. A dialog box opens to guide you in entering a value specification for the attribute in question, except for BOOLEAN attributes, which are set to True. For more information, see How the Grid Engine System Allocates Resources.

Figure – Requested Resources Dialog Box shows a resource profile for a job that requests a `solaris64` host with an available `permas` license offering at least 750 MBytes of memory. If more than one queue that fulfills this specification is found, any defined soft resource requirements are taken into account. However, if no queue satisfying both the hard and the soft requirements is found, any queue that grants the hard requirements is considered suitable.

> **ℹ Note**
> The `queue_sort_method` parameter of the scheduler configuration determines where to start the job only if more than one queue is suitable for a job. See the `sched_conf(5)` man page for more information.

The attribute `permas`, an integer, is an administrator extension to the global resource attributes. The attribute `arch`, a string, is a host resource attribute. The attribute `h_vmem`, memory, is a queue resource attribute.

An equivalent resource requirement profile can as well be submitted from the `qsub` command line:

```
% qsub -l arch=solaris64,h_vmem=750M,permas=1 \
    permas.sh
```

The implicit `-hard` switch before the first `-l` option has been skipped.

The notation `750M` for 750 MBytes is an example of the quantity syntax of the Grid Engine system. For those attributes that request a memory consumption, you can specify either integer decimal, floating-point decimal, integer octal, and integer hexadecimal numbers. The following multipliers must be appended to these numbers:

- k – Multiplies the value by 1000
- K – Multiplies the value by 1024
- m – Multiplies the value by 1000 times 1000
- M – Multiplies the value by 1024 times 1024

Octal constants are specified by a leading zero and digits ranging from 0 to 7 only. To specify a hexadecimal constant, you must prefix the number with 0x. You must also use digits ranging from 0 to 9, a through f, and A through F. If no multipliers are appended, the values are considered to count as bytes. If you are using floating-point decimals, the resulting value is truncated to an integer value.

For those attributes that impose a time limit, you can specify time values in terms of hours, minutes, or seconds, or any combination. Hours, minutes, and seconds are specified in decimal digits separated by colons. A time of `3:5:11` is translated to 11111 seconds. If zero is a specifier for hours, minutes, or seconds, you can leave it out if the colon remains. Thus a value of `:5:` is interpreted as 5 minutes. The form used in the Requested Resources dialog box that is shown in Figure – Requested Resources Dialog Box is an extension, which is valid only within QMON.

How the Grid Engine System Allocates Resources

Knowing how the Grid Engine software processes resource requests and allocates resources is important. The resource allocation algorithm that Grid Engine software uses is as follows:

1. Read in and parse all default request files. See Default Request Files for details.
2. Process the script file for embedded options. See Active Comments for details.
3. Read all script-embedding options when the job is submitted, regardless of their position in the script file.
4. Read and parse all requests from the command line.

As soon as all `qsub` requests are collected, hard and soft requests are processed separately.

The requests are evaluated in the following order of precedence:

1. From left to right of the script or default request file.
2. From top to bottom of the script or default request file.
3. From left to right of the command line. In other words, you can use the command line to override the embedded flags.

Hard requests are processed first. If a hard request is not valid, the submission is rejected. If one or hard more requests cannot be met at submit time, the job is spooled and rescheduled to be run at a later time. For example, a hard request might not be met if a requested queue is busy. If all hard requests can be met, the resources are allocated and the job can be run.

The soft resource requests are then checked. The job can run even if some or all of these requests cannot be met. If multiple queues that meet the hard requests provide parts of the soft resources list, the Grid Engine software selects the queues that offer the most soft requests.

The job is started and covers the allocated resources.

You might want to gather experience of how argument list options and embedded options or hard and soft requests influence each other. You can experiment with small test script files that execute UNIX commands such as `hostname` or `date`.

Requestable Attributes

When you submit a job, a requirement profile can be specified. You can specify attributes or characteristics of a host or queue that the job requires to run successfully.

The attributes that can be used to specify the job requirements are related to one of the following:

- The cluster, for example, space required on a network shared disk
- Individual hosts, for example, operating system architecture
- Queues, for example, permitted CPU time

The attributes can also be derived from site policies such as the availability of installed software only on certain hosts.

The available attributes include the following:

- Queue property list – See Displaying Queue Properties
- List of global and host-related attributes – See Assigning Resource Attributes to Queues, Hosts, and the Global Cluster
- Administrator-defined attributes

For convenience, however, the administrator commonly chooses to define only a subset of all available attributes to be requestable.

The Grid Engine system complex contains the definitions for all resource attributes. For more information about resource attributes, see Configuring Resource Attributes. See also the complex format description on the `complex(5)` man page.

Task	User Interface	Description
How to Display Requestable Attributes	CLI or QMON	Learn how to display requestable attributes.

How to Display Requestable Attributes From the Command Line

From the command line, type the following:

```
% qconf -sc
```

The following example shows sample output from the `qconf -sc` command:

```
gimli% qconf -sc
#name                shortcut    type        relop requestable consumable default  urgency
#--------------------------------------------------------------------------------
arch                 a           RESTRING    ==    YES         NO         NONE     0
calendar             c           STRING      ==    YES         NO         NONE     0
cpu                  cpu         DOUBLE      >=    YES         NO         0        0
h_core               h_core      MEMORY      <=    YES         NO         0        0
h_cpu                h_cpu       TIME        <=    YES         NO         0:0:0    0
h_data               h_data      MEMORY      <=    YES         NO         0        0
h_fsize              h_fsize     MEMORY      <=    YES         NO         0        0
h_rss                h_rss       MEMORY      <=    YES         NO         0        0
h_rt                 h_rt        TIME        <=    YES         NO         0:0:0    0
h_stack              h_stack     MEMORY      <=    YES         NO         0        0
h_vmem               h_vmem      MEMORY      <=    YES         NO         0        0
hostname             h           HOST        ==    YES         NO         NONE     0
load_avg             la          DOUBLE      >=    NO          NO         0        0
load_long            ll          DOUBLE      >=    NO          NO         0        0
load_medium          lm          DOUBLE      >=    NO          NO         0        0
load_short           ls          DOUBLE      >=    NO          NO         0        0
mem_free             mf          MEMORY      <=    YES         NO         0        0
mem_total            mt          MEMORY      <=    YES         NO         0        0
mem_used             mu          MEMORY      >=    YES         NO         0        0
min_cpu_interval     mci         TIME        <=    NO          NO         0:0:0    0
np_load_avg          nla         DOUBLE      >=    NO          NO         0        0
np_load_long         nll         DOUBLE      >=    NO          NO         0        0
np_load_medium       nlm         DOUBLE      >=    NO          NO         0        0
np_load_short        nls         DOUBLE      >=    NO          NO         0        0
num_proc             p           INT         ==    YES         NO         0        0
qname                q           STRING      ==    YES         NO         NONE     0
rerun                re          BOOL        ==    NO          NO         0        0
s_core               s_core      MEMORY      <=    YES         NO         0        0
s_cpu                s_cpu       TIME        <=    YES         NO         0:0:0    0
s_data               s_data      MEMORY      <=    YES         NO         0        0
s_fsize              s_fsize     MEMORY      <=    YES         NO         0        0
s_rss                s_rss       MEMORY      <=    YES         NO         0        0
s_rt                 s_rt        TIME        <=    YES         NO         0:0:0    0
s_stack              s_stack     MEMORY      <=    YES         NO         0        0
s_vmem               s_vmem      MEMORY      <=    YES         NO         0        0
seq_no               seq         INT         ==    NO          NO         0        0
slots                s           INT         <=    YES         YES        1        1000
swap_free            sf          MEMORY      <=    YES         NO         0        0
swap_rate            sr          MEMORY      >=    YES         NO         0        0
swap_rsvd            srsv        MEMORY      >=    YES         NO         0        0
swap_total           st          MEMORY      <=    YES         NO         0        0
swap_used            su          MEMORY      >=    YES         NO         0        0
tmpdir               tmp         STRING      ==    NO          NO         NONE     0
virtual_free         vf          MEMORY      <=    YES         NO         0        0
virtual_total        vt          MEMORY      <=    YES         NO         0        0
virtual_used         vu          MEMORY      >=    YES         NO         0        0
# >#< starts a comment but comments are not saved across edits --------
```

The column `name` is identical to the first column displayed by the `qconf -sq` command. The `shortcut` column contains administrator-definable abbreviations for the full names in the first column. The user can supply either the full name or the shortcut in the request option of a `qsub` command.

The column `requestable` tells whether the resource attribute can be used in a `qsub` command. The administrator can, for example, disallow the cluster's users to request certain machines or queues for their jobs directly. The administrator can disallow direct requests by setting the entries `qname`, `hostname`, or both, to be unrequestable. Making queues or hosts unrequestable implies that feasible user requests can be met in general by multiple queues, which enforces the load balancing capabilities of the Grid Engine system.

The column `relop` defines the relational operator used to compute whether a queue or a host meets a user request. The comparison that is executed is as follows:

```
User_Request     relop     Queue/Host/... -Property
```

If the result of the comparison is false, the user's job cannot be run in the queue or on the host. For example, let the queue `q1` be configured with a soft CPU time limit of 100 seconds. Let the queue `q2` be configured to provide 1000 seconds soft CPU time limit. See the queue_conf (5) and the setrlimit(2) man pages for a description of user process limits.

The columns `consumable` and `default` affect how the administrator declares consumable resources. See Consumable Resources.

The user requests consumables just like any other attribute. The Grid Engine system internal bookkeeping for the resources is different, however.

Assume that a user submits the following request:

```
% qsub -l s_cpu=0:5:0 nastran.sh
```

The `s_cpu=0:5:0` request asks for a queue that grants at least 5 minutes of soft limit CPU time. Therefore, only queues providing at least 5 minutes soft CPU runtime limit are set up properly to run the job. See the qsub(1) man page for details on the syntax.

> ℹ️ Note
> The Grid Engine software considers workload information in the scheduling process only if more than one queue or host can run a job.

How to Display Requestable Attributes With QMON

1. Click the Job Control button in the QMON Main Control window.
   The Job Control dialog box appears.

2. Select a pending job and and click the Submit button.
   The Submit Job dialog box appears.

3. Click the Request Resources button.
   The Requested Resources dialog box displays the currently requestable attributes under Available Resources, which is shown in the following figure.
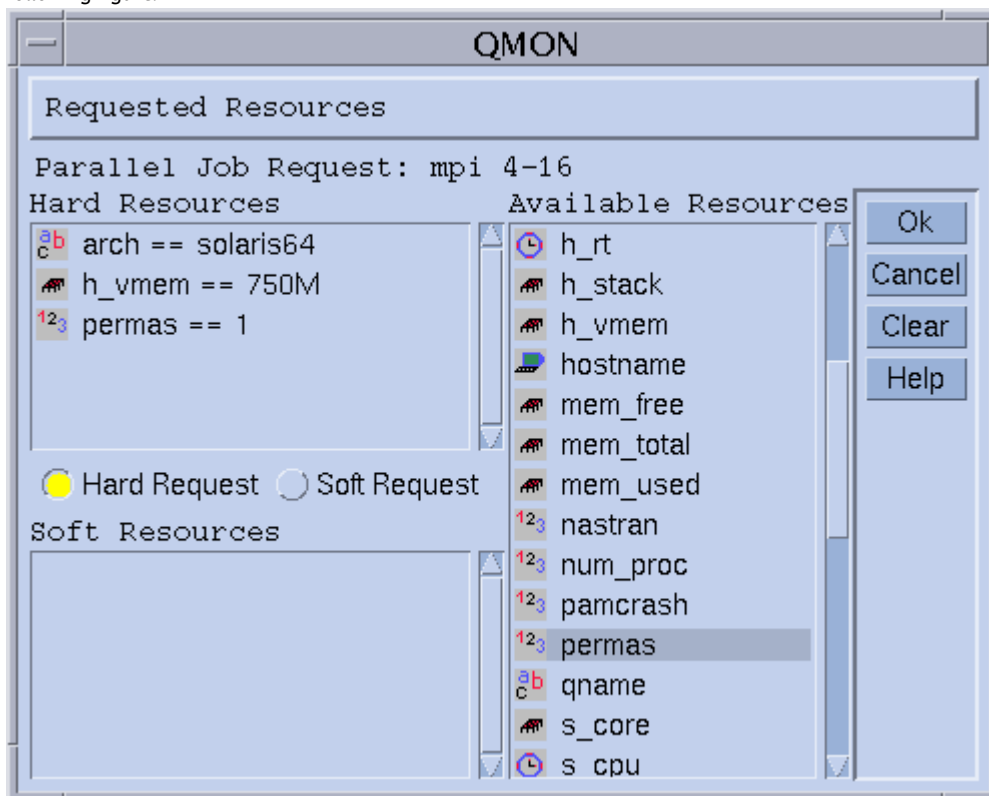
Submitting Batch Jobs

The following sections describe how to submit more complex jobs through the Grid Engine system:

- About Shell Scripts
- Example of a Shell Script
- Extensions to Regular Shell Scripts

For information about submitting simple jobs, see How to Submit a Simple Job.

About Shell Scripts

Shell scripts, also called batch jobs, are a sequence of command-line instructions that are assembled in a file. Each instruction is interpreted as if the instruction were typed manually by the user who is running the script. You can invoke arbitrary commands, applications, and other shell scripts from within a shell script.

Script files are made executable by the `chmod` command. If scripts are invoked, a command interpreter is started. `csh`, `tcsh`, `sh`, or `ksh` are typical command interpreters.

The command interpreter can be invoked as login shell. To do so, the name of the command interpreter must be contained in the `login_shells` list of the Grid Engine system configuration that is in effect for the particular host and queue that is running the job.

> **Note**
> The Grid Engine system configuration might be different for the various hosts and queues configured in your cluster. You can display the effective configurations with the `-sconf` and `-sq` options of the `qconf` command. For detailed information, see the qconf(1) man page.

If the command interpreter is invoked as login shell, your job environment is the same as if you logged in and ran the script. In using `csh`, for example, `.login` and `.cshrc` are executed in addition to the system default startup resource files, such as `/etc/login`, whereas only `.cshrc` is executed if `csh` is not invoked as `login-shell`. For a description of the difference between being invoked and not being invoked as `login-shell`, see the man page for your command interpreter.

Example of a Shell Script

The following example is a simple shell script that compiles the application `flow` from its Fortran77 source and then runs the application:

```
#!/bin/csh
# This is a sample script file for compiling and
# running a sample FORTRAN program under N1 Grid Engine 6
cd TEST
# Now we need to compile the program "flow.f" and
# name the executable "flow".
f77 flow.f -o flow
```

Your local system user's guide provides detailed information about building and customizing shell scripts. You might also want to look at the `sh`, `ksh`, `csh`, or `tcsh` man pages. The following sections emphasize special things that you should consider when you prepare batch scripts for the Grid Engine system.

In general, you can submit all shell scripts to the Grid Engine system that you can run from your command prompt by hand. These shell scripts must not require a terminal connection or need interactive user intervention. The exceptions are the standard error and standard output devices, which are automatically redirected.

Extensions to Regular Shell Scripts

Some extensions to regular shell scripts influence the behavior of scripts that run under Grid Engine system control. The following sections describe these extensions.

How a Command Interpreter Is Selected

At submit time, you can specify the command interpreter to use for the job script file as shown in Figure – Extended Job Submission Example. However, if nothing is specified, the configuration variable `shell_start_mode` determines how the command interpreter is selected:

- If `shell_start_mode` is set to `unix_behavior`, the first line of the script file specifies the command interpreter. The first line of the script file must begin with a pound symbol (`#`) followed by an exclamation point (`!`). If the first line does not begin with those characters, the Bourne Shell `sh` is used by default.
- For all other settings of `shell_start_mode`, the default command interpreter is determined by the `shell` parameter for the queue where the job starts. See Displaying Queue Properties and the `queue_conf(5)` man page.

Output Redirection

Since batch jobs do not have a terminal connection, their standard output and their standard error output must be redirected into files. The Grid Engine system enables the user to define the location of the files to which the output is redirected. Defaults are used if no output files are specified.

The standard location for the files is in the current working directory where the jobs run. The default standard output file name is job-name.o job-id. The default standard error output is redirected to job-name>.ejob-id. The job-name can be built from the script file name, or defined by the user. See, for example, the `-N` option in the submit(1) man page. job-id is a unique identifier that is assigned to the job by the Grid Engine system.

For array job tasks, the task identifier is added to these filenames, separated by a dot. The resulting standard redirection paths are job-name.o job-id.task-id> and job-name.ejob-id.task-id. For more information, see Submitting Array Jobs.

If the standard locations are not suitable, you can use one of the following to specify output directions:

- QMON as shown in Figure – Advanced Job Submission Example
- `-e` and `-o` options to the `qsub` command

Standard output and standard error output can be merged into one file. The redirections can be specified on a per execution host basis, in which case, the location of the output redirection file depends on the host on which the job is executed. To build custom but unique redirection file paths, use dummy environment variables together with the `qsub -e` and `-o` options. A list of these variables follows:

- `HOME` – Home directory on execution machine
- `USER` – User ID of job owner
- `JOB_ID` – Current job ID
- `JOB_NAME` – Current job name; see the `-N` option
- `HOSTNAME` – Name of the execution host
- `TASK_ID` – Array job task index number

When the job runs, these variables are expanded into the actual values, and the redirection path is built with these values. See the `qsub(1)` man page for further details.

Active Comments

Lines with a leading `#` sign are treated as comments in shell scripts. The Grid Engine system also recognizes special comment lines that supply options to commands or to the QMON interface. By default, these special comment lines are identified by the `#$` prefix string. You can redefine the prefix string with the `qsub -C` command.

This use of special comments is referred to as "script embedding of submit arguments." The following example shows a script file that uses script-embedded command-line options to supply arguments to the `qsub` command. These options also apply to the QMON Submit Job dialog box. The corresponding parameters are preset when a script file is selected.

Example – Using Script-Embedded Command Line Options

```csh
#!/bin/csh

#Force csh if not Grid Engine default
#shell

#$ -S /bin/csh

# This is a sample script file for compiling and
# running a sample FORTRAN program under N1 Grid Engine 6
# We want Grid Engine to send mail
# when the job begins
# and when it ends.

#$ -M EmailAddress
#$ -m b e

# We want to name the file for the standard output
# and standard error.

#$ -o flow.out -j y

# Change to the directory where the files are located.

cd TEST

# Now we need to compile the program "flow.f" and
# name the executable "flow".

f77 flow.f -o flow

# Once it is compiled, we can run the program.

flow
```

Environment Variables

> **Note**
> If you would to change the predefined values of these variables, use the `-V` or `-v` options with `qsub` or `qalter`. For more information, see the `qsub(1)` man page.

When a job runs, the following variables are preset into the job's environment:

- `ARC` – The architecture name of the node on which the job is running. The name is compiled into the `sge_execd` binary.
- `$SGE_ROOT` – The root directory of the Grid Engine system as set for `sge_execd` before startup, or the default `/usr/SGE` directory.
- `SGE_BINARY_PATH` – The directory in which the Grid Engine system binaries are installed.
- `$SGE_CELL` – The cell in which the job runs.
- `SGE_JOB_SPOOL_DIR` – The directory used by `sge_shepherd` to store job-related data while the job runs.
- `SGE_O_HOME` – The path to the home directory of the job owner on the host from which the job was submitted.
- `SGE_O_HOST` – The host from which the job was submitted.
- `SGE_O_LOGNAME` – The login name of the job owner on the host from which the job was submitted.
- `SGE_O_MAIL` – The content of the `MAIL` environment variable in the context of the job submission command.
- `SGE_O_PATH` – The content of the `PATH` environment variable in the context of the job submission command.
- `SGE_O_SHELL` – The content of the `SHELL` environment variable in the context of the job submission command.
- `SGE_O_TZ` – The content of the `TZ` environment variable in the context of the job submission command.
- `SGE_O_WORKDIR` – The working directory of the job submission command.
- `SGE_CKPT_ENV` – The checkpointing environment under which a checkpointing job runs. The checkpointing environment is selected with the `qsub -ckpt` command.
- `SGE_CKPT_DIR` – The path `ckpt_dir` of the checkpoint interface. Set only for checkpointing jobs. For more information, see the `checkpoint(5)` man page.
- `SGE_STDERR_PATH` – The path name of the file to which the standard error stream of the job is diverted. This file is commonly used for enhancing the output with error messages from prolog, epilog, parallel environment start and stop scripts, or checkpointing scripts.
- `SGE_STDOUT_PATH` – The path name of the file to which the standard output stream of the job is diverted. This file is commonly used for enhancing the output with messages from prolog, epilog, parallel environment start and stop scripts, or checkpointing scripts.
- `SGE_TASK_ID` – The unique index number for an array job task. You can use the `SGE_TASK_ID` to reference various input data

records. This environment variable is set to `undefined` for non-array jobs. It is possible to change the predefined value of this variable with the `-v` or `-V` submit option. For more information, see the `qsub(1)` man page.

- `SGE_TASK_FIRST` – The index number of the first array job task. For more information, see the `-t` option for `qsub`. It is possible to change the predefined value of this variable with the `-v` or `-V` submit option.
- `SGE_TASK_LAST` – The index number of the last array job task. For more information, see the `-t` option for `qsub`. It is possible to change the predefined value of this variable with the `-v` or `-V` submit option.
- `SGE_TASK_STEPSIZE` – The step size of the array job specification. For more information, see the `-t` option for `qsub`. It is possible to change the predefined value of this variable with the `-v` or `-V` submit option.
- `ENVIRONMENT` – Always set to `BATCH`. This variable indicates that the script is run in batch mode.
- `HOME` – The user's home directory path as taken from the `passwd` file.
- `HOSTNAME` – The host name of the node on which the job is running.
- `JOB_ID` – A unique identifier assigned by the `sge_qmaster` daemon when the job was submitted. The job ID is a decimal integer from 1 through 9,999,999.
- `JOB_NAME` – The job name, which is built from the file name provided with the `qsub` command, a period, and the digits of the job ID. You can override this default with `qsub -N`.
- `LOGNAME` – The user's login name as taken from the `passwd` file.
- `NHOSTS`-- The number of hosts in use by a parallel job.
- `NQUEUES` – The number of queues that are allocated for the job. This number is always 1 for serial jobs.
- `NSLOTS` – The number of queue slots in use by a parallel job.
- `PATH` – A default shell search path of: `/usr/local/bin:/usr/ucb:/bin:/usr/bin`.
- `PE` – The parallel environment under which the job runs. This variable is for parallel jobs only.
- `PE_HOSTFILE` – The path of a file that contains the definition of the virtual parallel machine that is assigned to a parallel job by the Grid Engine system. This variable is used for parallel jobs only. See the description of the `$pe_hostfile` parameter in `sge_pe` for details on the format of this file.
- `QUEUE` – The name of the queue in which the job is running.
- `REQUEST` – The request name of the job. The name is either the job script file name or is explicitly assigned to the job by the `qsub -N` command.
- `RESTARTED` – Indicates whether a checkpointing job was restarted. If set to value 1, the job was interrupted at least once. The job is therefore restarted.
- `SHELL` – The user's login shell as taken from the `passwd` file.

> 🛈 **Note**
> `SHELL` is not necessarily the shell that is used for the job.

- `TMPDIR` – The absolute path to the job's temporary working directory.
- `TMP` – The same as `TMPDIR`. This variable is provided for compatibility with NQS.
- `TZ` – The time zone variable imported from `sge_execd`, if set.
- `USER` – The user's login name as taken from the `passwd` file.

Submitting Array Jobs

Submitting Array Jobs

Parameterized and repeated execution of the same set of operations that are contained in a job script is an ideal application for the array job facility of the Grid Engine system. Typical examples of such applications are found in the Digital Content Creation industries for tasks such as rendering. Computation of an animation is split into frames. The same rendering computation can be performed for each frame independently.

The Grid Engine system provides an efficient implementation of array jobs, handling the computations as an array of independent tasks joined into a single job. The tasks of an array job are referenced through an array index number. The indexes for all tasks span an index range for the entire array job. The index range is defined during submission of the array job by a single `qsub` command.

You can monitor and control an array job. For example, you can suspend, resume, or cancel an array job as a whole or by individual task or subset of tasks. To reference the tasks, the corresponding index numbers are suffixed to the job ID. Tasks are executed very much like regular jobs. Tasks can use the environment variable `SGE_TASK_ID` to retrieve its own task index number and to access input data sets designated for this task identifier.

Task	User Interface	Description

How to Submit an Array Job	CLI or QMON	
How to Configure Array Job Dependencies	CLI	

How to Configure Array Task Dependencies From the Command Line

While most interdependent tasks can be supported by Sun Grid Engine's job dependency facility, certain array jobs require the flexibility provided by the array task dependency facility. The array task dependency facility allows users to make one array job's tasks dependent on the tasks of another array job. For example, if you use Sun Grid Engine to render video effects, the array task dependency allows you to submit each step as an array job where each task represents a frame. Each task then depends on the corresponding task in the previous step.

To configure an array task dependency, use the following command:

```
qsub -hold_jid_ad wc_job_list
```

The -hold_jid_ad option defines or redefines the job array dependency list of the submitted job. A reference by job name or pattern is only accepted if the referenced job is owned by the same user as the referring job. Each sub-task of the submitted job is not eligible for execution unless the corresponding sub-tasks of all jobs referenced in the comma-separated job id and/or job name list have completed.

For more on the -hold_jid_ad option, see the qsub(1) man page. The wc_job_list type is detailed in sge_types(1).

Examples – Using Job Dependencies Versus Array Task Dependencies to Complete Array Jobs

The following example illustrates the difference between the job dependency facility and the task array dependency facility:

- In the following example, array task B is dependent on array task A:

    ```
    $ qsub -t 1-3 A
    $ qsub -hold_jid A -t 1-3 B
    ```

    All the sub-tasks in job B will wait for all sub-tasks 1,2 and 3 in A to finish before starting the tasks in job B. The tasks will be executed in the following approximate order: A.1, A.2, A.3, B.1, B.2, B.3, as shown below:

    ```
    | A.1 |     | B.1 |
    | A.2 | --> | B.2 |
    | A.3 |     | B.3 |
    ```

- In the following example, each sub-task in array job B is dependent on each corresponding sub-task in job A in a one-to-one mapping:

    ```
    $ qsub -t 1-3 A
    $ qsub -hold_jid_ad A -t 1-3 B
    ```

    Sub-task B.1 will only start when A.1 completes. B.2 will only start once A.2 completes, etc. On a single machine renderfarm, the tasks thus could be executed in the following approximate order: A.1, B.1, A.2, B.2, A.3, B.3, as shown below:

    ```
    | A.1 | --> | B.1 |
    | A.2 | --> | B.2 |
    | A.3 | --> | B.3 |
    ```

    It should only be able to specify the option if we are submitting an array job, it is dependent on another array job, and that array job has the same number of sub-tasks.

Examples – Using Array Task Dependencies to Chunk Tasks

When using 3D rendering applications, it is often more efficient to render several frames at once on the same CPU instead of distributing the frames across several machines. The generation of several frames at once we will refer to as chunking.

> ℹ When using the task dependency facility, the array task must have the same range of sub-tasks as its dependent array task, otherwise the job will be rejected at submit time.

The following examples illustrate chunking:

- Array task `B` is dependent on array task `A`, which has a step size of 2:

  ```
  $ qsub -t 1-6:2 A
  $ qsub -hold_jid_ad A -t 1-6 B
  ```

  In the results shown below, it is assumed that array task `A` is chunking, which means that `B.1` and `B.2` are dependent on `A.1`, `B.3` and `B.4` are dependent on `A.3`, and so on. If job `A.1` didn't render frame 2, then job `B.2` would fail:

  ```
  | A.1 | --> | B.1 |
  |     | --> | B.2 |
  | A.3 | --> | B.3 |
  |     | --> | B.4 |
  | A.5 | --> | B.5 |
  |     | --> | B.6 |
  ```

- Array task `B` is dependent on array task `A`, which has a step size of 1:

  ```
  $ qsub -t 1-6 A
  $ qsub -hold_jid_ad A -t 1-6:2 B
  ```

  In this example shown below, array task `B` is chunking, which means that job `B.1` is dependent on job `A.1` and job `A.2`, job `B.3` is dependent on job `A.3` and job `A.4`, and so on. It is reasonable to always assume that array task `B` is chunking because otherwise `A.2`, `A.4`, and `A.6` would be needlessly run and the result would never be used:

  ```
  | A.1 | --> | B.1 |
  | A.2 | --> |     |
  | A.3 | --> | B.3 |
  | A.4 | --> |     |
  | A.5 | --> | B.5 |
  | A.6 | --> |     |
  ```

- Array task `A` has a step size of 3 and array task `B` has a step size of 2. The tasks are dependent on each other:

  ```
  $ qsub -t 1-6:3 A
  $ qsub -hold_jid_ad A -t 1-6:2 B
  ```

  In this example shown below, both array task `A` and array task `B` are chunking. So, job `B.1` is dependent on job `A.1`, job `B.3` is dependent on job `A.1` and job `A.4`, and job `B.5` is dependent on job `A.4`. When the hold array dependency option `-hold_jid_ad` is specified and the step sizes of the array job and the dependent array job are different, we always assume that both are chunking:

  ```
  | A.1 | --> | B.1 |
  |     | --> |     |
  |     | --> | B.3 |
  | A.4 | --> |     |
  |     | --> | B.5 |
  |     | --> |     |
  ```

How to Submit an Array Job From the Command Line

To submit an array job from the command line, type the following command:

```
qsub  -t <n[-m[:s]]> <job.sh>
```

The `-t` option defines the task index range. The `n[-m[:s]]` argument indicates the following:

- The lowest index number (`n`)
- The highest index number (`m`)
- The step size (`s`)

The range may be a single number (n), a simple range (n-m), or a range with a step size (n-m:s).

For more information, see the `qsub(1)` man page.

Example – Array Job

The following is an example of how to submit an array job:

```
% qsub -l h_cpu=0:45:0 -t 2-10:2 render.sh data.in
```

Each task requests a hard CPU time limit of `45` minutes with the `-l` option. The `-t` option defines the task index range. In this case, `2-10:2` specifies that `2` is the lowest index number, and `10` is the highest index number. Only every second index, the `:2` part of the specification, is used. Thus, the array job is made up of 5 tasks with the task indices 2, 4, 6, 8, and 10. Each task executes the job script `render.sh` once the task is dispatched and started by the Grid Engine system. Tasks can use `SGE_TASK_ID` to find their index number, which they can use to find their input data record in the data file `data.in`.

How to Submit an Array Job With QMON

To submit an array job, follow the instructions in How to Submit a Simple Job With QMON, additionally taking into account the following information.

The only difference is that the Job Tasks input window that is shown in Figure – Extended Job Submission Example must contain the task range specification. The task range specification uses syntax that is identical to the `qsub -t` command. See the `qsub(1)` man page for detailed information about array index syntax.

For information about monitoring and controlling jobs in general, and about array jobs in particular, see Monitoring and Controlling Jobs. See also the man pages for qstat(1), qhold(1), qrls(1), qmod(1), and qdel(1).

> **ⓘ** Note
> Array tasks cannot have interdependencies with other jobs or with other array tasks.

Submitting Interactive Jobs

The submission of interactive jobs instead of batch jobs is useful in situations where a job requires your direct input to influence the job results. Such situations are typical for X Windows applications or for tasks in which your interpretation of immediate results is required to steer further processing.

You can create interactive jobs in three ways:

- `qlogin` – An rlogin-like session that is started on a host selected by the Grid Engine software.
- `qrsh` – The equivalent of the standard UNIX `rsh` facility. A command is run remotely on a host selected by the Grid Engine system. If no command is specified, a remote `rlogin` session is started on a remote host.
- `qsh` – An `xterm` that is displayed from the machine that is running the job. The display is set corresponding to your specification or to the setting of the `DISPLAY` environment variable. If the `DISPLAY` variable is not set, and if no display destination is defined, the Grid Engine system directs the `xterm` to the 0.0 screen of the X server on the host from which the job was submitted.

> ℹ **Note**
> Contact your system administrator to find out if your cluster is prepared for interactive job execution. To function correctly, all the facilities need proper configuration of cluster parameters of the Grid Engine system. The correct `xterm` execution paths must be defined for `qsh`. Interactive queues must be available for this type of job.

The default handling of interactive jobs differs from the handling of batch jobs. Interactive jobs are not queued if the jobs cannot be executed when they are submitted. When a job is not queued immediately, the user is notified that the cluster is currently too busy.

You can change this default behavior with the `-now no` option to `qsh`, `qlogin`, and `qrsh`. If you use this option, interactive jobs are queued like batch jobs. When you use the `-now yes` option, batch jobs that are submitted with `qsub` can also be handled like interactive jobs. Such batch jobs are either dispatched for running immediately, or they are rejected.

> ℹ **Note**
> Interactive jobs can be run only in queues of the type `INTERACTIVE`. See Configuring Queues for details.

The following sections describe how to use the `qlogin` and `qsh` facilities. The `qrsh` command is explained in a broader context in Transparent Remote Execution.

Task	User Interface	Description
How to Submit Interactive Jobs	CLI or QMON	Learn how to submit interactive jobs.

How to Submit Interactive Jobs From the Command Line

> ℹ **Note**
> The output for an interactive job cannot be redirected with the `-j y|n`, `-o`, and `-e` options. However, since the output for a prolog and epilog script is sent to the default `stdout` and `stderr` files, you can use the `-j y|n`, `-o`, and `-e` options to redirect this output to different files. For more information, see the `qsub(1)` man page.

Using **qrsh** to Submit Interactive Jobs

`qrsh` supports most of the `qsub` options. If no options are given, `qrsh` will open an `rlogin`-like session.

To submit an interactive job with the **qrsh** command, type a command like the following:

```
qrsh -pty y vi
```

This command starts a `vi` editor on any available system in the Sun Grid Engine cluster. The `-pty y` option starts a job in a pseudo-terminal session. The pseudo-terminal allows full cursor control from within the `vi` session.

Using **qsh** to Submit Interactive Jobs

`qsh` is very similar to `qsub`. `qsh` supports several of the `qsub` options, as well as the additional option `-display` to direct the display of the `xterm` to be invoked. See the `qsub(1)` man page for details.

To submit an interactive job with the **qsh** command, type a command like the following:

```
% qsh -l arch=solaris64
```

This command starts an `xterm` on any available Sun Solaris 64-bit operating system host.

Using `qlogin` to Submit Interactive Jobs

Use the `qlogin` command from any terminal window to start an interactive session under the control of the Grid Engine system.

To submit an interactive job with the **qlogin** command, type a command like the following:

```
% qlogin -l star-cd=1,h_cpu=6:0:0
```

This command locates a low-loaded host. The host has a Star-CD license available. The host also has at least one queue that can provide a minimum of six hours hard CPU time limit.

> **ⓘ Note**
> Depending on the remote login facility that is configured to be used by the Grid Engine system, you might have to provide your user name, your password, or both, at a login prompt.

How to Submit Interactive Jobs With QMON

> **ⓘ Note**
> The only type of interactive jobs that you can submit from QMON are jobs that bring up an `xterm` on a host selected by the Grid Engine system.

1. Click the Job Control button in the QMON Main Control window.
   The Job Control dialog box appears.

2. Select the Submit Jobs button.

3. Verify that the top button on the right side of the dialog box says "Interactive." If not, click the button to change from Batch to Interactive.
   This prepares the Submit Job dialog box to submit interactive jobs. The meaning and the use of the selection options in the dialog box is almost the same as that described for batch jobs in Submitting Batch Jobs. The difference is that several input fields are grayed out because those fields do not apply to interactive jobs. The following figures show the general and advanced variations of the Interactive Submit Job dialog box.

Submit Job

Job Submission

Sun™ GE 6.2

| General | Advanced |

Interactiv
Jobscript

Submit
Edit
Clear
Reload
Save Settings
Load Settings
Done
Help

Parallel Environment

Environment
DISPLAY=sr1-ubrm-40.central.su
Context

Checkpoint Object

Account

Advance Reservation
0
JSV URL

Verify Mode
Skip
Mail
☐ Start of Job
☐ End of Job
☐ Abort of Job
☐ Suspend of Job
Mail To

Hard Queue List

Soft Queue List

Master Queue List

Job Dependencies

Hold Array Dependencies

Deadline

Transparent Remote Execution

- Remote Execution With qrsh
    - Invoking Transparent Remote Execution With qrsh
- Transparent Job Distribution With qtcsh
    - qtcsh Usage
- Parallel Makefile Processing With qmake

- qmake Usage

The Grid Engine system provides a set of closely related facilities that support the transparent remote execution of certain computational tasks. The core tool for this functionality is the qrsh command, which is described in Remote Execution With qrsh. Two high-level facilities, qtcsh and qmake, build on top of qrsh. These two commands enable the Grid Engine system to transparently distribute implicit computational tasks, thereby enhancing the standard UNIX facilities make and csh. qtcsh is described in Transparent Job Distribution With qtcsh. qmake is described in Parallel Makefile Processing With qmake.

Remote Execution With qrsh

qrsh is the major enabling infrastructure for the implementation of the qtcsh and the qmake facilities. qrsh is also used for the tight integration of the Grid Engine system with parallel environments such as MPI or PVM.

You can use qrsh for various purposes, including the following:

- To provide remote execution of interactive applications that use the Grid Engine system. This is comparable to the standard UNIX facility rsh, which is also called remsh on HP-UX systems.
- To offer interactive login session capabilities that use the Grid Engine system. By default, qlogin is similar to the standard UNIX facility rlogin but it can also be configured to use the UNIX telnet facility or any similar remote login facility.
- To allow for the submission of batch jobs that support terminal I/O (standard output, standard error, and standard input) and terminal control.
- To provide a way to submit a standalone program that is not embedded in a shell script.

> **ℹ Note**
> You can also submit scripts with qrsh by using the -b n option. For more information, see the qrsh man page.

- To provide a submission client that remains active while a batch job is pending or running and that goes away only if the job finishes or is cancelled.
- To allow for the Grid Engine system-controlled remote running of job tasks within the framework of the dispersed resources allocated by parallel jobs. See Tight Integration of Parallel Environments and Grid Engine Software.

Invoking Transparent Remote Execution With qrsh

Type the qrsh command, adding options and arguments according to the following syntax:

```
% qrsh    [<options>] <program>|<shell-script> [<arguments>] \
     [> stdout] [>&2 stderr] [< stdin]
```

qrsh understands almost all options of qsub. qrsh provides the following options:

- -now yes|no – -now yes specifies that the job is scheduled immediately. The job is rejected if no appropriate resources are available. -now yes is the default. -now no specifies that the job is queued like a batch job if the job cannot be started at submission time.
- -inherit – qrsh does not go through the scheduling process to start a job-task. Instead, qrsh assumes that the job is embedded in a parallel job that already has allocated suitable resources on the designated remote execution host. This form of qrsh is commonly used in qmake and in a tight parallel environment integration. The default is not to inherit external job resources.
- -binary yes|no – When specified with the n option, enables you to use qrsh to submit script jobs.
- -noshell – With this option, you do not start the command line that is given to qrsh in a user's login shell. Instead, you execute the command without the wrapping shell. Use this option to speed up execution.
- -nostdin – Suppresses the input stream STDIN. With this option set, qrsh passes the -n option to the rsh command. Suppression of the input stream is especially useful if multiple tasks are executed in parallel using qrsh, for example, in a make process. It is undefined which process gets the input.
- -pty yes|no – Available for qrsh and qlogin only, -pty yes starts the job in a pseudo terminal (pty). If no pty is available, the job fails to start. -pty no starts the job without a pseudo terminal. By default, qrsh without a command and qlogin start the job in a pty, qrsh with a command starts the job without a pty.
- -verbose – This option presents output on the scheduling process. -verbose is mainly intended for debugging purposes and is switched off by default.

Transparent Job Distribution With qtcsh

qtcsh is a fully compatible replacement for the widely known and used UNIX C shell derivative `tcsh`. qtcsh is built around `tcsh`. See the information that is provided in `$SGE_ROOT/3rd_party` for details on the involvement of `tcsh`.

qtcsh provides a command shell with the extension of transparently distributing execution of designated applications to suitable and lightly loaded hosts that use the Grid Engine system. The `.qtask` configuration files define the applications to execute remotely and the requirements that apply to the selection of an execution host.

These applications are transparent to the user and are submitted to the Grid Engine system through the `qrsh` facility. `qrsh` provides standard output, error output, and standard input handling as well as terminal control connection to the remotely executing application.

Three noticeable differences between running such an application remotely and running the application on the same host as the shell are:

- The remote host might be more powerful, lower-loaded, and have required hardware and software resources installed.
- A small delay is incurred by the remote startup of the jobs and by their handling through the Grid Engine system.
- Administrators can restrict the use of resources through interactive jobs (qrsh) and thus through `qtcsh`. If not enough suitable resources are available for an application to be started through `qrsh`, or if all suitable systems are overloaded, the implicit `qrsh` submission fails. A corresponding error message is returned, such as `Not enough resources ... try later.`

In addition to the standard use, qtcsh is a suitable platform for third-party code and tool integration. The single-application execution form of qtcsh is `qtcsh -c` *app-name*. The use of this form of qtcsh inside integration environments presents a persistent interface that almost never needs to be changed. All the required application, tool, integration, site, and even user-specific configurations are contained in appropriately defined `.qtask` files. A further advantage is that this interface can be used in shell scripts, in C programs, and even in Java applications.

qtcsh Usage

The invocation of qtcsh is exactly the same as for `tcsh`. qtcsh extends `tcsh` by providing support for the `.qtask` file and by offering a set of specialized shell built-in modes.

The `.qtask` file is defined as follows. Each line in the file has the following format:

```
% [!]<app-name> <qrsh-options>
```

The optional leading exclamation mark (`!`) defines the precedence between conflicting definitions in a global cluster `.qtask` file and the personal `.qtask` file of the qtcsh user. If the exclamation mark is missing in the global cluster file, a conflicting definition in the user file overrides the definition in the global cluster file. If the exclamation mark is in the global cluster file, the corresponding definition cannot be overridden.

app-name specifies the name of the application that is submitted to the Grid Engine system for remote execution. The application name must appear in the command line exactly as the application is defined in the `.qtask` file. If the application name is prefixed with a path name, a local binary is addressed. No remote execution is intended.

qrsh-options specifies the options to the `qrsh` facility to use. These options define resource requirements for the application.

csh aliases are expanded before a comparison with the application names is performed. The applications intended for remote execution can also appear anywhere in a qtcsh command line, in particular before or after standard I/O redirections.

The following examples demonstrate the syntax:

```
# .qtask file
netscape -v DISPLAY=myhost:0
grep -l h=filesurfer
```

Given this `.qtask` file, the following qtcsh command lines:

```
netscape
~/mybin/netscape
cat very_big_file | grep pattern | sort | uniq
```

Result in:

```
qrsh -v DISPLAY=myhost:0 netscape
~/mybin/netscape
cat very_big_file | qrsh -l h=filesurfer grep pattern | sort | uniq
```

`qtcsh` can operate in different modes, influenced by switches that can be set on or off:

- Local or remote execution of commands. Remote is the default.
- Immediate or batch remote execution. Immediate is the default.
- Verbose or nonverbose output. Nonverbose is the default.

The setting of these modes can be changed using option arguments of `qtcsh` at start time or with the shell built-in command `qrshmode` at runtime. See the `qtcsh(1)` man page for more information.

Parallel Makefile Processing With qmake

`qmake` is a replacement for the standard UNIX `make` facility. `qmake` extends `make` by enabling the distribution of independent `make` steps across a cluster of suitable machines. `qmake` is built around the popular GNU-make facility `gmake`. See the information that is provided in `$SGE_ROOT/3rd_party` for details on the involvement of `gmake`.

To ensure that a distributed `make` process can run to completion, `qmake` does the following:

1. Allocates the required resources in a way analogous to a parallel job.
2. Manages this set of resources without further interaction with the scheduling.
3. Distributes `make` steps as resources become available, using the `qrsh` facility with the `-inherit` option.

`qrsh` provides standard output, error output, and standard input handling as well as terminal control connection to the remotely executing `make` step. There are only three noticeable differences exist between executing a `make` procedure locally and using `qmake`:

- The parallelization of the `make` process will speed up significantly, provided that individual `make` steps have a certain duration and that enough independent `make` steps exist to process.
- In the `make` steps to be started up remotely, an implied small overhead exists that is caused by `qrsh` and the remote execution.
- To take advantage of the `make` step distribution of `qmake`, the user must specify as a minimum the degree of parallelization. That is, the user must specify the number of concurrently executable `make` steps. In addition, the user can specify the resource characteristics required by the `make` steps, such as available software licenses, machine architecture, memory, or CPU-time requirements.

The most common use of `make` is the compilation of complex software packages. However, compilation might not be the major application for `qmake`. Program files are often quite small as a matter of good programming practice. Therefore, compilation of a single program file, which is a single `make` step, often takes only a few seconds. Furthermore, compilation usually implies significant file access. Nested include files can cause this problem. File access might not be accelerated if done for multiple `make` steps in parallel because the file server can become a bottleneck. Such a bottleneck effectively serializes all the file access. Therefore, the compilation process sometimes cannot be accelerated in a satisfactory manner.

Other potential applications of `qmake` are more appropriate. An example is the steering of the interdependencies and the workflow of complex analysis tasks through makefiles. Each `make` step in such environments is typically a simulation or data analysis operation with nonnegligible resource and computation time requirements. A considerable acceleration can be achieved in such cases.

qmake Usage

The command-line syntax of `qmake` looks similar to the syntax of `qrsh`:

```
% qmake [-pe <pe-name pe-range>][<options>] \
  -- [<gnu-make-options>][<target>]
```

> **ⓘ Note**
> The `-inherit` option is also supported by `qmake`, as described later in this section.

Pay special attention to the use of the `-pe` option and its relation to the `gmake -j` option. You can use both options to express the amount of parallelism to be achieved. The difference is that `gmake` provides no possibility with `-j` to specify something like a parallel environment to use. Therefore, `qmake` assumes that a default environment for parallel makes is configured that is called `make`. Furthermore, `gmake`'s `-j` allows for no specification of a range, but only for a single number. `qmake` interprets the number that is given with `-j` as a range of `1n`. By contrast, `-pe` permits the detailed specification of all these parameters. Consequently the following command line examples are identical:

```
% qmake -- -j 10
% qmake -pe make 1-10 --
```

The following command lines cannot be expressed using the `-j` option:

```
% qmake -pe make 5-10,16 --
% qmake -pe mpi 1-99999 --
```

Apart from the syntax, `qmake` supports two modes of invocation: interactively from the command line without the `-inherit` option, or within a batch job with the `-inherit` option. These two modes start different sequences of actions:

- Interactive – When `qmake` is invoked on the command line, the `make` process is implicitly submitted to the Grid Engine system with `qrsh`. The process is as follows:
    1. The resource requirements that are specified in the `qmake` command line are taken into account.
    2. The Grid Engine system selects a master machine for the execution of the parallel job that is associated with the parallel `make` job.
    3. The Grid Engine system starts the `make` procedure. The procedure must start there because the `make` process can be architecture-dependent. The required architecture is specified in the `qmake` command line.
    4. The `qmake` process on the master machine delegates execution of individual `make` steps to the other hosts that are allocated for the job. The steps are passed to `qmake` through the parallel environment hosts file.
- Batch – In this case, `qmake` appears inside a batch script with the `-inherit` option. If the `-inherit` option is not present, a new job is spawned, as described in the first case earlier. This results in `qmake` making use of the resources already allocated to the job into which `qmake` is embedded. `qmake` uses `qrsh -inherit` directly to start `make` steps. When calling `qmake` in batch mode, the specification of resource requirements, the `-pe` option and the `-j` option are ignored.

    > **ⓘ Note**
    > Single CPU jobs also must request a parallel environment:

```
qmake -pe make 1 --
```

If no parallel execution is required, call `qmake` with `gmake` command-line syntax without Grid Engine system options and without `--`. This `qmake` command behaves like `gmake`.

See the qmake(1) man page for further details.

How to Submit a Simple Job From the Command Line

Before You Begin

> **ⓘ Note**
> If you installed the Sun Grid Engine software under an unprivileged user account, you must log in as that user to be able to run jobs. See Installation Accounts for details.

Before you run any Grid Engine system command, you must first set your executable search path and other environment conditions properly.

Steps

1. From the command line, type one of the following commands:
   - If you are using `csh` or `tcsh` as your command interpreter, type the following:

     ```
     % source $SGE_ROOT/$SGE_CELL/common/settings.csh
     ```

     $SGE_ROOT specifies the location of the root directory of the Grid Engine system. This directory was specified at the beginning of the installation procedure.

   - If you are using `sh`, `ksh`, or `bash` as your command interpreter, type the following:

     ```
     # . $SGE_ROOT/$SGE_CELL/common/settings.sh
     ```

     > **ⓘ Note**
     > You can add these commands to your `.login`, `.cshrc`, or `.profile` files, whichever is appropriate. By adding these commands, you guarantee proper settings for all interactive session you start later.

2. Submit a simple job script to your cluster by typing the following command:

   ```
   % qsub simple.sh
   ```

   The command assumes that `simple.sh` is the name of the script file, and that the file is located in your current working directory. You can find the following job in the file `$SGE_ROOT/examples/jobs/simple.sh`.

   ```
   #!/bin/sh
   #
   #
   # (c) 2004 Sun Microsystems, Inc. Use is subject to license terms.

   # This is a simple example of a SGE batch script

   # request Bourne shell as shell for job
   #$ -S /bin/sh

   #
   # print date and time
   date
   # Sleep for 20 seconds
   sleep 20
   # print date and time again
   date
   ```

   If the job submits successfully, the `qsub` command responds with a message similar to the following example:

   ```
   your job 1 ('simple.sh') has been submitted
   ```

3. Type the following command to retrieve status information about your job.

   ```
   % qstat
   ```

   You should receive a status report that provides information about all jobs currently known to the Grid Engine system. For each job, the status report lists the following items:

- Job ID, which is the unique number that is included in the submit confirmation
- Name of the job script
- Owner of the job
- State indicator; for example, `r` means running
- Submit or start time
- Name of the queue in which the job runs

If `qstat` produces no output, no jobs are actually known to the system. For example, your job might already have finished.

You can control the output of the finished jobs by checking their `stdout` and `stderr` redirection files. By default, these files are generated in the job owner's home directory on the host that ran the job. The names of the files are composed of the job script file name with a `.o` extension for the `stdout` file and a `.e` extension for the `stderr` file, followed by the unique job ID. The `stdout` and `stderr` files of your job can be found under the names `simple.sh.o1` and `simple.sh.e1` respectively. These names are used if your job was the first ever executed in a newly installed Grid Engine system.

How to Submit a Simple Job With QMON

Before You Begin

> **Note**
> If you installed the Sun Grid Engine software under an unprivileged user account, you must log in as that user to be able to run jobs. See Installation Accounts for details.

A more convenient way to submit and control jobs and of getting an overview of the Grid Engine system is the graphical user interface QMON. Among other facilities, QMON provides a job submission dialog box and a Job Control dialog box for the tasks of submitting and monitoring jobs.

Steps

1. Type the following command to launch QMON:

   ```
   % qmon
   ```

   During startup, a message window appears, and then the QMON Main Control window appears.

2. Click the Job Control button, and then click the Submit Jobs button, as shown below.

   > ✅ **Tip**
   > The button names, such as Job Control, are displayed when you rest the mouse pointer over the buttons.

...and then click here

Click here first...

The Job Control and Submit Job dialog boxes appear.

3. In the Submit Job dialog box, click the icon at the right of the Job Script field.

Click here first to
select the script...

... then click Submit
to submit the job.
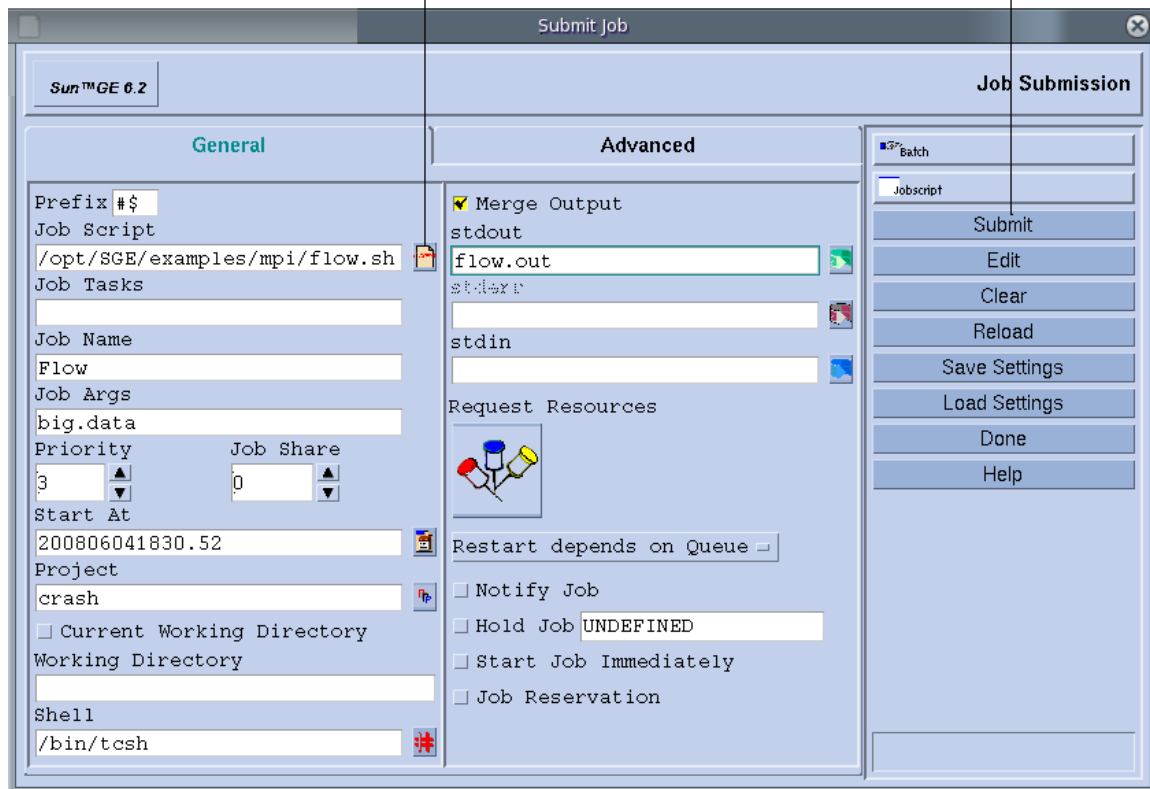
The Select a File dialog box appears.
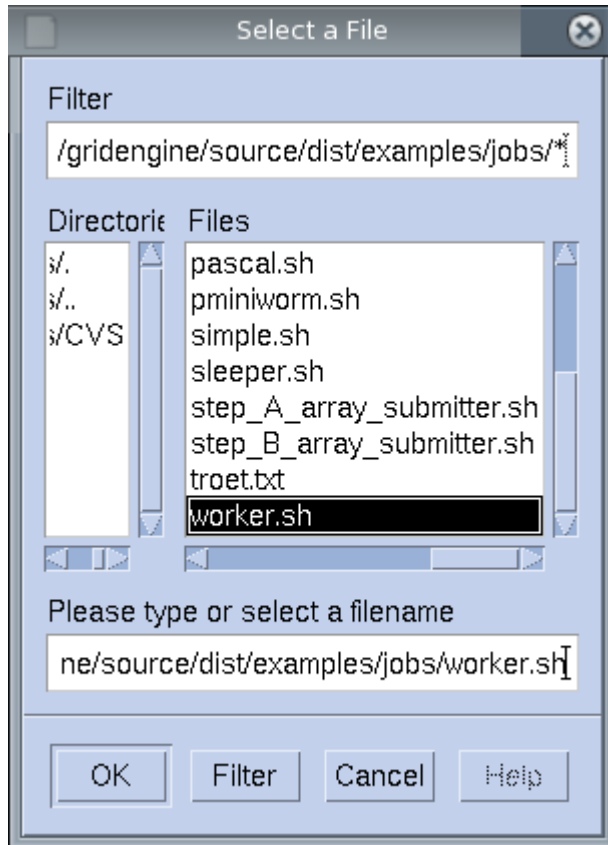
4. Select your script file.
   For example, select the file `simple.sh` that was used in the command line example.

5. Click OK to close the Select a File dialog box.

6. On the Submit Job dialog box, click Submit.
   After a few seconds, you should be able to monitor your job on the Job Control dialog box. First you see your job on the Pending Jobs tab. Once the job starts running, the job quickly moves to the Running Jobs tab.
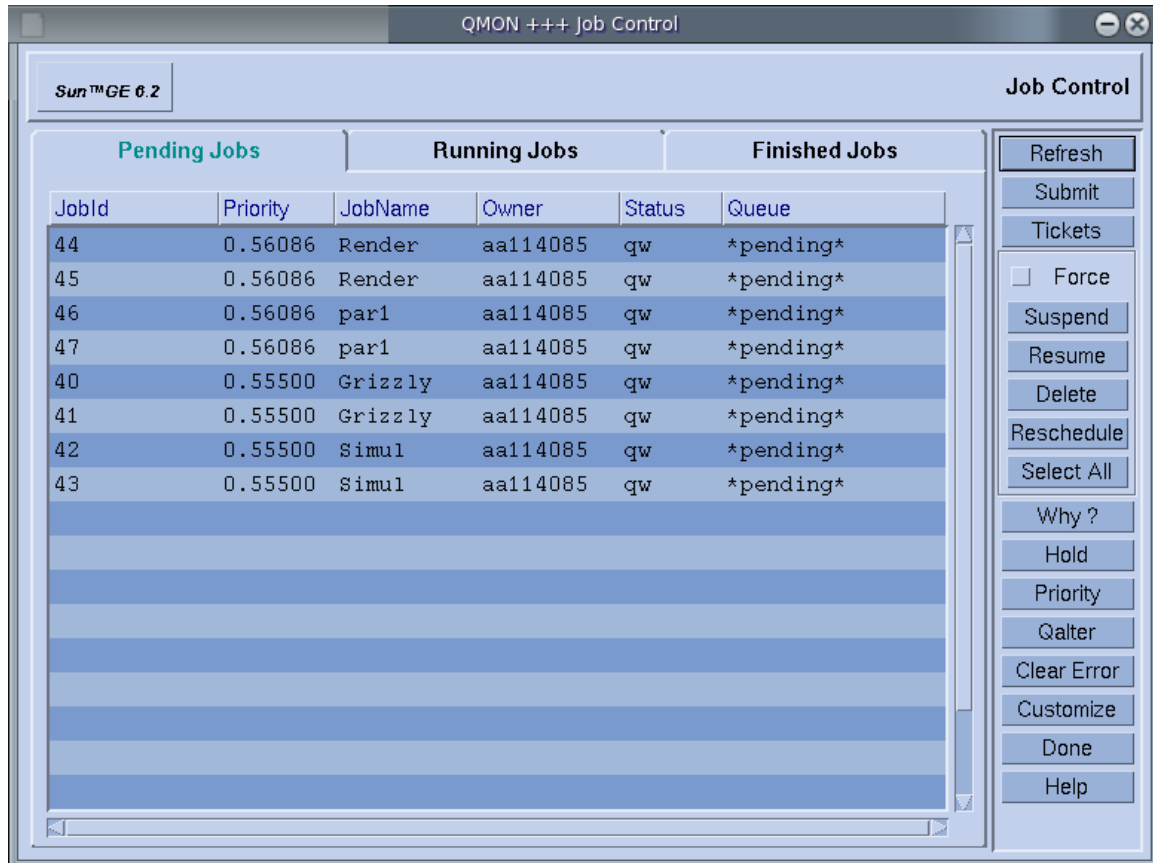
How to Submit an Extended Job From the Command Line

To submit the extended job request that is shown in Figure – Extended Job Submission Example from the command line, type the following command:

```
% qsub -N Flow -p -111 -P devel -a 200404221630.44 -cwd \
    -S /bin/tcsh -o flow.out -j y flow.sh big.data
```

How to Submit an Extended Job With QMON

1. Click the Job Control button in the QMON Main Control window.
   The Job Control dialog box appears.

2. Select a pending job and and click the Submit button.
   The Submit Job dialog box appears. See the example below.
   The General tab of the Submit Job dialog box enables you to configure the following parameters for an extended job:
   - Prefix – A prefix string that is used for script-embedded submit options. See Active Comments for details.
   - Job Script – The job script to use. Click the icon at the right of the Job Script field to open a file selection box.
   - Job Tasks – The task ID range for submitting array jobs. See Submitting Array Jobs for details.
   - Job Name – The name of the job. A default is set after you select a job script.
   - Job Args – Arguments to the job script.
   - Priority – A counting box for setting the job's initial priority This priority ranks a single user's jobs. Priority tells the scheduler how to choose among a single user's jobs when several of that user's jobs are in the system simultaneously.

- Job Share – Defines the share of the job's tickets relative to other jobs. The job share influences only the share tree policy and the functional policy.
- Start At – The time at which the job is considered eligible for execution. Click the icon at the right of the Start At field to open a dialog box.
- Project – The project to which the job is subordinated. Click the icon at the right of the Project field to select among the available projects.
- Current Working Directory – A flag that indicates whether to execute the job in the current working directory. Use this flag only for identical directory hierarchies between the submit host and the potential execution hosts.
- Shell – The command interpreter to use to run the job script. See How a Command Interpreter Is Selected for details. Click the icon at the right of the Shell field to open a dialog box. Enter the command interpreter specifications of the job.
- Merge Output – A flag indicating whether to merge the job's standard output and standard error output together into the standard output stream.
- stdout – The standard output redirection to use. See Output Redirection for details. A default is used if nothing is specified. Click the icon at the right of the stdout field to open a dialog box. Enter the output redirection alternatives.
- stderr – The standard error output redirection to use, similar to the standard output redirection.
- stdin – The standard input file to use, similar to the standard output redirection.
- Request Resources – Click this button to define the resource requirement for your job. If resources are requested for a job, the button changes color.
- Restart depends on Queue – Click this button to define whether the job can be restarted after being aborted by a system crash or similar events. This button also controls whether the restart behavior depends on the queue or is demanded by the job.
- Notify Job – A flag that indicates whether the job is to be notified by SIGUSR1 or by SIGUSR2 signals if the job is about to be suspended or canceled.
- Hold Job – A flag that indicates either a user hold or a job dependency is to be assigned to the job. The job is not eligible for execution as long as any type of hold is assigned to the job. See Monitoring and Controlling Jobs for more details. To restrict a hold, enter a specific range of tasks for an array job in the Hold Job field. For more information, see Submitting Array Jobs.
- Start Job Immediately – A flag that forces the job to be started immediately, if possible, or to be rejected. Jobs are not queued if this flag is selected.
- Job Reservation – A flag that specifies which resources should be reserved for a job. For more information, see Resource Reservation and Backfilling.
The buttons at the right side of the Submit Job dialog box enable you to start various actions:
- Submit – Submit the currently specified job.
- Edit – Edit the selected script file in an X terminal, using either vi or the editor defined by the EDITOR environment variable.
- Clear – Clear all settings in the Submit Job dialog box, including any specified resource requests.
- Reload – Reload the specified script file, parse any script-embedded options, parse default settings, and discard intermediate manual changes to these settings. For more information, see Active Comments and Default Request Files. This action is the equivalent to a Clear action with subsequent specifications of the previous script file. The option has an effect only if a script file is already selected.
- Save Settings – Save the current settings to a file. Use the file selection box to select the file. The saved files can either be loaded later or be used as default requests. For more information, Default Request Files.
- Load Settings – Load settings previously saved with the Save Settings button. The loaded settings overwrite the current settings.
- Done – Closes the Submit Job dialog box.

Example – Extended Job Example

The following figure shows the Submit Job dialog box with most of the parameters set.

Figure – Extended Job Submission Example

The parameters of the job configured in the example are:

- The job has the script file `flow.sh`, which must reside in the working directory of QMON.
- The job is called `Flow`.
- The script file takes the single argument `big.data`.
- The job starts with priority `3`.
- The job is eligible for execution not before 4:30.44 AM of the 22th of April in the year 2004.
- The project definition means that the job is subordinated to project `crash`.
- The job is executed in the submission working directory.
- The job uses the `tcsh` command interpreter.
- Standard output and standard error output are merged into the file `flow.out`, which is created in the current working directory.

How to Submit an Advanced Job From the Command Line

To submit the advanced job request that is shown in Figure – Advanced Job Submission Example from the command line, type the following command:

```
% qsub -N Flow -p -111 -P devel -a 200012240000.00 -cwd \
  -S /bin/tcsh -o flow.out -j y -pe mpi 4-16 \
  -v SHARED_MEM=TRUE,MODEL_SIZE=LARGE \
  -ac JOB_STEP=preprocessing,PORT=1234 \
  -A FLOW -w w -m s,e -q big_q\
  -M me@myhost.com,me@other.address \
  flow.sh big.data
```

Specifying the Use of a Script or a Binary

> **ℹ Note**
>
> Submitting a command as a script can add a number of operations to the submission process and have a negative impact on performance. This impact can be significant if you have short running jobs and big job scripts. If job scripts are available on the execution nodes, i.e. via NFS, binary submission may be a better choice.

You can use the `-b n|y` submit option to indicate explicitly whether the command should be treated as a binary or a script:

- To specify that the command should be treated as a binary or a script, use the `-b y` option with the `qrsh` command.
- To specify the command should be treated only as a script, use the `-b n` option with the `qsub` command.

For more information, see the qsub(1) man page.

Default Request Files

The preceding command shows that advanced job requests can be complex, especially if similar requests need to be submitted frequently. To avoid these problems, you can embed `qsub` options in the script files, or use default request files. For more information, see Active Comments.

The cluster administrator can set up a global default request file for all Grid Engine system users. Users can define a private default request file located in their home directories. In addition, users can create application specific default request files.

If more than one of these files are available, the files are merged into one default request, with the following order of precedence:

1. Application-specific default request file
2. General private default request file
3. Global default request file

Default request files contain the `qsub` options to apply by default to the jobs in one or more lines. The location of the global cluster default request file is $SGE_ROOT/cell/`common/sge_request`. The private general default request file is located under `$HOME/.sge_request`. The application-specific default request files are located under `$cwd/.sge_request`.

Script embedding and the `qsub` command line have higher precedence than the default request files. Therefore, script embedding overrides default request file settings. The `qsub` command line options can override these settings again.

To discard any previous settings, use the `qsub -clear` command in a default request file, in embedded script commands, or in the `qsub` command line.

Example – Private Default Request File

Here is an example of a private default request file:

```
-A myproject -cwd -M me@myhost.com -m b e
-r y -j y -S /bin/ksh
```

Unless overridden, the following is true for all of this user's jobs:

- The account string is `myproject`
- The jobs execute in the current working directory
- Mail notification is sent to `me@myhost.com` at the beginning and at the end of the jobs
- The standard output and standard error output are merged
- The `ksh` is used as command interpreter

How to Submit an Advanced Job With QMON

1. Click the Job Control button in the QMON Main Control window.
   The Job Control dialog box appears.

2. Select a pending job and and click the Qalter button.
   The Submit Job dialog box appears.

3. Click the Advanced Tab, which shown below.
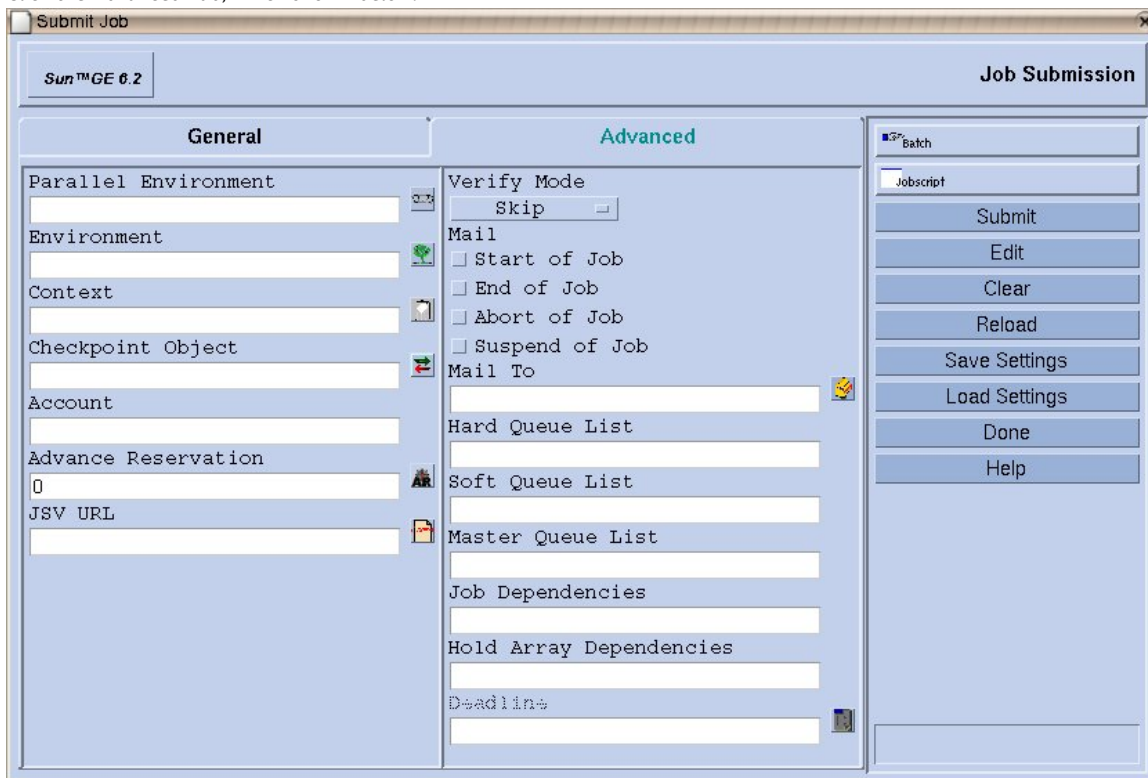
The Advanced tab of the Submit Job dialog box enables you to define the following additional parameters:

- Parallel Environment – A list of available, configured parallel environments.
- Environment – A set of environment variables to set for the job before the job runs. Environment variables can be taken from QMON`s runtime environment, or you can define your own environment variables.
- Context – A list of name/value pairs that can be used to store and communicate job-related information. This information is accessible anywhere from within a cluster. You can modify context variables from the command line with the `-ac`, `-dc`, and `-sc` options to `qsub`, `qrsh`, `qsh`, `qlogin`, and `qalter`.
- Checkpoint Object – The checkpointing environment to use if checkpointing the job is desirable and suitable. See Using Job Checkpointing for details.
- Account – An account string to associate with the job. The account string is added to the accounting record that is kept for the job. The accounting record can be used for later accounting analysis.
- Verify Mode – The Verify flag determines the consistency checking mode for your job. To check for consistency of the job request, the Grid Engine system assumes an empty and unloaded cluster. The system tries to find at least one queue in which the job could run. Possible checking modes are as follows:
  - Skip – No consistency checking at all.
  - Warning – Inconsistencies are reported, but the job is still accepted. Warning mode might be desirable if the cluster configuration should change after the job is submitted.
  - Error – Inconsistencies are reported. The job is rejected if any inconsistencies are encountered.
  - Just verify - The job is not submitted. An extensive report is generated about the suitability of the job for each host and queue in an empty cluster.
  - Poke – The job is not submitted. An extensive report is generated about the suitability of the job for each host and queue in the cluster with all resource utilizations in place.
- Advance Reservation – A list of available, configured advance reservations.
- JSV URL – Access to your directory to select from configured server JSV scripts.
- Mail – The events about which the user is notified by email. The events' start, end, abort, and suspend are currently defined for jobs.
- Mail To – A list of email addresses to which these notifications are sent. Click the icon at the right of the Mail To field to open a dialog box for defining the mailing list.
- Hard Queue List, Soft Queue List – A list of queue names that are requested to be the mandatory selection for the execution of the job. The Hard Queue List and the Soft Queue List are treated identically to a corresponding resource requirement.
- Master Queue List – A list of queue names that are eligible as master queue for a parallel job. A parallel job is started in the master queue. All other queues to which the job spawns parallel tasks are called slave queues.
- Job Dependencies – A list of IDs of jobs that must finish before the submitted job can be started. The newly created job depends on completion of those jobs.
- Hold Array Dependencies – A list of job IDs/and/or job names and sub-tasks. Each sub-task of the submitted job is not eligible for execution unless the corresponding sub-tasks of all jobs referenced in the comma-separated job ID and/or job name list have completed.

- Deadline – The deadline initiation time for deadline jobs. Deadline initiation defines the point in time at which a deadline job must reach maximum priority to finish before a given deadline. To determine the deadline initiation time, subtract an estimate of the running time, at maximum priority, of a deadline job from its desired deadline time. Click the icon at the right of the Deadline field to open the dialog box that enables you to set the deadline.

> 🛈 **Note**
> Not all users are allowed to submit deadline jobs. Ask your system administrator if you are permitted to submit deadline jobs. Contact the cluster administrator for information about the maximum priority that is given to deadline jobs.

Monitoring Hosts

Task	User Interface	Description
How to Monitor Hosts	CLI or QMON	Learn how to monitor hosts.

How to Monitor Hosts From the Command Line

Using `qconf`

To display an execution host configuration, type the following command:

```
% qconf -se <hostname>
```

The `-se` option (show execution host) shows the configuration of the specified execution host as defined in `host_conf`.

To display an execution host list, type one of the following command:

```
% qconf -sel
```

The `-sel` option (show execution host list) displays a list of hosts that are configured as execution hosts.

For more information, see the qconf(1) man page.

Using `qhost`

To monitor execution hosts from the command line, type the following command:

```
% qhost
```

This command produces output that is similar to the following example:

```
HOSTNAME ARCH NCPU LOAD MEMTOT MEMUSE SWAPTO SWAPUS
-----------------------------------------------------------------------------
global - - - - - - -
grid1 sol-sparc64 2 0.27 2.0G 256.0M 8.0G 0.0
gridengine2 sol-amd64 4 0.00 3.9G 421.0M 2.0G 0.0
gridengine5 sol-amd64 4 0.00 3.9G 488.0M 7.9G 0.0
gridengine6 sol-amd64 4 0.07 3.9G 2.6G 4.0G 0.0
```

See the qhost(1) man page for a description of the output format and for more options.

How to Monitor Hosts With QMON

1. Click the Queue Control button in the QMON Main Control window.
   The Cluster Queues dialog box appears.

2. Click the Hosts tab.
   The Hosts tab provides a quick overview of all hosts that are available for the cluster.

Hosts Status

Each row in the hosts table represents one host. For each host, the table lists the following information:

- Host – Name of the host

- Arch – Architecture of the host
- #CPU – Number of processors
- LoadAvg – Load average of the host
- %CPU – LoadAvg / (#CPU * 100)
- MemUsed – Used Memory
- MemTotal – Total Memory
- SwapUsed – Used Swap Memory
- SwapTotal – Total Swap Memory
- VirtUsed – Virtual Used Memory
- VirtTotal – Virtual Total Memory

Monitoring and Controlling Jobs

After you submit jobs, you need to monitor and control them. The following page provides information about monitoring and controlling jobs.

> **Note**
> Only the job owner or Grid Engine managers and operators can suspend and resume jobs, delete jobs, hold back jobs, modify job priority, and modify attributes. See Managers, Operators, and Owners.

Task	User Interface	Description
How to Monitor Jobs	CLI or QMON	Learn how to monitor jobs.
How to Monitor Jobs by Email	CLI or QMON	Learn how to monitor jobs by email.
How to Control Jobs	CLI or QMON	Learn how to control jobs.

How to Monitor Jobs From the Command Line

Use the `qstat` command to perform the following monitoring functions:

- To display a list of jobs with no queue status information, type the following command:

  ```
  qstat
  ```

  The purpose of most of the columns should be self-explanatory. The `state` column, however, contains single character codes with the following meaning: `r` for running, `s` for suspended, `q` for queued, and `w` for waiting.

- To display summary information on all queues and the queued job list, type the following command:

  ```
  qstat -f
  ```

  The display is divided into the following two sections:

  - Available Queues - This section displays the status of all available queues. The first line of the queue section defines the meaning of the columns with respect to the queues that are listed. The queues are separated by horizontal lines. If jobs run in a queue, the job names appear below the associated queue in the same format as in the `qstat` command in its first form. The columns of the queue description provide the following information:
    - `qtype` – Queue type. Queue type is either `B` (batch) or `I` (interactive).
    - `used/free` – Count of used and free job slots in the queue.
    - `states` – State of the queue. See the `qstat(1)` man page for detailed information about queue states.
  - `Pending Jobs` – This section shows the status of the `sge_qmaster` job spool area. The pending jobs in the second output

section are also listed as in `qstat`'s first form.

- To display current job usage and ticket information for a job, type the following command:

```
{{qstat -ext}}
```

This command contains details such as up-to-date job usage and tickets assigned to a job. The following information is displayed:

- The usage and ticket values assigned to a job, shown in the following columns:
    - `cpu/mem/io` – Currently accumulated CPU, memory, and I/O usage.
    - `tckts` – Total number of tickets assigned to the job.
    - `ovrts` – Override tickets assigned through `qalter -ot`.
    - `otckt` – Tickets assigned through the override policy.
    - `ftckt` – Tickets assigned through the functional policy.
    - `stckt` – Tickets assigned through the share-based policy.
    - Share - Current Resource share that each job has with respect to the usage generated by all jobs in the cluster.
- The deadline initiation time in the column deadline, if applicable.

Additional options to the `qstat` command enhance the functionality. Use the `-r` option to display the resource requirements of submitted jobs. Furthermore, the output can be restricted to a certain user or to a specific queue. You can use the `-l` option to specify resource requirements, as described in Defining Resource Requirements, for the `qsub` command. If resource requirements are used, only those queues, and the jobs that are running in those queues, are displayed that match the resource requirement specified by `qstat`.

> ℹ️ **Note**
> The `qstat` command has been enhanced so that the administrator and the user may define files that can contain useful options. See the sge_qstat(5) man page. A cluster-wide sge_qstat file may be placed under $xxQS_NAME_Sxx_ROOT/$xxQS_NAME_Sxx_CELL/common/sge_qstat. The user private file is processed under the location $HOME/.sge_qstat. The home directory request file has the highest precedence, then the cluster global file. You can use the command line to override the flags contained in a file.

See the qstat(1) man page for a detailed explanation of the `qstat` output format.

The following examples show output from the `qstat` and `qstat -f` commands.

Example – qstat -f Output

```
queuename              qtype  used/free  load_avg  arch       states

dq                     BIP    0/1        99.99     sun4       au

durin.q                BIP    2/2        0.36      sun4
   231    0   hydra            craig      r         07/13/96   20:27:15    MASTER
   232    0   compile          penny      r         07/13/96   20:30:40    MASTER

dwain.q                BIP    3/3        0.36      sun4
   230    0   blackhole        don        r         07/13/96   20:26:10    MASTER
   233    0   mac              elaine     r         07/13/96   20:30:40    MASTER
   234    0   golf             shannon    r         07/13/96   20:31:44    MASTER

fq                     BIP    0/3        0.36      sun4

############################################################################

- PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS -

############################################################################

   236    5   word             elaine     qw        07/13/96   20:32:07

   235    0   andrun           penny      qw        07/13/96   20:31:43
```

Example – qstat Output

```
job-ID   prior   name       user      state   submit/start at      queue      function
231      0       hydra      craig     r       07/13/96             durin.q    MASTER
                                              20:27:15
232      0       compile    penny     r       07/13/96             durin.q    MASTER
                                              20:30:40
230      0       blackhole  don       r       07/13/96             dwain.q    MASTER
                                              20:26:10
233      0       mac        elaine    r       07/13/96             dwain.q    MASTER
                                              20:30:40
234      0       golf       shannon   r       07/13/96             dwain.q    MASTER
                                              20:31:44
236      5       word       elaine    qw      07/13/96
                                              20:32:07
235      0       andrun     penny     qw      07/13/96 20:31:43
```

How to Monitor Jobs With QMON

To monitor jobs with QMON, click the Job Control button in the QMON Main Control window.
The Job Control dialog box appears, as shown below.

How to Get Additional Information About Jobs With the QMON Object Browser

You can use the QMON Object Browser to quickly retrieve additional information about jobs without having to customize the Job Control dialog box, as explained in How to Monitor Jobs With QMON.

To display information about jobs using the Object Browser, use one of the following methods:

- From the Job Control dialog box, move the pointer over a job name.
- From the Browser dialog box, click Job.

How to Monitor Jobs by Email

From the command line, type the following command with appropriate arguments.

```
% qsub -m <arguments>
```

The `qsub -m` command requests email to be sent to the user who submitted a job or to the email addresses specified by the `-M` flag if certain events occur. See the `qsub(1)` man page for a description of the flags. An argument to the `-m` option specifies the events. The following arguments are available:

- `b` – Send email at the beginning of the job.
- `e` – Send email at the end of the job.
- `a` – Send email when the job is rescheduled or aborted For example, by using the `qdel` command.
- `s` – Send email when the job is suspended.
- `n` – Do not send email. `n` is the default.

Use a string made up of one or more of the letter arguments to specify several of these options with a single `-m` option. For example, `-m be` sends email at the beginning and at the end of a job.

How to Monitor Jobs by Email With QMON

1. Click the Job Control button in the QMON Main Control window.
   The Job Control dialog box appears.

2. Select a pending job and and click the Qalter button.
   The Submit Job dialog box appears, as shown below.

3. Select the Advanced Tab.

4. Click on the icon left of the Mail To field to select or add email addresses of the user or users who are responsible for monitoring jobs.

ⓘ You can also configure this parameter at the time of job submission using the Submit Job dialog box.

How to Control Jobs From the Command Line

ⓘ In order to delete, suspend, or resume a job, you must be the owner of the job or a Grid Engine manager or operator. For more information, see Users and User Categories.

Use `qdel` and `qmod` in the following ways to control jobs from the command line:

- To delete a job, regardless of whether a job is running or spooled, type the following command:

```
qdel <job-id>
```

- To suspend a job that is already running, type the following command:

```
qmod -sj <job-id>
```

- To restart a suspended job, type the following command:

```
qmod -usj <job-id>
```

To retrieve a job_id number, use `qstat`. For more information, see How to Monitor Jobs From the Command Line.

If an execution daemon is unreachable, you can use the `-f` (force) option with both commands to register a job status change at master daemon. The `-f` option is intended for use only by an administrator. However, In the case of `qdel`, users can force deletion of their own jobs if the flag `ENABLE_FORCED_QDEL` in the cluster configuration `qmaster_params` entry is set. See the sge_conf(5) man page for more information.

For more information, see the qmod(1) man page.

How to Control Jobs With QMON

- How to Modify Job Attributes
- How to Change Job Priority
- How to Put Jobs and Array Job Tasks on Hold
- How to Force Jobs
- How to Verify Job Consistency
- How to Use the Why? Button to Get Information About Pending Jobs
- How to Clear Error States
- How to Filter the Job List
- How to Customize the Job Control Display

1. Click the Job Control button in the QMON Main Control window.
   The Job Control dialog box appears, as shown below.
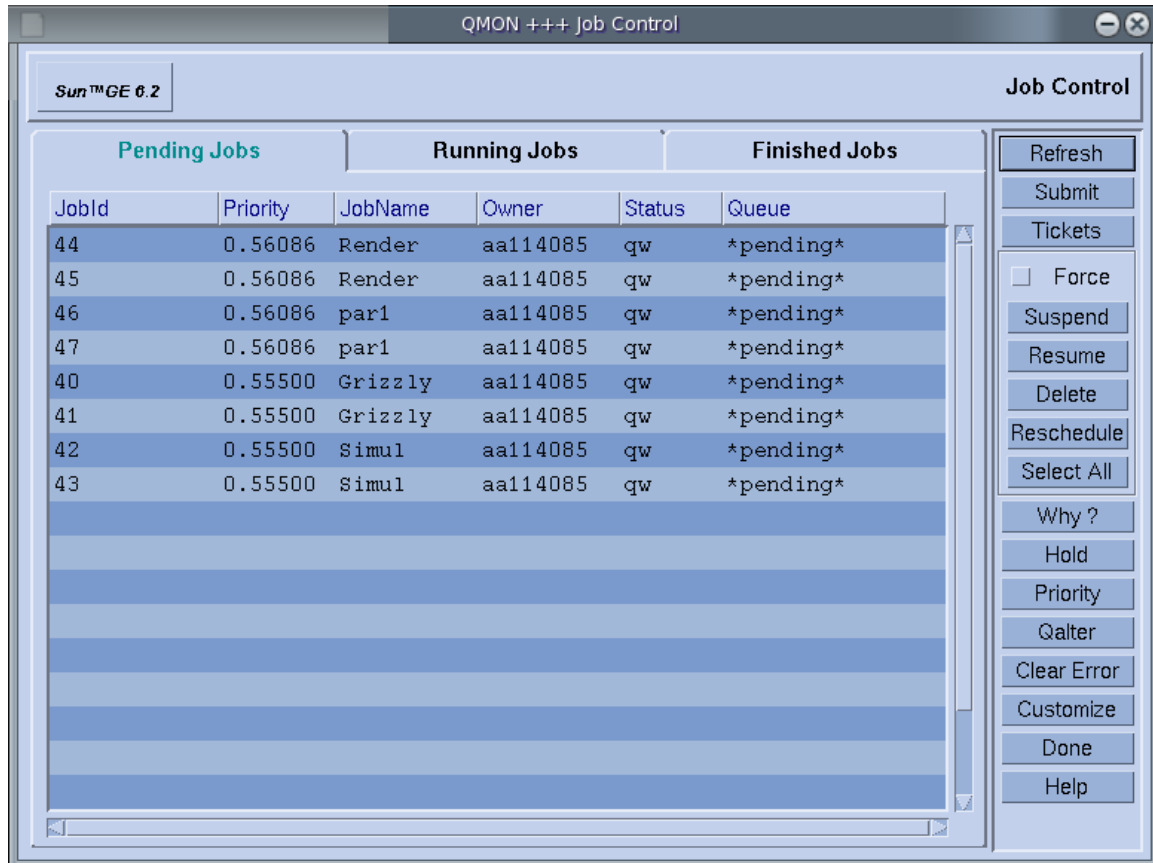
2. You can perform the following tasks from the Job Control dialog box:

> **ℹ Note**
>
> To select jobs, use the following mouse and key combinations:
>
>   a.
>   - To select multiple noncontiguous jobs, hold down the Control key and click two or more jobs.
>   - To select a contiguous range of jobs, hold down the Shift key, click the first job in the range, and then click the last job in the range.
>   - To toggle between selecting a job and clearing the selection, click the job while holding down the Control key.

- To monitor jobs, click the Pending Jobs, Running Jobs, or Finished Jobs tab.
- To refresh the Job Control display, click the Refresh button to force an update. QMON then uses a polling scheme to retrieve the status of the jobs from `sge_qmaster`.
- To modify job attributes, select a pending or running job and then click the Qalter button. For more information, see How to Modify Job Attributes.
- To change job priority, select a pending or running job and then press the Priority button. For more information, see How to Change Job Priority.
- To put a job or an array task on hold, select a pending job and then press the Hold button. For more information, see How to Put Jobs and Array Job Tasks on Hold.
- To force a job, first select a pending job or running job, next select the Force option and then click the Suspend, Resume, or Delete buttons. For more information, see How to Force Jobs.
- To verify job consistency, select a pending job and then click the Qalter button. For more information, see How to Verify Job Consistency.
- To get information about pending jobs using the Why? button, select a pending job and then click the Why? button. For more information, see How to Use the Why? Button to Get Information About Pending Jobs.
- To clear error states, select a pending job and then click the Clear Error button. For more information, see How to Clear Error States.
- To filter the job list, click the Customize button. For more information, see How to Filter the Job List.
- To customize the job control display, click the Customize button. For more information, see How to Customize the Job Control Display.

How to Modify Job Attributes

1. Click a pending or running job on the and Job Control dialog box and then click the Qalter button.
   The Submit Job dialog box appears. All the entries of the dialog box correspond to the attributes of the job that were defined when the job was submitted.

   > ℹ **Note**
   > Entries that cannot be changed are grayed out.

2. Edit available entries appropriately.

3. Click the Qalter button, a substitute for the Submit button on the Submit Job dialog box, to register changes with the Grid Engine system.

How to Change Job Priority

1. Select a pending or running job on the Job Control dialog box and then click the Priority button.
   The priority dialog box appears, as shown below. This dialog box enables you to change the priority of selected pending or running jobs. The priority ranks a single user's jobs among themselves. Priority tells the scheduler how to choose among a single user's jobs when several jobs are in the system simultaneously.
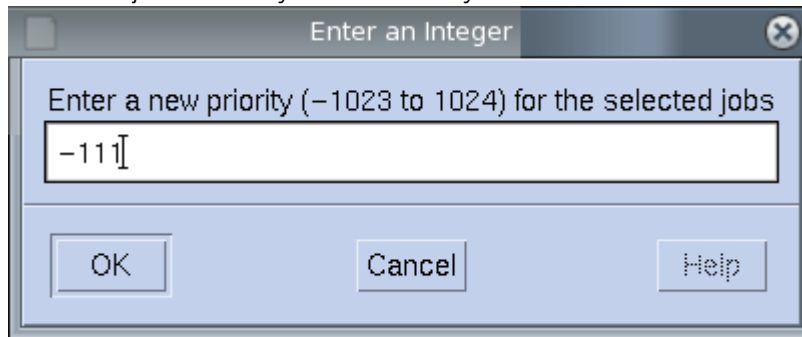
2. Enter a new priority for the selected job(s) in the field and then click OK.

How to Put Jobs and Array Job Tasks on Hold

As long as any hold is assigned to a job or an array job task, the job or array job task is not eligible for running.

> ℹ **Note**
> User holds can be set or reset by the job owner as well as by Grid Engine managers and operators. Operator holds can be set or reset by managers and operators. System holds can be set or reset by managers only. You can also set or reset holds by using the `qalter`, `qhold`, and `qrls` commands.

- To put a job on hold, select a pending job from the Job Control Dialog dialog box, shown above, and click Hold.
  The Set Hold dialog box appears. The Set Hold dialog box enables you to set and reset user, operator, and system holds.
- To put an array task on hold, do the following:
  1. Select a pending job from the Job Control dialog box and click Hold.
     The Set Hold dialog box appears.
  2. Use the Tasks field to put a hold on particular subtasks of an array job.
     The task ID range specified in this field can be a single number, a simple range of the form n-m, or a range with a step size. For example, the task ID range specified by `2-10:2` results in the task ID indexes 2, 4, 6, 8, and 10. This range represents a total of five identical tasks, with the environment variable `SGE_TASK_ID` containing one of the five index numbers. For detailed information about job holds, see the qsub(1) man page.

How to Force Jobs

Only running jobs can be suspended or resumed. Only pending jobs can be rescheduled, held back and modified, in priority as well as in other attributes.

1. To force jobs, select a job from the Pending Jobs tab or the Running Jobs tab and then select the Force option.

2. Click the Suspend, Resume, or Delete buttons.

> **ℹ Note**
> You can force suspending, resuming, and deleting jobs. In other words, you can register these actions with `sge_qmaster` without notifying the `sge_execd` that controls the jobs. Forcing is useful when the corresponding `sge_execd` is unreachable, for example, due to network problems.

Suspension of a job sends the signal `SIGSTOP` to the process group of the job with the UNIX `kill` command. `SIGSTOP` halts the job and no longer consumes CPU time. Resumption of the job sends the signal `SIGCONT`, thereby unsuspending the job. See the `kill(1)` man page for your system for more information on signaling processes.

How to Verify Job Consistency

> **ℹ Note**
> The Verify flag on the Submit Job dialog box has a special meaning when the flag is used in the Qalter mode. You can check pending jobs for consistency, and you can investigate why jobs are not yet scheduled.

1. Select a pending job from the Job Control dialog box and and click the Qalter button.

2. Click the Advanced tab.

3. Select the desired consistency-checking mode for the Verify flag, and then click Qalter.

> **ℹ Note**
> The system displays warnings on inconsistencies, depending on the checking mode you select. See [How to Submit Advanced Jobs With QMON] and the `-w` option on the `qalter(1)` man page for more information.

How to Use the Why? Button to Get Information About Pending Jobs

> **ℹ Note**
> The Why? button delivers meaningful output only if the scheduler configuration parameter `schedd_job_info` is set to `true`. See the `sched_conf(5)` man page.

To get information about pending jobs, select a pending job from the Job Control dialog box and click the Why? button.
The Object Browser dialog box appears. As shown below, this dialog box displays a list of reasons that prevented the scheduler from dispatching the job in its most recent pass.

> **ℹ Note**
> The displayed scheduler information relates to the last scheduling interval. The information might not be accurate by the time you investigate why your job was not scheduled.

```
                                       QMON +++ Browser                          ⊖ ⊕ ⊗

  Sun™GE 6.2                                                          Object Browser

  +++++++++++++++++++++++++++++++++++++++++++                          ┌ Objects ┐
  scheduling info:         queue instance "testQueue_1@gally2" dropped because it is overloaded: no value for comple
                           queue instance "all.q@gally2" dropped because it is full        stdout
                           queue instance "crashQueue@gally2" dropped because it is full
                           queue instance "dilbert@gally2" dropped because it is full      stderr
  I
                                                                                           Queue

                                                                                           Job
```
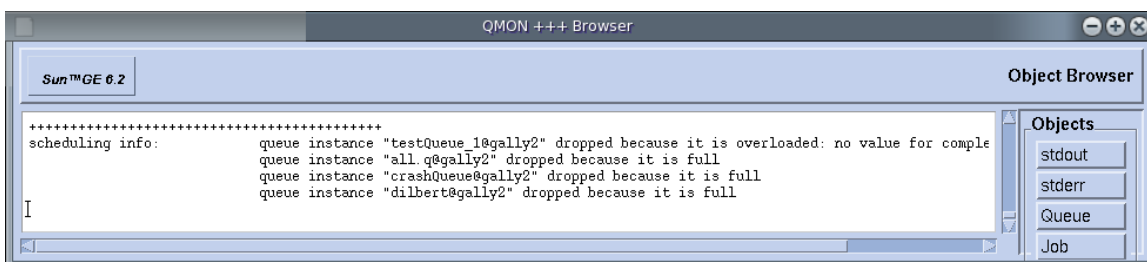
How to Clear Error States

To clear error states, select a pending job from the Job Control dialog box and then click the Clear Error button.
This removes an error state from a pending job that failed due to a job-dependent problem. For example, the job might have insufficient permissions to write to the specified job output file.

Error states appear in red text in the pending jobs list. You should remove jobs only after you correct the error condition, for example, using `qalter`. Such error conditions are automatically reported through email if the job requests to send email when the job is aborted. For example, the job might have been aborted with the `qsub -m a` command.

How to Filter the Job List

1. Click the Customize button in the Job Control dialog box.
   The Job Customize box appears, as shown below.

2. Click the Filter Jobs tab.

   Example – Filtering the Job List

   The following example of the filtering facility selects only jobs that are suitable to be run on the architecture `solaris64`.
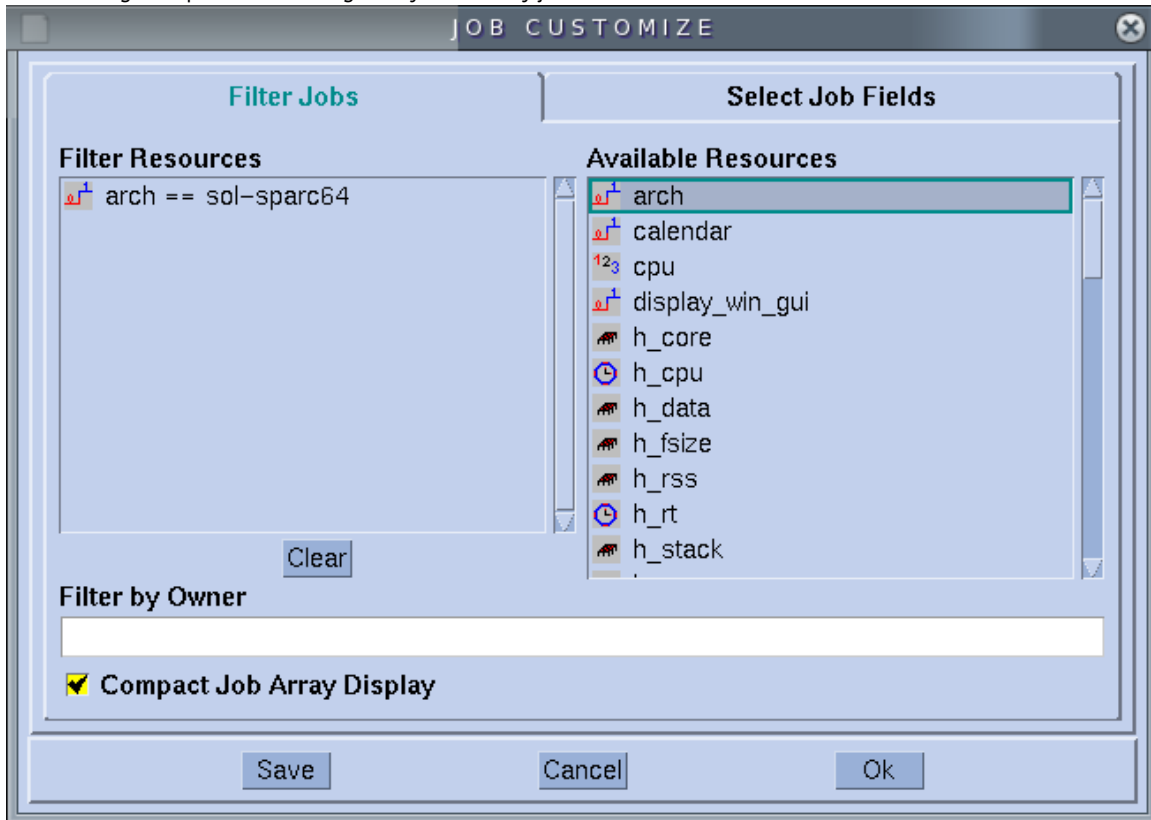
   The following figure shows the resulting Running Jobs tab of the Job Control dialog box.

The Job Control dialog box that is shown in the previous figure is also an example of how QMON displays array jobs.

How to Customize the Job Control Display

1. Click the Customize button on the Job Control dialog box.
    The Job Customize dialog box appears.

2. Click the Select Job Fields tab.
    A sample Select Job Fields tab is shown in the following figure.
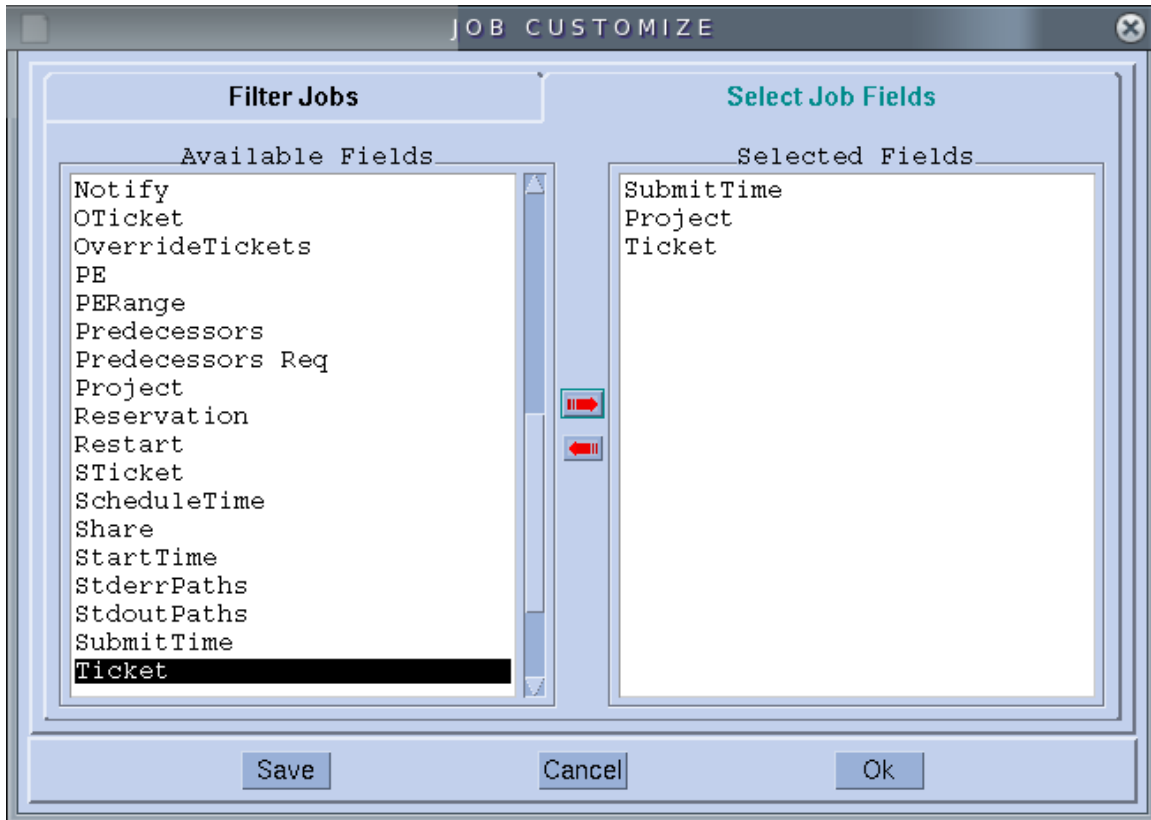
3. Use the Job Customize dialog box to configure the set of information to display.
   You can select more entries of the job object to be displayed.

4. Use the Save button on the Job Customize dialog box to store the customizations in the file `.qmon_preferences`.
   This file is located in the user's home directory. By saving your customizations, you redefine the appearance of the Job Control dialog box.

Monitoring and Controlling Queues

After you configure queues, you need to monitor and control them. This page provides information about monitoring and controlling queues.

Task	User Interface	Description
How to Monitor Queues	CLI or QMON	Learn how to monitor queues.
How Control Queues	CLI or QMON	Learn how to control queues.

How to Monitor Queues With QMON

1. Click the Queue Control button in the QMON Main Control window.
   The Cluster Queues dialog box appears, as shown below.

CLUSTER QUEUE	CQLOAD	USED	RES	AVAIL	TOTAL	aoACDS	cdsuE	s
all.q	0.02	0	0	4	11	0	7	0
crashQueue	0.02	0	0	1	8	0	7	0
dilbert.q	0.02	0	0	1	8	0	7	0
testQueue_1	0.02	0	0	0	8	1	7	0
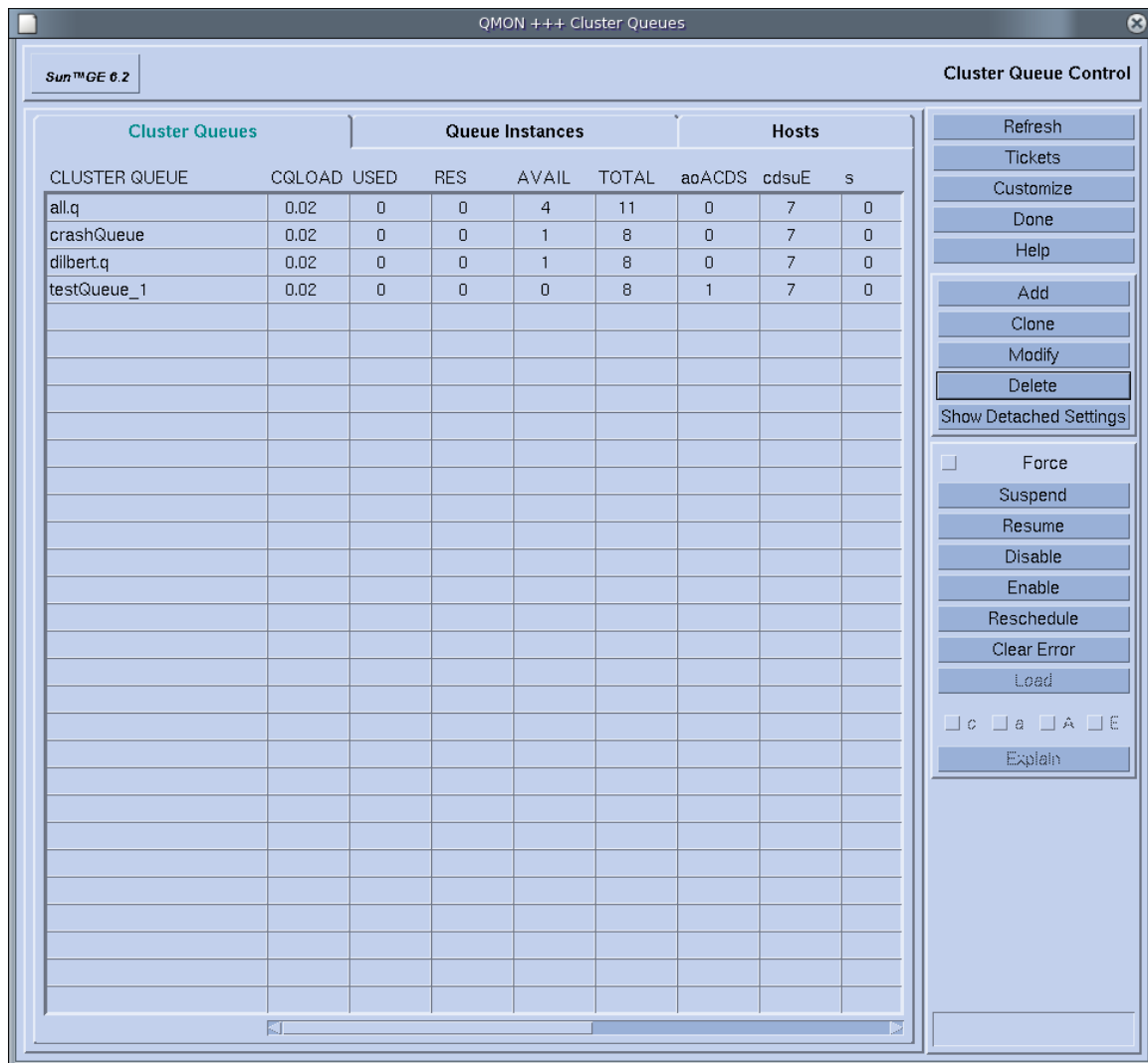
2. Click the Cluster Queues tab.
   The Cluster Queues tab provides a quick overview of all cluster queues that are defined for the cluster.

> **ⓘ Note**
> Information displayed in the Cluster Queues dialog box is updated periodically. Click Refresh to force an update.

How to Control Queues From the Command Line

> **ⓘ** Suspending and resuming queues as well as disabling and enabling queues requires queue owner permission, manager permission, or operator permission. For more information, see Users and User Categories.

You can use qmod to control queues in the following ways:

- To suspend a queue and any active jobs on that queue, type the following command:

```
qmod -sq <q-name>,...
```

- To unsuspend a queue and any active jobs on that queue, type the following command:

```
qmod -usq <q-name>,...
```

- To disable a queue and stop any jobs from being dispatched to the queue, type the following command:

```
qmod -d <q-name>,...
```

- To enable a queue, type the following command:

```
qmod -e <q-name>,...
```

The `-f` option forces registration of the status change in `sge_qmaster` when the corresponding `sge_execd` is not reachable, for example, due to network problems.

Automating Grid Engine Functions Through DRMAA

You can automate Sun Grid Engine functions by writing scripts that run Sun Grid Engine commands and parse the results. However, for more consistent and efficient results, you can use the C or Java language and the Distributed Resource Management Application API. This section introduces the DRMAA concept and explains how to use it with the C and Java languages.

The Distributed Resource Management Application API (DRMAA, which is pronounced like "drama") is an Open Grid Forum specification to standardize job submission, monitoring, and control in Distributed Resource Management Systems (DRMS). The objective of the DRMAA Working Group was to produce an API that would be easy to learn, easy to implement, and that would enable useful application integrations with DRMS in a standard way.

The DRMAA specification is language, platform, and DRMS agnostic. A wide variety of systems should be able to implement the DRMAA specification. To provide additional guidance for DRMAA implementation in specific languages, the DRMAA Working Group also produced several DRMAA language binding specifications. These specifications define what a DRMAA implementation should resemble in a given language.

The DRMAA specification is currently at version 1.0. The DRMAA Java Language Binding Specification is also at version 1.0, as is the DRMAA C Language Binding Specification. Sun Grid Engine provides implementations of both the 1.0 Java language binding and the 1.0 C language binding.

For more information about the DRMAA 1.0 specification, see the language specific binding specifications on the Open Grid Forum DRMAA Working Group Web Site

Topic	Description
Developing With the C Language Binding	Learn how to develop with the C language binding.
Developing With the Java Language Binding	Learn how to develop with the java language binding.

Developing With the C Language Binding

Important Files for the C Language Binding

To use the DRMAA C language binding implementation included with Sun Grid Engine, you need to know where to find the important files. The most important file is the DRMAA header file that you included from your C application to make the DRMAA functions available to your application. The DRMAA header file resides in the `$SGE_ROOT/include/drmaa.h`, where `$SGE_ROOT` defaults to `/usr/SGE`. For detailed

reference information about the DRMAA functions, see section 3 of the Sun Grid Engine man pages, located in the `$SGE_ROOT/man` directory. To compile and link your application, use the DRMAA shared library at `$SGE_ROOT/lib/$SGE_ARCH/libdrmaa.so`.

Including the DRMAA Header File

To use the DRMAA functions in your application, every source file that uses a DRMAA function must include the DRMAA header file. To include the DRMAA header file in your source file, add the following line to your source code:

```
#include "drmaa.h"
```

Compiling Your C Application

When you compile your DRMAA application, you need to include some additional compiler directives to direct the compiler and linker to use DRMAA. The following directions apply to the Sun Studio Compiler Collection and to gcc. These instructions might not apply for other compilers and linkers. Consult the documentation for your specific compiler and linker products.

You must include the following two directives:

- Tell the compiler to include the DRMAA header file by adding the following statement to the compiler command line:

```
-$SGE_ROOT/include
```

- Tell the linker to include the DRMAA library by adding the following statement to the compiler and/or linker command line:

```
-ldrmaa
```

You also need to verify that the `$SGE_ROOT/lib/$SGE_ARCH` directory is included in your library search path. The path is `LD_LIBRARY_PATH` on the Solaris Operating Environment and Linux. The `$SGE_ROOT/lib/$SGE_ARCH` directory is not included automatically when you set your environment using the `settings.sh` or `settings.csh` files.

Example - Compiling Your C Application Using Sun Studio Compiler

The following example shows how you would compile your DRMAA application using the Sun Studio Compiler. The following assumptions apply:

- You are using the `csh` shell on a Solaris host.
- Sun Grid Engine is installed in `/sge`.
- The DRMAA application is stored in app.c.

Sample commands would look like the following:

```
% source /sge/default/common/settings.csh
% cc -I/sge/include -ldrmaa app.c
```

Running Your C Application

To run your compiled DRMAA application, verify the following:

The `$SGE_ROOT/lib/$SGE_ARCH` directory must be included in the library search path (`LD_LIBRARY_PATH` on the Solaris Operating Environment and Linux). The `$SGE_ROOT/lib/$SGE_ARCH` directory is not included automatically when you set your environment using the `settings.sh` or `settings.csh` files.

You must be logged into a machine that is a Sun Grid Engine submit host. If the machine is not a Sun Grid Engine submit host, all DRMAA function calls will fail, returning `DRMAA_ERRNO_DRM_COMMUNICATION_FAILURE`.

C Application Examples

The following examples illustrate some application interactions that use the C language bindings. You can find additional examples on the "How To" section of the Grid Engine Community Site.

Example – Starting and Stopping a Session

Every call to a DRMAA function returns an error code. If everything goes well, that code is DRMAA_ERRNO_SUCCESS. If an error occurs, an appropriate error code is returned.

Every DRMAA function also takes at least two parameters:

- A string to populate with an error message in case of an error
- An integer representing the maximum length of the error string

On line 8, the example calls drmaa_init(). This function sets up the DRMAA session and must be called before most other DRMAA functions. Some functions, like drmaa_get_contact(), can be called before drmaa_init(), but these functions only provide general information. Any function that performs an action, such as drmaa_run_job() or drmaa_wait() must be called after drmaa_init() returns. If such a function is called before drmaa_init() returns, it will return the error code DRMAA_ERRNO_NO_ACTIVE_SESSION.

The dmraa_init() function creates a session and starts an event client listener thread. The session is used for organizing jobs submitted through DRMAA, and the thread is used to receive updates from the queue master about the state of jobs and the system in general. Once drmaa_init() has been called successfully, the calling application must also call drmaa_exit() before terminating. If an application does not call drmaa_exit() before terminating, the queue master might be left with a dead event client handle, which can decrease queue master performance.

At the end of the program, on line 17, drmaa_exit() cleans up the session and stops the event client listener thread. Most other DRMAA functions must be called before drmaa_exit(). Some functions, like drmaa_get_contact(), can be called after drmaa_exit(), but these functions only provide general information. Any function that performs an action, such as drmaa_run_job() or drmaa_wait() must be called before drmaa_exit() is called. If such a function is called after drmaa_exit() is called, it will return the error code DRMAA_ERRNO_NO_ACTIVE_SESSION.

```
01: #include
02: #include "drmaa.h"
03:
04: int main(int argc, char **argv) {
05:     char error[DRMAA_ERROR_STRING_BUFFER];
06:     int errnum = 0;
07:
08:     errnum = drmaa_init(NULL, error, DRMAA_ERROR_STRING_BUFFER);
09:
10:     if (errnum != DRMAA_ERRNO_SUCCESS) {
11:         fprintf(stderr, "Could not initialize the DRMAA library: %s\n", error);
12:         return 1;
13:     }
14:
15:     printf("DRMAA library was started successfully\n");
16:
17:     errnum = drmaa_exit(error, DRMAA_ERROR_STRING_BUFFER);
18:
19:     if (errnum != DRMAA_ERRNO_SUCCESS) {
20:         fprintf(stderr, "Could not shut down the DRMAA library: %s\n", error);
21:         return 1;
22:     }
23:
24:     return 0;
25: }
```

Example – Running a Job

The following code segment shows how to use the DRMAA C binding to submit a job to Sun Grid Engine. The beginning and end of this program are the same as in the preceding example. The differences are on lines 16 through 59. On line 16, DRMAA allocates a job template. A job template is a structure used to store information about a job to be submitted. The same template can be reused for multiple calls to drmaa_run_job() or drmaa_run_bulk_job().

On line 22, the DRMAA_REMOTE_COMMAND attribute is set. This attribute tells DRMAA where to find the program to run. Its value is the path to the executable. The path can be relative or absolute. If relative, the path is relative to the DRMAA_WD attribute, which defaults to the user's home directory. For this program to work, the script sleeper.sh must be in your default path.

On line 32, the `DRMAA_V_ARGV` attribute is set. This attribute tells DRMAA what arguments to pass to the executable. For more information on DRMAA attributes, see the `drmaa_attributes(3)` man page.

On line 43 , `drmaa_run_job()` submits the job. DRMAA places the id assigned to the job into the character array that is passed to `drmaa_run_job()`. The job is now running as though submitted by `qsub`. At this point, calling `drmaa_exit()` or terminating the program will have no effect on the job.

To clean things up, the job template is deleted on line 54. This frees the memory DRMAA set aside for the job template, but has no effect on submitted jobs.

Finally, on line 61, `drmaa_exit()` is called. The `drmaa_exit()` call is outside of the if structure started on line 18 because when `drmaa_init()` is called, `drmaa_exit()` must be called before terminating, regardless of successive commands.

```
01: #include
02: #include "drmaa.h"
03:
04: int main(int argc, char **argv) {
05:     char error[DRMAA_ERROR_STRING_BUFFER];
06:     int errnum = 0;
07:     drmaa_job_template_t *jt = NULL;
08:
09:     errnum = drmaa_init(NULL, error, DRMAA_ERROR_STRING_BUFFER);
10:
11:     if (errnum != DRMAA_ERRNO_SUCCESS) {
12:         fprintf(stderr, "Could not initialize the DRMAA library: %s\n", error);
13:         return 1;
14:     }
15:
16:     errnum = drmaa_allocate_job_template(&jt, error, DRMAA_ERROR_STRING_BUFFER);
17:
18:     if (errnum != DRMAA_ERRNO_SUCCESS) {
19:         fprintf(stderr, "Could not create job template: %s\n", error);
20:     }
21:     else {
22:         errnum = drmaa_set_attribute(jt, DRMAA_REMOTE_COMMAND, "sleeper.sh",
23:                                      error, DRMAA_ERROR_STRING_BUFFER);
24:
25:         if (errnum != DRMAA_ERRNO_SUCCESS) {
26:             fprintf(stderr, "Could not set attribute \"%s\": %s\n",
27:                     DRMAA_REMOTE_COMMAND, error);
28:         }
29:         else {
30:             const char *args[2] = {"5", NULL};
31:
32:             errnum = drmaa_set_vector_attribute(jt, DRMAA_V_ARGV, args, error,
33:                                                 DRMAA_ERROR_STRING_BUFFER);
34:         }
35:
36:         if (errnum != DRMAA_ERRNO_SUCCESS) {
37:             fprintf(stderr, "Could not set attribute \"%s\": %s\n",
38:                     DRMAA_REMOTE_COMMAND, error);
39:         }
40:         else {
41:             char jobid[DRMAA_JOBNAME_BUFFER];
42:
43:             errnum = drmaa_run_job(jobid, DRMAA_JOBNAME_BUFFER, jt, error,
44:                                    DRMAA_ERROR_STRING_BUFFER);
45:
46:             if (errnum != DRMAA_ERRNO_SUCCESS) {
47:                 fprintf(stderr, "Could not submit job: %s\n", error);
48:             }
49:             else {
50:                 printf("Your job has been submitted with id %s\n", jobid);
51:             }
52:         } /* else */
53:
54:         errnum = drmaa_delete_job_template(jt, error, DRMAA_ERROR_STRING_BUFFER);
55:
56:         if (errnum != DRMAA_ERRNO_SUCCESS) {
57:             fprintf(stderr, "Could not delete job template: %s\n", error);
58:         }
59:     } /* else */
60:
61:     errnum = drmaa_exit(error, DRMAA_ERROR_STRING_BUFFER);
62:
63:     if (errnum != DRMAA_ERRNO_SUCCESS) {
64:         fprintf(stderr, "Could not shut down the DRMAA library: %s\n", error);
65:         return 1;
66:     }
67:
68:     return 0;
69: }
```

Developing With the Java Language Binding

Important Files for the Java Language Binding

To use the DRMAA Java language binding implementation included with Sun Grid Engine, you need to know where to find the important files. The most important file is the DRMAA JAR file `$SGE_ROOT/lib/drmaa.jar`. To compile your DRMAA application, you must include the DRMAA JAR file in your CLASSPATH. The DRMAA classes are documented in the DRMAA Javadoc, located in the `$SGE_ROOT/doc/javadocs` directory. To access the Javadocs, open the file `$SGE_ROOT/doc/javadocs/index.html` in your browser. When you are ready to run your application, you also need the DRMAA shared library, `$SGE_ROOT/lib/$SGE_ARCH/libdrmaa.so`, which provides the required native routines.

Importing the DRMAA Java Classes and Packages

To use the DRMAA classes in your application, your classes should import the DRMAA classes or packages. In most cases, only the classes in the `org.ggf.drmaa` package will be used. You can import these packages individually or using a wildcard package import. In some rare cases, you might need to reference the Sun Grid Engine DRMAA implementation classes found in the `com.sun.grid.drmaa` package. In those cases, you can import the classes individually or you can import all the classes in a given package. The names of the `com.sun.grid.drmaa` classes do not overlap with the `org.ggf.drmaa` classes, so you can import both packages without creating a namespace collision.

Compiling Your Java Application

To compile your DRMAA application, you must include the `$SGE_ROOT/lib/drmaa.jar` file in your CLASSPATH. The `drmaa.jar` file will not be included automatically when you set your environment using the `settings.sh` or `settings.csh` files.

How to Use DRMAA With NetBeans 5.x

To use the DRMAA classes with your NetBeans 5.0 or 5.5 project, follow these steps:

1. Click mouse button 3 on the project node and select **Properties**.

2. Determine whether your project generates a build file or uses an existing file.
    - If your project uses a generated build file:
        a. Select `Libraries` in the left column.
        b. Click `Add Library`.
        c. Click `Manage Libraries` in the Libraries dialog box.
        d. Click `New Library` in the Library Management dialog box.
        e. Type **DRMAA** in the Library Name field in the New Library dialog box.
        f. Click `OK` to dismiss the New Library dialog box.
        g. Click `Add JAR/Folder`.
        h. Browse to the `$SGE_ROOT/lib` directory in the file chooser dialog box and select the `drmaa.jar` file.
        i. Click `Add JAR/Folder` to dismiss the file chooser dialog box.
        j. Click `OK` to dismiss the Library Management dialog box.
        k. Select the DRMAA library and click `Add Library` to dismiss the Libraries dialog box.
    - If your project uses an existing build file:
        a. Select `Java Sources Classpath` in the left column.
        b. Click `Add JAR/Folder`.
        c. Browse to the `$SGE_ROOT/lib` directory in the file chooser dialog box and select the `drmaa.jar` file.
        d. Click `Choose` to dismiss the file chooser dialog box.
3. Click **OK** to dismiss the properties dialog box.

4. Verify that the DRMAA shared library is in the library search path.
   To run your application from NetBeans, the DRMAA shared library file `$SGE_ROOT/lib/$SGE_ARCH/libdrmaa.so` must be included in the library search path (`LD_LIBRARY_PATH` on the Solaris Operating Environment and Linux). The `$SGE_ROOT/lib/$SGE_ARCH` directory is not included automatically when you set your environment using the `settings.sh` or `settings.csh` files. To set up the path for the shared library, perform one of the following:
    - Set up your environment in the shell before launching NetBeans.
    - Add to the netbeans-root`/etc/netbeans.conf` file to set up the environment, such as:

```
# Setup environment for SGE
. $SGE_ROOT/$SGE_CELL/common/settings.sh
SGE_ARCH=`$SGE_ROOT/util/arch`
LD_LIBRARY_PATH=$SGE_ROOT/lib/$SGE_ARCH; export LD_LIBRARY_PATH
```

Running Your Java Application

To run your compiled DRMAA application, verify the following:

- The `$SGE_ROOT/lib/$SGE_ARCH` directory must be included in the library search path (`LD_LIBRARY_PATH` on the Solaris Operating Environment and Linux). The `$SGE_ROOT/lib/$SGE_ARCH` directory is not included automatically when you set your environment using the `settings.sh` or `settings.csh` files.
- You must be logged into a machine that is a Sun Grid Engine submit host. If the machine is not a Sun Grid Engine submit host, all DRMAA method calls will fail, throwing a `DrmCommunicationException`.

Java Application Examples

The following examples illustrate some application interactions that use the Java language bindings. You can find additional examples on the "How To" section of the Grid Engine Community Site.

Example – Starting and Stopping a Session

The following code segment shows the most basic DRMAA Java language binding program.

You must have a `Session` object to do anything with DRMAA. You get the `Session` object from a `SessionFactory`. You get the `SessionFactory` from the static `SessionFactory.getFactory()` method. The reason for this chain is that the org.ggf.drmaa.* classes should be considered an immutable package to be used by every DRMAA Java language binding implementation. Because the package is immutable, to load a specific implementation, the `SessionFactory` uses a system property to find the implementation's session factory, which it then loads. That session factory is then responsible for creating the session in whatever way it sees fit. It should be noted that even though there is a session factory, only one session may exist at a time.

On line 9, `SessionFactory.getFactory()` gets a session factory instance. On line 10, `SessionFactory.getSession()` gets the session instance. On line 13, `Session.init()` initializes the session. `""` is passed in as the contact string to create a new session because no initialization arguments are needed.

`Session.init()` creates a session and starts an event client listener thread. The session is used for organizing jobs submitted through DRMAA, and the thread is used to receive updates from the queue master about the state of jobs and the system in general. Once `Session.init()` has been called successfully, the calling application must also call `Session.exit()` before terminating. If an application does not call `Session.exit()` before terminating, the queue master might be left with a dead event client handle, which can decrease queue master performance. Use the `Runtime.addShutdownHook()` method to make sure `Session.exit()` gets called.

At the end of the program, on line 14, `Session.exit()` cleans up the session and stops the event client listener thread. Most other DRMAA methods must be called before `Session.exit()`. Some functions, like `Session.getContact()`, can be called after `Session.exit()`, but these functions only provide general information. Any function that performs an action, such as `Session.runJob()` or `Session.wait()` must be called before `Session.exit()` is called. If such a function is called after `Session.exit()` is called, it will throw a `NoActiveSessionException`.

```
01: package com.sun.grid.drmaa.howto;
02:
03: import org.ggf.drmaa.DrmaaException;
04: import org.ggf.drmaa.Session;
05: import org.ggf.drmaa.SessionFactory;
06:
07: public class Howto1 {
08:     public static void main(String[] args) {
09:         SessionFactory factory = SessionFactory.getFactory();
10:         Session session = factory.getSession();
11:
12:         try {
13:             session.init("");
14:             session.exit();
15:         } catch (DrmaaException e) {
16:             System.out.println("Error: " + e.getMessage());
17:         }
18:     }
19: }
```

Example – Running a Job

The following code segment shows how to use the DRMAA Java language binding to submit a job to Sun Grid Engine. The beginning and end of this program are the same as in the preceding example. The differences are on lines 16 through 24.

On line 16 , DRMAA allocates a `JobTemplate`. A `JobTemplate` is an object that is used to store information about a job to be submitted. The same template can be reused for multiple calls to `Session.runJob()` or `Session.runBulkJobs()`.

On line 17, the `RemoteCommand` attribute is set. This attribute tells DRMAA where to find the program to run. Its value is the path to the executable. The path can be relative or absolute. If relative, the path is relative to the `WorkingDirectory` attribute, which defaults to the user's home directory. For more information on DRMAA attributes, see the DRMAA Javadoc or the drmaa_attributes(3) man page. For this program to work, the script `sleeper.sh` must be in your default path.

On line 18, the `args` attribute is set. This attribute tells DRMAA what arguments to pass to the executable. For more information on DRMAA attributes, see the DRMAA Javadoc or the drmaa_attributes(3) man page.

On line 20, `Session.runJob()` submits the job. This method returns the ID assigned to the job by the queue master. The job is now running as though submitted by `qsub`. At this point, calling `Session.exit()` or terminating the program will have no effect on the job.

To clean things up, the job template is deleted on line 24. This action frees the memory DRMAA set aside for the job template, but has no effect on submitted jobs.

```java
01: package com.sun.grid.drmaa.howto;
02:
03: import java.util.Collections;
04: import org.ggf.drmaa.DrmaaException;
05: import org.ggf.drmaa.JobTemplate;
06: import org.ggf.drmaa.Session;
07: import org.ggf.drmaa.SessionFactory;
08:
09: public class Howto2 {
10:     public static void main(String[] args) {
11:         SessionFactory factory = SessionFactory.getFactory();
12:         Session session = factory.getSession();
13:
14:         try {
15:             session.init("");
16:             JobTemplate jt = session.createJobTemplate();
17:             jt.setRemoteCommand("sleeper.sh");
18:             jt.setArgs(Collections.singletonList("5"));
19:
20:             String id = session.runJob(jt);
21:
22:             System.out.println("Your job has been submitted with id " + id);
23:
24:             session.deleteJobTemplate(jt);
25:             session.exit();
26:         } catch (DrmaaException e) {
27:             System.out.println("Error: " + e.getMessage());
28:         }
29:     }
30: }
```