April 1991
Order Number: 312061-002

iPSC®/2 AND iPSC®/860 NETWORK QUEUEING SYSTEM MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Beta release version	02/91
-002	Standard release version	04/91

RESTRICTED RIGHTS

PREFACE

This manual describes the Network Queueing System (NQS) implemented for the Intel iPSC®/2 and iPSC®/860 supercomputers.

NOTE

In this manual, the term "iPSC system(s)" refers to any or all of the following SSD products: iPSC®/2, iPSC®/2S, iPSC®/860, iPSC®/860S, and iPSC®/860Plus.

This manual assumes that you are an application programmer proficient in the C and Fortran languages and the UNIX operating system. The manual provides you with enough detail to use the NQS software to queue jobs to your iPSC system.

ORGANIZATION

Chapter 1	Provides an overview of the NQS software, and describes the important concepts implemented by NQS. There are sections describing the supported queue types, request states, types of user operations, networking topics, and using NQS with the iPSC system.
Chapter 2	Provides a procedural description of NQS operations targeted at NQS *operators*. There are sections describing workstation setup, status display, submitting a batch request, submitting a device request, using pipe/network queues, and using print queues.
Chapter 3	Provides a procedural description of NQS operations targeted at NQS *managers*. There are sections describing manager-level use of the **qmgr** command, and discussions of queue access controls, transaction logging, account mapping, and status reporting.

Chapter 4	This is the command reference for all NQS commands.
Glossary	Brief definitions for terms that are unique to NQS and the iPSC system.

NOTATIONAL CONVENTIONS

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Some terms appear in Italic type the first time they are introduced in text. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

Bold-Italic-Monospace
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>` **`<s>`** **`<Ctrl-Alt-Del>`**

[ ] (Brackets) Surround optional items.

. . . (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

APPLICABLE DOCUMENTS

For more information, refer to the following manuals:

iPSC® System Manuals

iPSC®/2 and iPSC®/860 Hardware Installation Manual
Describes installation and powering up of all iPSC system configurations.

iPSC®/2 and iPSC®/860 Site Preparation Guide
Tells the customer how to prepare a site for the installation of an iPSC system.

iPSC®/2 and iPSC®/860 System Administrator's Guide
Describes the system administration tasks related to operating and maintaining an iPSC system.

iPSC®/2 and iPSC®/860 User's Guide
Overviews the iPSC system, including hardware and software architectures. Tells how to develop and run programs.

Intel® Manuals

UNIX System V Manual Set
Describes UNIX System V.

SYP301 Installation and User's Guide
Tells how to install and start the System Resource Manager. Also provides hardware technical data.

Other Manuals

The X Window System Manual Set
Describes the X Windows System application programming.

TABLE OF CONTENTS

CHAPTER 1
OVERVIEW OF NQS

CHAPTER 2
GETTING STARTED WITH NQS

CHAPTER 3
MANAGING NQS OPERATIONS

CHAPTER 4
NQS COMMAND REFERENCE

APPENDIX A
RECONFIGURING NQS

LIST OF ILLUSTRATIONS

LIST OF TABLES

OVERVIEW OF NQS 1

INTRODUCTION

The Network Queueing System (NQS) is a UNIX-based utility that allows you to submit job requests for later execution. With NQS, you can submit jobs for execution on a local or remote workstation. Intel's Supercomputer Systems Division has enhanced NQS so that you can also submit jobs that are to be executed on an iPSC®/2 or an iPSC®/860 supercomputer. This chapter provides an overview of the NQS utility and the concepts that are related to NQS operations.

NQS REQUESTS

The two request types recognized by NQS, *batch* and *device* requests, provide you with enough flexibility to perform most operations. These request types are used to address one of four queue types: *batch*, *device*, *pipe*, or *network* queue types. The following sections provide more detailed descriptions of the request and queue types used by the NQS utility.

Request Types

The NQS utility recognizes two request types: batch requests and device requests.

A batch request is defined as a shell script that only contains commands that:

- Do not require the direct services of a physical device (other than the CPU that is executing the batch request).

  and

- Can be executed without any user intervention when the proper command interpreter (such as */bin/csh* or */bin/sh*) is invoked.

A device request is defined as a set of independent instructions that require the direct services of a specific device (such as a line printer) for execution.

In general, the request type directly relates to the type of queue to which the request is being sent. For example, a batch request ultimately ends up in a batch queue. Pipe and network queues accept both request types, but these queue structures ultimately pass all requests off to either batch or device queues.

Request States

Each request, whether a batch or a device request, may be in any of a number of different states at a given time. The request states currently supported by NQS are as follows:

arriving	Requests in this state are in the process of being queued from another (possibly remote) pipe queue. After leaving this state, the request will enter the waiting, queued, running, or routing state.
queued	The request has been accepted in a queue, and is completely ready to enter the running or routing request state.
routing	A request residing in a pipe queue is being routed and delivered to another queue destination.
running	A request residing in a batch or device queue is currently being executed.
waiting	The request is waiting for some finite time interval to pass. After leaving this state, the request will enter the queued, running, or routing state.

NQS DAEMONS

Each machine in the NQS environment must set up and use three separate daemons for NQS to function properly. The daemons consist of the *local daemon*, the *log daemon*, and the *network daemon*.

The *local daemon* processes requests executing on the local machine. It manages the batch queues and schedules all requests for execution. It also detects and forwards any requests that are to be executed on a remote machine.

The *log daemon* records a log of NQS requests and activities.

The *network daemon* handles all remote requests. It is essentially the same as the *local daemon* that forwards requests, but it receives its requests through Berkeley sockets from remote machines.

NQS commands communicate with the NQS daemons through UNIX pipes and Berkeley sockets. The daemons are permanent and are created when NQS is started up. It may be instructive to observe the process followed by a batch request. Figure 1-1 charts the path followed by a request, and shows the relationship between daemons, requests, and NQS commands.

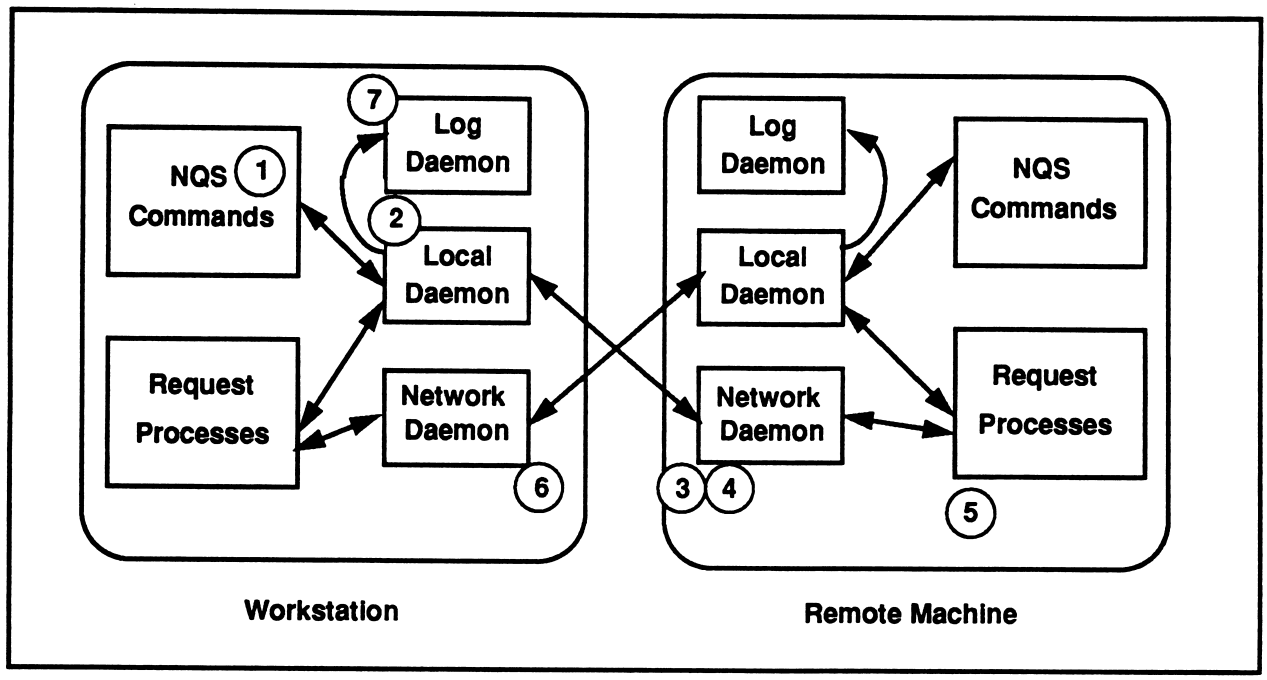

Figure 1-1. NQS Network Request Processing

A typical job flow from a workstation to a remote machine is shown in Figure 1-1. The sequence of events followed for the Figure 1-1 process is as follows:

1. The user submits a request. For the iPSC environment, the user could submit a batch request from the workstation to the iPSC system (the remote machine). This request would be submitted using the NQS **qsub** command as follows:

```
% qsub -x -q remote_q_1 script1
```

2. The **qsub** command packages up the job and sends it to the *local daemon*.

3. The local daemon determines that the job is remote and sends it (via a socket call) to the network daemon on the remote machine.

4. The network daemon on the remote machine retrieves the shell script from the spool directory, and sets up the *stdout* and *stderr* file descriptors.

5. The shell script is executed.

6. After the shell script executes, the *stderr* and *stdout* files are sent back to the originating machine through instructions to the originating machine's network daemon.

7. A message is sent to the log daemon to have an entry placed in the log file for the job.

There are many more details that apply to request processing, but this general sequence shows the physical and chronological relationships that exist between machines in a network.

NQS QUEUES

The four queue types recognized by NQS are the *batch*, *device*, *pipe*, or *network* queue types. The following sections provide more detailed descriptions of the queue types used by the NQS utility.

Batch Queues

As was stated at the beginning of this chapter, all batch requests ultimately end up in a batch queue. The destination (target) batch queue may exist at the same location where the request originated, or at a remote location.

BATCH EXECUTION SEQUENCE

The execution sequence of a batch request within the target batch queue is as follows:

1. A temporary copy of any output file is created in a location that is known to NQS. This allows output files of *stderr* and *stdout* to be spooled to other (possibly remote) destinations.

2. Any additional environment variables associated with the request are placed in the environment set for the shell that is about to be executed. These variables are optionally exported with the request from the originating host.

3. The local host for the destination batch queue interprets the request shell specifications and its own shell strategy policy, then chooses the proper shell (e.g., */bin/csh*, */bin/ksh*, */bin/sh*, etc.). The batch request will be executed within this chosen shell. The chosen shell is virtually indistinguishable from the shell that the user would get by logging directly into the machine.

4. The resource limits as determined when the request was queued are applied to this new shell process.

5. The proper shell is *executed* and the shell script that defines the *batch* request is executed.

6. The spooled output files (*stderr* and *stdout*) are returned to their (possibly remote) machine destinations.

BATCH RESOURCE LIMITS

Both the batch request and the destination batch queue have defined resource limits associated with them. The resource limits of the destination queue take priority over those of the request, but at the *time the request is first queued* to the destination batch queue, the resource limits for that request are frozen. If the system manager changes resource limits for a batch queue, the limits for all existing requests are "grandfathered" (i.e., requests that have been queued are allowed to complete regardless of their limits), so those requests will not be affected by the new limits.

BATCH RUN LIMITS

In addition to resource limits, NQS imposes a set of run limits on batch queues. These run limits are used to ensure that the local host will not get swamped with running batch requests at any one time. The following run limits apply to batch queues on the iPSC/860 SRM:

- The *global batch run limit* places a ceiling on the maximum number of *batch* requests allowed to execute simultaneously on the local host. This global limit applies to the local host, and applies to all requests on this host, regardless of the queue destination.

- The *queue run limit* places a ceiling on the maximum number of batch requests allowed to execute simultaneously in the containing batch queue.

SPAWNING A BATCH REQUEST

The process followed when spawning batch requests is as follows:

1. As a batch request completes execution, the entire set of batch queues is examined in decreasing order of batch queue priority. This imposes a *priority* order among all batch queues, and a *chronological* order on the requests within each queue.

2. Each batch request in the batch queue is spawned as it is examined until either the queue run limit or the global batch run limit is reached.

3. The next lower priority batch queue is examined, and batch requests are spawned using the above criteria.

4. The process continues until all batch queues have been examined.

5. The process restarts at the highest priority batch queue when the next batch request completes execution.

The relatively simple prioritization scheme listed above is able to control a large number of diverse and complicated batch requests without problem. More sophisticated control mechanisms (for example, to limit the number of simultaneously executing batch requests) may be added to NQS in the future.

Device Queues

Each *device* queue has one or more devices that are associated with it. In turn, each of these devices has a server associated with it. A server is a program that is always spawned to handle a request that is given to a device for execution.

DEVICE QUEUE MAPPING

Device queues use two dimension (*n*,*m*) mapping, and support a wide range of mapping options. In general, "n" device queues can be configured to feed "m" devices. The device queue dimensions must be greater than or equal to 0 (integers only), but there are no other restrictions on dimensions. It is possible for one device queue to feed several devices, or for several device queues to feed the same device. It is even possible (though probably not useful) for the mapping to include zero devices or device queues.

SPAWNING A DEVICE REQUEST

The activity at the device itself controls spawning of device requests. In contrast, the local host monitors and controls batch request processing. The following process is used when a device request is spawned:

1. *Either* a device completes handling of a device request.

   Or a device is found to be idle after a device request is queued to its associated device queue.

2. All of the device queues that feed the device are scanned to determine if they have requests that can be handled by the device.

3. If particular *forms* are specified with a device request, they must match the forms defined for the device. A forms type is a set of parameters (for example, margins and font specifications for a printer) that determine operating characteristics for a device. A device can have many forms defined at the same time. If a forms type is not specified with a request, it is assumed any device associated with the device queue can satisfy the request, and the default forms type will be used.

4. If two or more device queues are found with requests that the newly idled device can handle, the queue with numerically higher queue priority is chosen.

5. If two or more device queues have the same queue priority, the queues are serviced in "round robin" fashion.

6. Once the device completes handling of a request, it resumes the spawning process at step #1.

Any run limits on device queues are imposed by the associated devices. System administrators can limit the impact of device queues on system operation by changing device queue mapping as necessary.

Pipe Queues

Pipe queues are responsible for routing and delivering requests to other (possibly remote) queue destinations. Pipe queues are similar in concept to a pipeline, because they transport valid requests to other queue destinations. Pipe queues can accept either batch or device requests, and transport these requests to batch, device, other pipe, or network queues.

Pipe queues do not have any associated quota limits or devices, but they do have an associated set of destinations to which requests may be routed. Valid destinations may be batch, device, other pipe, or network queues that exist on the local host, or on other hosts or devices.

SPAWNING A PIPE REQUEST

Each pipe queue has an associated server that is spawned to service each request that passes through the queue. Because of the use of the word server (in the context of a *client/server* network connection), the spawned instance of a *pipe queue server* is called a *pipe client*.

When a request is routed through the pipe queue, the pipe server spawns a pipe client, and this client must then route and deliver the request to a destination. For each attempted remote destination, a network server process must be created on the remote host to act as agent for the pipe queue request.

The pipe client is given complete freedom to choose any of the destinations from the destination set defined for this pipe queue to receive the pipe queue request. If a destination does not accept the pipe queue request, the pipe client is free to try another destination for the request. The pipe client only needs to find one destination that will accept the request in order to be successful.

A pipe queue request may be rejected at a destination for a variety of reasons. Some reasons for rejection are related to the underlying request type, and are not due to a pipe queue problem. For example, a piped device request may have an unsupported forms type, or a batch request may conflict with a run limit for the only available destinations. Rejection reasons more directly related to a pipe queue problem may be divided into "temporary" and "permanent" types of problems. A "temporary" problem could be that a destination has crashed and will likely be rebooted in the future. A "permanent" problem might be a lack of proper account authorization at the destination.

Pipe queues are both powerful and complex. The NQS manager can use the **qmgr** subcommands to configure the pipe queues to perform many tasks. The pipe client can be set up to examine a request and determine an appropriate destination queue. For example, "large" batch requests might be sent to a queue that only runs at night, while "small" or "high priority" batch requests would be sent to fast batch queues. Pipe queues can be used to ease network and hardware reliability problems. If a destination machine is down or otherwise inaccessible, the request can be requeued for delivery at some later time.

PIPE QUEUE RUN LIMITS

Pipe queue run limits are similar to the set of run limits placed on batch queues. These run limits are used to ensure that the local host will not get swamped with running pipe requests at any one time. The following run limits apply to pipe queues:

- The *global pipe run limit* places a ceiling on the maximum number of pipe requests that may execute simultaneously on the local host. This global limit applies to the local host, and applies to all requests on this host, regardless of the queue destination.
- The *queue run limit* places a ceiling on the maximum number of pipe requests allowed to execute simultaneously in the containing pipe queue.

Network Queues

A queue type of *network* is implicitly used by pipe queues to pass NQS requests between machines, and also to perform queued file transfer operations. NQS supports network operations within the UNIX environment, and observes the conventions established for UNIX networks. NQS network conversations use a stream connection, and assume that the requisite bytes will be transmitted to/from a server in the same order in which they were written.

All network conversations are performed using the classic *client/server* model, in which a client process on the local host creates a connection to the remote machine. At the remote machine, a *network server* process is created to act on behalf of the client process.

When the initial client/server connection for a network queue is established, some standard information is exchanged between the processes. This standard information includes user-id, user name, and the time-zone in effect at the client machine.

After the connection is established, the network server (at the destination machine) performs the account mapping for the network. This mapping is performed for all network conversations so that the ownership of batch and device requests can be established and adjusted as required. Performing the account mapping at the remote machine ensures that some semblance of security is maintained for the network.

NQS USER OPERATIONS

NQS recognizes two user types and grants user privileges based upon the user type of the account. An NQS *operator* identifies a person who can execute only the operator commands as a proper subset of all the commands provided by the **qmgr** utility. An NQS operator is able to submit batch and device queue requests, and perform other operations associated with these requests.

An NQS *manager* is able to define, configure, and manage queues in addition to performing all the functions that an NQS operator can perform. An NQS manager identifies a person who is capable of changing any NQS characteristic on the local machine.

NQS Operator Commands

The full NQS command set is available to both the NQS operator and the NQS manager, but only a subset of the **qmgr** functions are available to an NQS operator. Table 1-1 lists the **qmgr** functions (subcommands) that are available to NQS operators.:

Table 1-1. QMGR Functions for NQS Operators

Abort Queue	Show Device
Disable Device	Show Forms
Disable Queue	Show Limits_supported
Enable Device	Show Long Queue
Enable Queue	Show Managers
Exit	Show Parameters
Help	Show Queue
Lock Local_daemon	Shutdown
Purge Queue	Start Queue
Set Run_limit	Stop Queue
Show All	Unlock Local_daemon

In general, an NQS operator is able to manage any operator-owned queues and requests, to get information (**Show** and **Help** functions), and to perform some network operations (lock/unlock the local_daemon). All other operations are reserved for the NQS manager. Chapter 2 of this manual describes how to use the operations and commands that are available to NQS operators.

NQS Manager Commands

An NQS manager has full access to all NQS utilities, and to the full set of functions for the NQS **qmgr** command. NQS manager privileges are initially granted when NQS is configured by the UNIX system manager. Additional NQS manager privileges may be extended to other users by any valid NQS manager. Unless a user has been granted manager privileges, only NQS operator privileges are available.

An NQS manager is able to add/create/delete any of the NQS queue structures, forms, users, managers, and groups. A manager is also able to define limits, operation parameters, strategies, and "nice values". Chapter 3 of this manual describes how to use the operations and commands that are restricted to users with NQS manager privileges.

NETWORKING

You can work on one workstation and submit requests to another workstation. To do this, you submit the request to a pipe queue. The pipe queue has attached to it a number of destination queues. The request is submitted to the first destination queue that accepts it.

The destination queues do not necessarily have to reside on other machines. The destination queues may be batch queues, device queues, or other pipe queues.

NQS AND THE iPSC®/860 SUPERCOMPUTER

When you submit an iPSC/860 job to NQS, you must first set the shell environment variable CUBETYPE to what would normally be the *cubetype* argument to **getcube**. You cannot include the *cubename* and *srmname* options.

For example, assume that you are running the C shell and that you want to submit an iPSC/860 job that requires a two-dimensional RX cube. Define the environment variable as follows:

```
% setenv CUBETYPE 4rx
```

The only way that NQS knows that your job is intended for an iPSC/860 supercomputer is through this environment variable. You must ensure when you submit the job with the NQS command **qsub** that you include the **-x** switch, which indicates that all environment variables are to be exported with the request. For example, you could submit a job named "myscript" as follows:

```
% qsub -x myscript
```

What you actually submit is a shell script containing the iPSC/860 **load** command followed by a **waitcube**. The shell script must *not* contain a **getcube** or **relcube**. Those functions are performed by NQS. For example, here is a typical C shell script that loads a node program called *myprog*.

```
#
load myprog
waitcube
```

The **waitcube** is necessary because the **load** comes back before there is any guarantee that the node program is finished. Without the **waitcube**, NQS would consider the iPSC/860 job completed after the load. NQS allocates a cube when it sees CUBETYPE as part of its job's environment and releases the cube when the job is completed. The **waitcube** ensures that the nodes remain allocated for the extent of your job.

NQS tries to allocate the number of nodes requested by the first cube job it encounters. If that number is unavailable, NQS keeps trying. The iPSC/860 request remains at the head of the queue until it is honored.

When a iPSC/860 supercomputer is part of your network environment, you should take special care when setting up your NQS queues. For example, consider the following situation. You have a 32-node iPSC/860 supercomputer and one NQS batch queue whose run limit is 5. A job is already running on 16 nodes of the iPSC/860 supercomputer. A job requesting 32 nodes comes to the head of the queue. Nothing else comes out of the queue until all 32 nodes are available. This is true even if the jobs behind the 32-node request are non-iPSC/860 jobs and the run limit is not exceeded. It is also true if the next job is an iPSC/860 job requiring only 16 nodes, which in this example are available.

There are several ways of dealing with this situation. First of all, you should not mix iPSC/860 jobs and non-iPSC/860 jobs in the same queue. Set up one queue for jobs requiring only the resources of the workstations and other queues for jobs requiring iPSC/860 nodes. If you do this, you won't be in the situation where a waiting iPSC/860 job prevents a non-iPSC/860 job from executing.

Also, you might consider setting up several iPSC/860 queues, one for each size of cube. For example, in a 32-node system, you could have a queue for 1 node, another for 2 nodes, a third for 4 nodes, up to a sixth for 32 nodes. In this way an iPSC/860 job always gets to run if the requested number of nodes are available.

NQS uses the UNIX mail facilities extensively. The *stderr* and *stdout* files returned by NQS are typically returned to the originating machine using UNIX mail facilities. Refer to the "Setting Up the SRM Mail Service" section on page 3-5 for more information on setting up the UNIX mail facilities for NQS operations.

GETTING STARTED WITH NQS 2

INTRODUCTION

The Network Queueing System (NQS) utility allows you to package numerous jobs as batch or device requests, and submit them to any of the supported queue types for later processing. The two levels of NQS user, operator and manager, have different tasks associated with them. An NQS operator is concerned with using the NQS utility to accomplish other tasks, such as submitting a print job or submitting a processing job to a cube of the iPSC/860 system. An NQS manager is concerned with managing the NQS environment to ensure that other NQS users can best utilize the other resources connected to NQS. This chapter is a "User's Guide" for NQS *operators*.

DEFINING WORKSTATION PROPERTIES

You may need a special set of resource limits and a specific shell strategy in place for your workstation. As an NQS operator you are able to determine the resource limits and shell strategy in place for elements of the NQS system, but you must get someone with NQS manager privileges to change the NQS properties on your workstation.

The NQS **qlimit** command displays the current shell strategy and the supported resource limits for your workstation or any other workstation in the network. As an NQS operator, you can also use the **SHOw All** or **SHOw LImits_supported** subcommands of the **qmgr** program to determine the limits and shell strategy applying to the local machine. An NQS manager must redefine the shell strategy if you want it changed.

The shell strategy for your workstation determines the shell used by NQS when it interprets the shell script for batch requests. You can have a particular shell for all requests, use the shell that is the one specified in the user's **passwd** entry, or use the shell that would be used if the request were executed interactively. The detailed description of the **qlimit** command in Chapter 4 discusses shell strategy in more detail.

The three possible shell strategies, and the resulting actions are:

- The *fixed* strategy will cause the configured fixed shell to be executed to interpret all batch requests.
- The *free* strategy will cause the user's login shell as defined in the password file to be executed which in turn chooses and spawns the appropriate shell for running the batch shell script.
- The *login* strategy will cause only the user's login shell to be executed to interpret the script.

A shell strategy of *fixed* means that the same shell as chosen by the system administrator, will be used to execute all batch requests.

A shell strategy of *free* will run the batch request script exactly as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of *fixed* and *login* exist for host systems that are short on available free processes. In these two strategies, a single shell is executed, and that same shell is the shell that executes all of the commands in the batch request shell script.

The following examples show the respective displays returned when the **qlimit** and **SHOw All** commands are invoked:

```
% qlimit
  Per-process permanent file size limit (-lf)
  Nice value (-ln)

  Shell strategy = FREE

% qmgr
Mgr: show all
      :
      :
      :
  Batch request shell choice strategy = FREE

Limits supported:

  Per-process permanent file size limit (-lf)
  Nice value (-ln)
      :
      :
      :
```

The **qlimit** command is invoked from the shell, while the **SHOw All** subcommand must be invoked from within the **qmgr** environment. The information returned by either command is equivalent, but the **SHOw All** subcommand returns considerable additional information. The examples show the limits supported for the SRM. Other systems or machines will support different resource limits.

The previous examples show only resource limits supported by your workstation, not the current resource limits. You must use the **SHOw LOng Queue** subcommand of the **qmgr** program to see the resource limit values in place for your workstation under each defined queue. The following example shows a portion of the display returned by this subcommand:

```
Mgr: show long queue
e2@nqs_h;  type=BATCH;  [ENABLED, INACTIVE];  pri=0
      0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
  Run_limit = 1;
  Cumulative system space time = 82.52 seconds
  Cumulative user space time = 67.05 seconds
  Unrestricted access
  Per-process permanent file size limit = 1 megabytes <DEFAULT>
  Per-process execution nice value = 0 <DEFAULT>
     :
     :
     :
```

As was shown for the **qlimit** command, only the "Per-process `permanent file size limit`" and "Per-process `execution nice value`" are supported for the iPSC SRM. Your workstation resource limits can be changed by using **qsub** flags and defining new values when a batch request is submitted. You can specify other limit values when submitting (using the **qsub** command) batch requests by setting the appropriate flags (**`-lf`** and **`-ln`**) and specifying the new values. If you try to define resource limits that cannot be supported by the underlying UNIX implementation, they are simply ignored. If you try to define resource limits that conflict with limits in place for the destination queue, or limits that will fail the NQS limit check for any reason, the batch request will not be accepted in the queue.

DISPLAYING NQS OPERATING PARAMETERS

The NQS operating parameters include default values and paths that apply to your workstation and its operation in the NQS environment. You can use the **SHOw All** or **SHOw Parameters** subcommands of the **qmgr** program to determine the parameters that apply to your workstation. The following example shows the display returned by the **SHOw Parameters** subcommands of the **qmgr** program:

```
Mgr: show parameters

  Debug level =  0
  Default batch_request priority = 31
  Default batch_request queue = bq
  Default destination_retry time = 72 hours
  Default destination_retry wait = 5 minutes
  Default device_request priority = 31
  No default print forms
  Default print queue = NONE
```

```
(Pipe queue request) Lifetime = 168 hours
Log_file = logfile
Mail account = root
Maximum number of print copies = 2
Maximum failed device open retry limit = 2
Maximum print file size = 1000000 bytes
Netdaemon = /usr/lib/nqs/netdaemon
Netclient = /usr/lib/nqs/netclient
Netserver = /usr/lib/nqs/netserver
(Failed device) Open_wait time = 5 seconds
NQS daemon is not locked in memory
Next available sequence number = 136
Batch request shell choice strategy = FREE
```

As an NQS *operator*, you are able to observe what operating parameters are in place for your workstation, but only an NQS *manager* is able to change any of these parameters.

SUBMITTING A BATCH REQUEST

In Chapter 1, you learned that there are fundamentally two request types in NQS, batch and device requests. Most NQS requests handled in a network (and all requests for the iPSC/860 system) are normally batch requests. The procedures in this section show you how to create, run, and monitor batch requests within the NQS environment.

Composing the Shell Script

A batch request is actually defined by a shell script. The proper shell is based on the shell strategy in place at the local host, and on the shell type that will be used when the script is executed. Keep in mind that, when the script is executed, the shell will be spawned as a login shell that is virtually indistinguishable from the shell you would get if you logged in directly on the target machine.

Compose the batch request shell script using a text editor of your choice. If you are unfamiliar with shell scripts, the standard UNIX documentation gives user and reference information on developing and invoking shell scripts. Your system administrator should also be able to help you if you need more information or assistance in developing the shell script(s) for your batch requests.

For example, assume that you have a node program called "mynode" that you want to run on a 16-node cube. Create a shell script that contains an iPSC/860 **load** command followed by a **waitcube** command, as follows:

```
load mynode
waitcube
```

This simple shell script contains only the commands to load your "mynode" program, and wait for it to execute. Additional commands can be included in a shell script as needed.

Submitting the Batch Request

The actual batch request is wrapped in the shell script that you composed in the previous section. You submit the batch request to a batch queue using the NQS **qsub** command. For example, assume that you have the node program called "mynode" that you want to run on a 16-node cube. First set the environment variable CUBETYPE to the desired node type.

```
% setenv CUBETYPE 16rx
```

You might also have a second job that you want to send to the same cube. In the following example, two jobs (`job1` and `job2`) have been queued to batch queue `bq`. Now issue the **qsub** commands.

```
% qsub -q bq -a 14:15 -x job1
Request 136.treebrd submitted to queue: bq.
% qsub -q bq -a 14:15 -x job2
Request 137.treebrd submitted to queue: bq.
```

The **-q** *queue* option specifies the queue that will accept the request. If the queue is not a batch queue, you must use this option. If you are submitting a request to a batch queue, you can leave the option out. If you do that, the command first looks for an environment variable in your user set called QSUB_QUEUE. You can set this value to a default batch queue. If you leave the option out and do not define QSUB_QUEUE, NQS uses the default set up by the system administrator with **qmgr**.

The **-a** option means execute these programs at time `14:15` hours. The **-x** option means inherit the user environment variables. This is a required option if your request requires the use of an iPSC/860 supercomputer. NQS needs the environment variable CUBETYPE to determine how many and what type of nodes to allocate. As shown in this example, you can immediately submit another request without waiting for the other request to finish. With all other parameters being equal, batch requests execute in the order they are submitted.

Monitoring Request Execution

You can monitor queue status and the completion status of requests using the **qstat** command. In the example in the prior section, two jobs were submitted for execution at the same time on the same batch queue. In the following example, the **qstat** command has been invoked three times so you can see what is reported before, during, and after the two batch requests are processed.

```
% qstat
bq@treebrd; type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   2 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
    1:          job1       136.treebrd             ted  31  WAITING
    2:          job2       137.treebrd             ted  31  WAITING

% qstat
bq@treebrd;  type=BATCH;  [ENABLED, RUNNING];  pri=0
  0 exit;   1 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
    1:          job2       137.treebrd             ted  31  RUNNING    4950

% qstat
bq@treebrd;  type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
```

The queue status is also available to NQS operators in the **qmgr** program by using the **SHOw LOng Queue** subcommand.

Deleting/Modifying a Batch Request

You may occasionally need to delete or modify a batch request after it has been submitted. In general, a batch request cannot be modified once it has been queued, but you can delete it and then resubmit a modified request. The most direct way to delete a batch request is with the **qdel** command. Refer to the **QDEL** description on page 4-3 for more information on operation and required syntax for this command. In the following example, *job1* is submitted, checked, and immediately deleted using the **qdel** command.

```
% qsub -q bq -a 16:15 -x job1
Request 139.treebrd submitted to queue: bq.
% qstat
bq@treebrd; type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   1 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
    1:          job1       139.treebrd             ted  31  WAITING

% qdel -k 139
Request 139 has been deleted.
% qstat
bq@treebrd;  type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
```

You can also interrupt and restart batch request execution or delete a batch request using the **ABort Queue, DIsable Queue, ENable Queue, Purge Queue, SHUtdown, STArt Queue,** and **STOp Queue** subcommands of the **qmgr** program. The disadvantage to using the **qmgr** subcommands (instead of the **qdel** command) is that they affect all requests in a given queue, while the **qdel** command allows deletion of a single request. In the following examples, several different means of stopping/deleting requests are shown, and their affect on the queue is also shown.

```
% qsub -q bq -a 16:15 -x job1
Request 140.treebrd submitted to queue: bq.
% qmgr
Mgr: show queue
bq@treebrd;  type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   1 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
   1:           job1       140.treebrd             ted  31  WAITING
              :
              :
              :
Mgr: disable queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue
bq@treebrd;  type=BATCH;  [DISABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   1 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
   1:           job1       140.treebrd             ted  31  WAITING
              :
              :
              :
Mgr: stop queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue
bq@treebrd;  type=BATCH;  [ENABLED, STOPPED];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   1 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
   1:           job1       140.treebrd             ted  31  WAITING
              :
              :
              :
Mgr: purge queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue
bq@treebrd;  type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
              :
              :
              :
```

The only subcommand in the above examples that really deleted the request was **Purge Queue.** The **ABort Queue** and **SHUtdown** subcommands can also be used to delete a *running* request, but these subcommands should only be used if other options are unsuccessful.

USING PIPE/NETWORK QUEUES

Either batch or device requests can be used with pipe or network queues. These queues are pathways to the ultimate destination (either a batch or a device queue). A batch request will only be accepted in a pipe or network queue that has batch queue(s) as an ultimate destination. A device request will only be accepted in a pipe or network queue that has device queue(s) as an ultimate destination.

Pipe and network queues can only be created, deleted, or modified by an NQS manager. As an NQS operator, you can use pipe and network queues but you cannot change them.

Submitting a Pipe Request

You can use the standard NQS commands (**qsub** for batch requests or **qpr** for device requests to printers) when submitting a request to a pipe or network queue. You should be aware of the destinations of the pipe and network queues in your system before submitting requests to the queues. If you are unsure of these destinations, contact your system administrator or the NQS manager.

```
% qsub -q pq -a 10:10 -x job1
Request 144.treebrd submitted to queue: pq.
```

The **-q** *queue* option specifies the pipe queue that will accept the request. Because this is a pipe queue, you must use this option. The **-a** option means execute these programs at time 10:10 (ten minutes after ten o'clock AM local time). The **-x** option means inherit the user environment variables. This is a required option if your request requires the use of an iPSC/860 supercomputer. You can immediately submit another request without waiting for earlier requests to finish.

Monitoring Request Execution

You can monitor the execution of requests to *pipe* and *network* queues in the same manner as you would monitor request execution for other queue types. Use the NQS **qstat** command to monitor the status of queues and requests in the NQS environment. Additionally you can use the **qdev** command to monitor the status of devices in the NQS environment. The following example shows the status reported for the pipe request that was submitted in the prior section.

```
% qstat
bq@treebrd; type=BATCH;  [ENABLED, INACTIVE];  pri=0
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;


pq@treebrd;  type=PIPE;  [ENABLED, STOPPED];  pri=30
  0 depart;   0 route;   1 queued;   0 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     PGRP
   1:           job1       144.treebrd             ted  31   QUEUED
```

If, for some reason, the pipe or destination queues are unable to process a request, NQS sends mail to the originator and deletes the request. If the request fails to execute after being accepted, a message is sent to *stderr* and to the NQS log file. Refer to the "Setting Up the SRM Mail Service" section on page 3-5 for more information on setting up mail service for the iPSC system. The following example shows the messages and mail returned when a request to a pipe queue fails to complete.

```
% qsub -q pq -a 10:10 -x job1
Request 143.treebrd submitted to queue: pq.
You have new mail.

% mail
From root Wed Jan  9 18:05 GMT 1991
Subject:  NQS request:  143.treebrd failed.

Request name:   job1
Request owner:  ted
Mail sent at:   Wed Jan  9 10:05:06 PST 1991

Server for request did not return a completion code.
Request failed.
Request files placed in NQS failed directory.
```

Possible reasons for request failure at the destination include the following:

- Remote host failure.
- Queue type disagreements with the request type.
- Insufficient queue space.
- Destination queue disabled (unable to accept new requests).
- Lack of proper account authorization.

There are many other possible reasons for request failure, even after it is initially accepted. NQS supports "wait" and "retry" parameters that may allow an initially failed request to be ultimately successful.

SUBMITTING A DEVICE REQUEST

Device requests must be ultimately submitted to a device queue. A device request will only be accepted in a pipe or network queue that has device queue(s) as an ultimate destination. As with any other queue, after being created a device queue must be enabled (**Enable Queue** subcommand of **qmgr**) and started (**Start Queue** subcommand of **qmgr**) before the queue is active.

Device queues can only be created, deleted, or modified by an NQS manager. As an NQS operator, you can enable, disable, and use device queues, but you cannot change them.

Enabling/Disabling a Device

As an NQS operator, you have a limited set of commands available to affect operations at devices. Two of the actions you can perform are to disable or enable a device. The **Disable Device** and **Enable Device** subcommands of the **qmgr** utility allow you to directly control a device and the queues feeding that device. Although the **Enable Queue** and **Disable Queue** subcommands do not directly affect a given device, by controlling a queue feeding that device, these subcommands can have the same effect as the **Disable Device** and **Enable Device** subcommands. The key difference between these subcommands is that the "device" version stops or starts all queues feeding the device while the "queue" version affects a single queue at a time.

In the following sequence, the **Disable Queue**, **Disable Device**, **Enable Queue**, and **Enable Device** subcommands are issued to control a queue named `dq` and a device named `pr1`.

```
Mgr: enable queue dq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: start queue dq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue dq
dq@treebrd;  type=DEVICE;  [ENABLED, INACTIVE];  pri=60
  0 run;   0 queued;   0 wait;   0 hold;   0 arrive;

Mgr: exit
% qpr -q dq job1
Request 152.treebrd submitted to queue: dq.
% qstat dq
dq@treebrd;  type=DEVICE;  [ENABLED, INACTIVE];  pri=60
  0 run;   1 queued;   0 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE      SIZE
   1:           job1       152.treebrd             ted   31   QUEUED         8
```

```
% qmgr
Mgr: disable queue dq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: disable device prl
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: exit
% qpr -q dq job2
Queue is disabled at local host.
Retry later.
% qmgr
Mgr: enable queue dq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue dq
dq@treebrd;  type=DEVICE;  [ENABLED, INACTIVE];  pri=60
  0 run;   1 queued;   0 wait;   0 hold;   0 arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE     SIZE
   1:           job1       152.treebrd             ted  31   QUEUED        8
Mgr: enable device prl
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue dq
dq@treebrd;  type=DEVICE;  [ENABLED, INACTIVE];  pri=60
  0 run;   0 queued;   0 wait;   0 hold;   0 arrive;

Mgr: exit
%
```

No device requests can be processed by device `prl` until both the device queue is enabled and started, and the device is enabled.

Spawning the Device Request

Device requests must be submitted using the NQS **qpr** command. The name of this command reflects the fact that the majority of all device requests are directed to printers.

You can indirectly send requests to a specific device by using the **-q** switch to send the request to a specific queue, or by using the **-f** switch to send the request only to a device that supports a specific forms. If the queue or forms specified by these switches only apply to a single device, it has the same effect as directing the request to only that device. There is no other method (using the **qpr** command) to direct a device request to a specific device.

As was stated in the "Submitting a Pipe Request" section on page 2-8, you can send device requests through a pipe queue and thus gain an added measure of control over the device queue.

Deleting/Modifying a Device Request

You may occasionally need to delete or modify a device request after it has been submitted. In general, a device request cannot be modified once it has been queued, but you can delete it and then resubmit a modified request. The most direct way to delete a device request is with the **qdel** command. Refer to the **QDEL** description on page 4-3 for more information on operation and required syntax for this command.

You can also delete a request using the **Purge Queue** subcommand of the **qmgr** utility, but this action kills all requests in the stated queue. This action may have the same result as the **qdel** command, but other users may not appreciate having their requests deleted.

NQS PRINT OPERATIONS

In the iPSC/860 system the only supported device queues are actually print queues. Most devices in the NQS environment will actually be printers. You submit requests to printer devices using the NQS **qpr** command. As with any other request type, you can monitor print request execution using the **qstat** command. The following example shows submission of a print request to a pipe queue using a typical set of parameters.

```
% qpr -a 12:55 -mb -mu mynode -q pq makefile
Request 145.treebrd submitted to queue: pq.
%
% qstat pq
pq@treebrd;  type=PIPE;  [ENABLED, STOPPED];  pri=30
  0 depart;    0 route;    1 queued;    0 wait;    0 hold;    0 arrive;

         REQUEST NAME          REQUEST ID              USER  PRI     STATE      PGRP
    1:       makefile        145.treebrd                ted  31    QUEUED
```

The **-q** *queue* option specifies the device or pipe queue that will accept the request. If this option is not used, NQS searches for the default variable (QPR_QUEUE), and submits the request to that queue. The **-a** option means execute this program at 12:55 (fifty five minutes after twelve noon local time). The **-mb** and **-mu** options indicate mail is to be sent (and redirected to "mynode") when the request begins executing. You can immediately submit another request without waiting for earlier requests to finish.

As with other NQS commands, an NQS operator can submit print requests, but only an NQS manager can change any of the parameters (such as a forms definitions) that will affect the destinations and the formatting of print operations.

MANAGING NQS OPERATIONS 3

INTRODUCTION

An NQS manager is able to control the NQS environment to ensure that other NQS users can make use of the resources connected to the network. At a minimum, there must be one NQS manager in the network (the NQS network manager). The NQS network manager has the following responsibilities:

- Maintain and assign machine identification numbers (mid) for all machines in the NSQ network.
- Establish and/or maintain any queues that are visible to or reachable by all NQS users. Local queues can be established and maintained by the manager for the local machine.

For the most part, all NQS manager functions are performed using the **qmgr** utility. The full command reference for the **qmgr** utility is presented in Chapter 4. This chapter is a "User's Guide" for NQS *managers*.

USING THE QMGR UTILITY

To create a queue, you need NQS manager privileges. Both NQS operators and NQS managers can use the **qmgr** utility, but operators can only use a restricted subset of the **qmgr** subcommands. First you should ensure that you have manager privileges. Invoke the **qmgr** utility and enter the **SHOw Managers** subcommand as follows:

```
% qmgr
Mgr:show managers

  ted:m
  root:m
```

You must have NQS manager privileges in order to use most of the **qmgr** subcommands. If you do not have manager privileges, you will have to get an existing manager (or login as "root") to name you as manager using the **ADd Managers** subcommand. This command is used to add operators, add managers, or change the privilege level of users in the NQS environment.

Defining Limits

NQS supports an extensive set of batch request resource quota limits, but it is limited by the underlying UNIX implementation in the limits it supports on a given machine. In Chapter 2 you were shown how to check the limits supported using both the **qlimit** command and the **SHOw All** subcommand of the **qmgr** utility. As a manager, you can change the limits that will be observed when the **qsub** command is invoked. The following example shows the supported limits, then changes the two resource limits (and the run limit) supported for the example queue.

```
Mgr: show limits_supported

  Per-process permanent file size limit (-lf)
  Nice value (-ln)

Mgr: set run_limit = 2 myq1
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: set per_process permfile_limit = (500 kb) myq1
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: set nice_value_limit = 3 myq1
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: show long queue myq1
myq1@treebrd;  type=BATCH;  [ENABLED, INACTIVE];  pri=50
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
  Run_limit = 2;
  Cumulative system space time = 0.00 seconds
  Cumulative user space time = 0.00 seconds
  Unrestricted access
  Per-process permanent file size limit = 500 kilobytes
  Per-process execution nice value = 3
```

Note that in the above example, as in most of the other examples in this chapter, the full text of the **qmgr** subcommands is used. This is done so no mistake will be made in command usage. During a normal session, however, you would typically use command abbreviations instead of the full text.

Setting Up Batch Queues

You define a new batch queue using the **Create Batch_queue** subcommand of **qmgr**. For example, to create a batch queue called *myq1* with a priority of 50 and a run limit of 5, enter:

```
Mgr: create batch_queue myq1 priority=50 run_limit=5
Queue myq1 created.
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
```

The priority is an inter queue priority, not a UNIX execution priority. The range is 0...63, with larger numbers indicating higher priority. The other option not shown in the example is to identify the queue as pipe only (*PIpeonly*). A pipe only queue will only accept requests if their source is a pipe queue. In a network environment, you might set up a batch queue to handle incoming requests from other workstations. Users on those remote workstations would submit their requests to a pipe queue which would have this batch queue as its destination. The **SHOw Queue** subcommand displays some characteristics of the queue you just created. This command needs only operator privilege.

```
Mgr: show queue
myq1@treebrd;  type=BATCH;  [DISABLED, STOPPED];  pri=50
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
```

Notice that the queue has an unavailable status. After creating a queue, you must both enable it and start it. The order does not matter. Disabled queues do not accept requests. Stopped queues accept requests, but put them in a frozen state. The frozen requests do not run until the queue is started. If you stop or disable a queue, requests that are currently running are allowed to complete. Requests in the queue that have not yet run are frozen.

To enable a queue, use the **ENable Queue** subcommand, and to start a queue, use the **STart Queue** subcommand.

```
Mgr: enable queue myq1
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue myq1
myq1@treebrd;  type=BATCH;  [ENABLED, STOPPED];  pri=50
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;

Mgr: start queue myq1
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue myq1
myq1@treebrd;  type=BATCH;  [ENABLED, INACTIVE];  pri=50
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
```

The queue *myq1* is now ready to accept requests.

Setting Up Pipe Queues

There is very little difference between setting up a batch queue and setting up a pipe queue. Batch queues have more associated limits, while a pipe queue must have a destination associated with it. For example, to create a pipe queue, issue the following command:

```
Mgr:  create p mypl pr=10 s= (/usr/lib/nqs/pipeclient) d=myql
Queue pq created.
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
```

You must have NQS manager privilege to create a pipe queue. This command used some standard NQS abbreviations. The **p** after **create** signifies a pipe queue; the **pr** is priority; the **s** is **server**; and the **d** is **destination**. With each pipe queue, there is an associated server that is spawned to handle each request released from the queue for routing and delivery. This is the specified **server**. Its instance is called a pipe client.

As with the batch queue, a pipe queue must be both enabled and started before it can be used. Do the same with *mypl* as you did with *myql*, but this time reverse the order of the enable and start. Start m*ypl* and then enable it.

```
Mgr: show queue mypl
mypl@treebrd;  type=PIPE;  [DISABLED, STOPPED];  pri=10
  0 depart;    0 route;    0 queued;    0 wait;    0 hold;    0 arrive;

Mgr: start queue mypl
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue mypl
mypl@treebrd;  type=PIPE;  [DISABLED, INACTIVE];  pri=10
  0 depart;    0 route;    0 queued;    0 wait;    0 hold;    0 arrive;

Mgr: enable queue mypl
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
Mgr: show queue mypl
mypl@treebrd;  type=PIPE;  [ENABLED, INACTIVE];  pri=10
  0 depart;    0 route;    0 queued;    0 wait;    0 hold;    0 arrive;
```

You can get more information about a this (or any other) queue with the **SHOw LOng Queue** subcommand.

```
Mgr: show long queue mypl
mypl@treebrd;  type=PIPE;  [ENABLED, INACTIVE];  pri=10
  0 depart;   0 route;   0 queued;   0 wait;   0 hold;   0 arrive;
  Run_limit = 1;
  Cumulative system space time = 0.00 seconds
  Cumulative user space time = 0.00 seconds
  Unrestricted access
  Queue server: /usr/lib/nqs/pipeclient
  Destset = {myql@treebrd};
```

To exit the **qmgr** utility, issue the **EXit** subcommand.

```
Mgr: exit
```

CONTROLLING OTHER NQS FUNCTIONS

NQS manager privileges are required to set up new queue structures, to add new users to the NQS environment, and to change privileges of existing users. Other tasks that NQS managers need to perform include setting up the UNIX mail services in a form recognizable to NQS, and setting up the various log files that NQS maintains. The remaining sections in this chapter describe these other functions.

Setting Up the SRM Mail Service

NQS uses the UNIX mail facilities extensively. The *stderr* and *stdout* files returned by NQS are typically returned to the originating machine using UNIX mail facilities. In order for NQS to send mail to a remote destination, the mail software on the SRM must be configured correctly. Refer to the *UNIX System V Release 3.2 TCP/IP Administrator's Guide and Reference*, Chapter 4 for specific details on setting up a **sendmail** service mail capability for the SRM. You must set the iPSC/860 SRM up before it can successfully return mail to remote destinations.

NOTE

You must be logged in as "root" in order to establish the **sendmail** service mail capability.

It is important that you use the UNIX **mailx** command, and not the **mail** command when sending mail to remote destinations. If you establish **sendmail** service to the central mail node for the network, additional **sendmail** services will not be required for other remote destinations.

Queue Access Controls

There are several method available for controlling access to the different queue types in NQS. None of these methods directly control the users that are allowed access to individual queues, but by placing restrictions on the request types and sources of requests, these NQS access controls provide a limited level of control over operations.

The first way of controlling access to a queue is through the **SEt NO_Access** or **SEt UNrestricted_access** subcommands of the **qmgr** utility. When created, a queue has unrestricted access by default. After the queue is created, you can use the **SEt NO_Access** subcommand to restrict all access to the queue, and then selectively give users or groups access to the queue. In order for a user to have access to a queue, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited. Use the **SEt UNrestricted_access** subcommand to give all users access to the queue.

2. The user's login group has access. Use the **ADd Groups** subcommand to give the group access to the queue.

3. The user's account has access. Use the **ADd Users** subcommand to give the user access to the queue.

An additional way of restricting access to a queue is to use the *PIpeonly* parameter when creating the queue. The *PIpeonly* parameter forces all requests entering the queue to come only from a pipe queue. This means a request cannot come directly from a user, but must be queued through a pipe queue. The controls placed on pipe queues add another layer of control and implement a simple request execution class facility.

Transaction Logging

NQS uses the UNIX file system to record request state information. In doing this, NQS uses a technique called the two-phase commit protocol. NQS uses the modification time field of protected and preallocated files to store transaction state information for each request. The transaction state information is updated by setting the modification time of a preallocated file, making a link to the updated inode (to force its writing to disk), followed by an unlink to remove the temporary link that forced the I/O operation. The source code for the transaction logging mechanism is relatively small.

Account Mapping

The network server on the remote machine performs the account mapping. NQS uses one of two forms of account mapping during normal operations. Mapping is performed on all network transactions. This account mapping either uses the client's host and user-id, or the client's host and user name when noting the proper account for network transactions. Network transactions normally include machine-to-machine connections such as the transport of batch and device requests from a local host to a destination machine.

Because the user name or user-id may not be unique in the network, NQS also supports the concept of a machine-id that uniquely identifies each client machine in the network. The machine-id is supported by a mapping program named *nmapmgr*. Appendix A "Reconfiguring NQS" provides reference and user information on the *nmapmgr* program, and on other topics related to reconfiguring the NQS system.

Recording/Reporting NQS Status

NQS uses both NQS mail and an NQS log file to post the status of operations. Two **qmgr** subcommands, **SEt Mail** and **SEt LOg_file** define the paths for these two status reporting mechanisms. NQS supports status reporting on device, request, queue, and limit queries. Limit queries are used to determine the set of batch request resource limits supported by NQS on the local machine. The **qdev**, **qstat**, and **qlimit** commands use these status functions when reporting on previously queued requests and their containing queues.

NQS COMMAND REFERENCE 4

INTRODUCTION

This chapter documents the iPSC®/2 and iPSC®/860 Network Queueing System (NQS) commands. These commands allow you to perform local and remote queueing operations using batch, device, pipe, and network queue structures. The NQS command set was originally developed by the National Aeronautics and Space Administration (NASA) as a networked, UNIX-based queueing system. NQS has been enhanced by the Intel Supercomputer Systems Divisionto provide local and remote queueing support for iPSC/2 and iPSC/860 systems.

The NQS commands are presented in alphabetical order. Where appropriate, the command description text notes any commands or options that are only available to NQS managers. NQS managers may only be established by the UNIX System Administrator.

NQS COMMANDS

The following is a list of NQS commands:

qdel	Deletes requests from the specified queue. You can also send a UNIX signal to any request that has already started to run.
qdev	Displays the status of NQS devices.
qlimit	Displays the supported resource limits and the shell strategy for a specified workstation.
qmgr	Invokes the NQS manager program. This is an interactive program that allows you to define, configure, and manage queues.
qpr	Queues user files for printing.
qstat	Displays queue status.
qsub	Submits requests to queues.

The NQS commands presented in this command reference are arranged alphabetically, and include commands that apply to both NQS operators and to NQS managers. Commands (such as the **qmgr** command) that include a number of subcommands have the subcommands arranged alphabetically under the base command. Commands (such as the **qpr** and **qsub** commands) that include a number of flags have the detailed descriptions of the flags arranged alphabetically under the base command. Each base command starts on a new reference page. The name of the current base command appears in bold type in the upper corners of each page.

QDEL QDEL

Delete or signal NQS request(s).

Synopsis

qdel [**-k**] [**-h** *hostname*] [**-signo**] [**-u** *username*] *request-id* . . .

Description of Parameters

-k	Send the **SIGKILL** signal to the request.
-h *hostname*	Perform the request on the *hostname* host.
-signo	Send a specified signal to the identified running request.
-u *username*	Delete or signal requests owned by *username*.
request-id	The unique identity of a request in the NQS system.

Discussion

qdel deletes all queued NQS requests whose respective *request-id* is listed on the command line. Additionally, if the flag **-k** is specified, then the default signal of **SIGKILL** (-9) is sent to any running request whose *request-id* is listed on the command line. This will cause the receiving request to exit and be deleted. If the flag **-h** *hostname* is requested then the action will be taken on the given host. If the flag **-signo** is present, then the specified signal is sent instead of the **SIGKILL** signal to any running request whose *request-id* is listed on the command line. In the absence of the **-k** and **-signo** flags, **qdel** will not delete a running NQS request.

To delete or signal an NQS request, the invoking user *must* be the owner; namely the submitter of the request. The only exception to this rule occurs when the invoking user is the *superuser*, or has NQS operator privileges as defined in the NQS manager database. Under these conditions, the invoker may specify the **-u** *username* flag which allows the invoker to delete or signal requests owned by the user whose account name is *username*. When this form of the command is used, **all** *request-ids* listed on the command line are presumed to refer to requests owned by the specified user.

An NQS request is always uniquely identified by its *request-id*, no matter where it is in the network of the machines. A *request-id* is always of the form: *seqno* or *seqno.hostname* where *hostname* identifies the machine from whence the request was originally submitted, and *seqno* identifies the sequence number assigned to the request on the originating host. If the *hostname* portion of a *request-id* is omitted, then the local host is always assumed.

QDEL *(cont.)*

QDEL *(cont.)*

The *request-id* of any NQS request is displayed when the request is first submitted (unless the *silent* mode of operation for the given NQS command was specified). The user can also obtain the *request-id* of any request through the use of the **qstat** command.

Caveats

When an NQS request is signalled by the methods discussed above, the proper signal is sent to *all* processes comprising the NQS *request* that are in the same *process group*. Whenever an NQS request is spawned, a new *process group* is established for all processes in the request. However, should one or more processes of the request successfully execute a **setpgrp** () system call, then such processes will not receive any signals sent by the **qdel** command. This can lead to "rogue" request processes which must be killed by other means such as the **kill** command. For the UNIX implementations that support the ability to "lock" a process, and all of its progeny into a *process-group*, NQS will exploit this capability to prevent processes from "escaping" in this manner.

See Also

qdev, qlimit, qmgr, qpr, qstat, qsub, kill, setpgrp(), and **signal**().

QDEV QDEV

Display the status of NQS devices.

Synopsis

qdev [*device-name*] [*device-name@host-name ...*]

Description of Parameters

device-name One form of specifying the device or devices for which the status is being requested. The local host is assumed.

device-name@host-name ...
An alternate way of specifying the device or devices for which the status is being requested. The *host-name* is part of the specification.

Discussion

qdev displays the status of devices known to the Network Queueing System (NQS).

If no devices are specified, then the current state of each NQS device on the local host is displayed. Otherwise, the response is limited to the devices specified. Devices may be specified either as *device-name* or *device-name@host-name*. In the absence of a *host-name* specifier, the local host is assumed.

A device header with several headings is displayed for each of the selected devices. The first heading in a device header appears as Device:, and is followed by the name of the device formatted as *device-name@host-name*. The second heading of Fullname: is followed by the full path name of the special file associated with the device. The third heading of Server: is followed by the command line which will be used to **execve()** the device server. The fourth heading of Forms: is followed by the forms configured for the device.

The final heading of Status: prefaces a display of the general device state. The general state of a device is defined by two principal properties of the device.

The first property concerns whether or not the device is willing to continue accepting queued requests. If it is, the device is said to be ENABLED. If the device is unwilling to continue accepting queued requests, and is idle, its state is DISABLED. A third state of ENABLED/CLOSED is used to describe a device that is unwilling to continue accepting queued requests, but is not yet idle.

QDEV *(cont.)*

QDEV *(cont.)*

The second principal property of a device concerns whether or not the device is busy. There are three cases. If the device is busy, it is said to be ACTIVE. If the device is idle and not known to be out of service, it is said to be INACTIVE. Finally, if the device is idle and known to be out of service, it is said to be FAILED. FAILED covers both hardware and software failures.

If a device is busy, information about the active request follows the device header. The *request-name*, *request-id*, and the name of the user who submitted the request are all displayed.

See Also

qdel, qlimit, qmgr, qpr, qstat, and **qsub**

QLIMIT

QLIMIT

Show supported batch limits, and shell strategy for the named host(s).

Synopsis

qlimit [*host-name ...*]

Description of Parameters

host-name ... The named host or hosts to which this command applies. In the absence of a named host, the local host is assumed.

Discussion

The **qlimit** command displays the set of batch request resource limit types that can be directly enforced on the implied local host or named hosts, and also the batch request shell strategy defined for the implied local host or named hosts.

If no *host-names* are given, then the information displayed is only relevant to the local host. Otherwise, the supported batch request limits, and *batch request shell strategy* for each of the named hosts is displayed.

NQS supports many batch request resource limit types that can be applied to an NQS batch request. However, not all UNIX implementations are capable of supporting the rather extensive set of limit types that NQS provides.

The set of limits applied to a batch request, is always restricted to the set of limits that can be directly supported by the underlying UNIX implementation. If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type.

When an attempt is made to queue a batch request, each *limit-value* specified by the request (that can also be supported by the local UNIX implementation), is compared against the corresponding *limit-value* as configured for the destination batch queue. If the corresponding batch queue *limit-value* for all batch request *limit-values* is defined as unlimited, or is greater than or equal to the corresponding batch request *limit-value*, then the request can be successfully queued, provided that no other anomalous conditions occur. For request infinity *limit-values*, the corresponding queue *limit-value* must also be configured as infinity.

QLIMIT *(cont.)*

QLIMIT *(cont.)*

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the **qsub** command, or by the indirect placement of a batch request into a batch queue via a pipe queue. It is impossible for a batch request to be queued in an NQS batch queue if any of these resource limit checks fail.

Finally, if a request fails to specify a *limit-value* for a resource limit type that is supported on the execution machine, then the corresponding *limit-value* as configured for the destination queue, becomes the *limit-value* for the unspecified request limit.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent **qmgr** commands that alter the limits of the containing batch queue.

As mentioned above, this command also displays the shell strategy as configured for the implied local host, or named hosts. In the absence of a shell specification for a batch request, NQS must choose which shell should be used to execute that batch request. NQS supports three different algorithms, or strategies to solve this problem that can be configured for each system by a system administrator, depending on the needs of the user's involved, and upon system performance criterion.

The three possible shell strategies are called fixed, free, and login.

These shell strategies respectively cause the configured fixed shell to be executed to interpret all batch requests, cause the user's login shell as defined in the password file to be executed which in turn chooses and spawns the appropriate shell for running the batch shell script, or cause only the user's login shell to be executed to interpret the script.

A shell strategy of fixed means that the same shell as chosen by the System Administrator, will be used to execute all batch requests.

A shell strategy of free will run the batch request script exactly as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of fixed; and login exist for host systems that are short on available free processes. In these two strategies, a single shell is executed, and that same shell is the shell that executes all of the commands in the batch request shell script.

When a shell strategy of fixed has been configured for a particular NQS system, then the "fixed" shell that will be used to run all batch requests at that host is displayed.

See Also

qdel, qdev, qmgr, qpr, qstat, and **qsub**

QMGR QMGR

NQS queue manager program.

Synopsis

qmgr

Description

qmgr is a program used by the System Administrator or an NQS operator or manager to control NQS requests, queues, devices, and the general NQS configuration at the local machine.

User Privilege Requirements

All subcommands of the **qmgr** program can be used when the account has NQS manager privileges. A subset of these subcommands are also accessible to accounts that only have NQS operator privileges. Table 4-1 lists the **qmgr** subcommands that are accessible to NQS operators.

Table 4-1. QMGR Functions for NQS Operators

Abort Queue	Show Device
Disable Device	Show Forms
Disable Queue	Show Limits_supported
Enable Device	Show Long Queue
Enable Queue	Show Managers
Exit	Show Parameters
Help	Show Queue
Lock Local_daemon	Shutdown
Purge Queue	Start Queue
Set Run_limit	Stop Queue
Show All	Unlock Local_daemon

Commands

The following paragraphs describe the syntax of each **qmgr** command. All command keywords are recognized regardless of upper or lower case usage. Keyword characters shown in uppercase indicate the smallest possible abbreviation of the keyword for the particular command being described.

QMGR *(cont.)* QMGR *(cont.)*

Abort Queue

ABort Queue *queue* [*seconds*]

queue A named device, batch, or pipe queue.

seconds A real value corresponding to the delay time.

All requests in the named *queue* that are currently running are aborted as follows. A SIGTERM signal is sent to each process of each request presently running in the named *queue*. After the specified number of *seconds* of real time have elapsed, a SIGKILL signal is sent to all remaining processes for each request running in the named *queue*. If a *seconds* value is not specified, then the delay is sixty seconds. All requests aborted by this command are deleted, and all output files associated with the requests are returned to the appropriate destination.

NQS operator privileges are required to use this command.

Add Destination

ADd DEStination = *destination queue*
ADd DEStination = (*destination* [, *destination* . . .]) *queue*

destination The name of a valid destination.

queue A named *pipe* queue.

The specified *destination(s)* are added as valid destinations for a pipe queue named *queue*.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Add Device

ADd DEVice = *device queue*
ADd DEVice = (*device* [, *device* . . .]) *queue*

device The name of a valid device.

queue A named *device* queue.

The specified *device(s)* are added as resources to service requests from *queue*. The *device(s)* must exist (see the **Create Device** description on page 4-16). The *device* is not specified as the fully qualified device pathname, but rather as the NQS shorthand name assigned to the device by the **Create DEVICE** command.

Full NQS manager privileges are required to use this command.

Add Forms

ADd Forms *form-name* . . .

form-name The name of a valid form.

The specified *form-name(s)* are added to the list of valid forms. A *form-name* can contain any printable non-blank character with the exception that a *form-name* cannot start with a decimal digit (`0 . . . 9`), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", "(", and ")".

Full NQS manager privileges are required to use this command.

Add Groups

ADd Groups = *group queue*
ADd Groups = (*group* [, *group* . . .]) *queue*

group The login group name of the requester.

queue A named device, batch, or pipe queue

The specified *group(s)* are added to the access list for *queue*. It is not permitted to add a group when queue access is unrestricted. There are two ways to specify a group:

group name
[*group id*]

For example, if there is a group archers on your machine with group-id 10001, you may specify it with archers or [10001]. In order for a user to have access to a queue, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited. (See the **Set Unrestricted_access** description on page 4-52.)
2. The user's login group has access.
3. The user's account has access. (See the **Add Users** description on page 4-14.)

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* **QMGR** *(cont.)*

Add Managers

ADd Managers *manager* . . .

manager The valid account name of an NQS manager.

The specified *manager(s)* are added to the list of authorized NQS managers with privileges as specified. A *manager* specification consists of an account name specification, followed by a colon, followed by either the letter *m* or the letter *o*. There are four ways to specify an account name:

local_account_name
[*local_user_id*]
[*remote_user_id*]@*remote_machine_name*
[*remote_user_id*]@[*remote_machine_mid*]

If the account name specification is followed by *:m*, then the account is designated as an NQS manager account, capable of using all **qmgr** commands. If the account name specification is followed by *:o*, then the account is designated as an NQS operator account, capable of only using those commands appropriate for an NQS operator. Each of the following four examples demonstrates the use of the above four parse constructs for an NQS manager account name.

```
1. kingsbur            ; Local account of "kingsbur"
2. [149]             ; Local account (user-id = 149)
3. [149]@Hal9000   ; Remote account (user-id = 149)
                     ; on remote machine: "Hal9000"
4. [149]@[9000]    ; Remote account (user-id = 149)
                  ; on remote machine with machine-
                                       ; id = 9000.
```

Full NQS manager privileges are required to use this command.

Add Users

ADd Users = *user queue*
ADd Users = (*user* [, *user* . . .]) *queue*

user The valid name of an NQS user.

queue A named device, batch, or pipe queue.

The specified *user(s)* are added to the access list for *queue*. Users cannot be added to a *queue* when queue access is unrestricted. There are two ways to specify a user:

user name
[user id]

For example, if there is a user robin on your machine with *user-id* 1001, you may specify the user with robin or [1001]. In order for a user to have access to a queue, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited. (See the **Set Unrestricted_access** description on page 4-52.)
2. The user's login group has access. (See the **Add Groups** description on page 4-12.)
3. The user's account has access.

Full NQS manager privileges are required to use this command.

Create Batch_queue

Create Batch_queue *queue* PRiority = *n* [PIpeonly] [Run_limit = *n*]

queue A named batch queue.

n An integer. See text.

Define a batch queue named *queue* with inter-queue priority *n* (0..63). A **PRiority** of 0 is the highest priority. A *queue* name can contain any printable non-blank character with the exception that a *queue* name cannot start with a decimal digit (`0 ... 9`), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")".

If **PIpeonly** is specified, then requests may enter this *queue* only if their source is a pipe queue. The specification of a *Run_limit* sets a ceiling on the maximum number of requests allowed to run in the batch queue at any given time. The default run-limit is one. (See the "Queue Types" section on page 4-58 for more information.)

The queue is created in a disabled and stopped state, and must be explicitly enabled and started by the **ENable Queue** and **STArt Queue** commands.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Create Device

Create DEVICE *device* FOrms = *forms* FUllname = *filename* Server = (*server*)

device A valid *device* name.

forms A valid *forms* type defined for this device.

filename The absolute path name of this device. This name is only used by NQS to perform the appropriate device open operations when necessary. At all other times, the device is always referred to by the *device* parameter.

server The fully qualified pathname of the server program associated with this device, and any relevant invocation arguments.

Define a *device* with the specified *forms* and associate it with a *server*. This is done by specifying an absolute path name to the program binary (*server*) and any arguments required by the program. The *filename* parameter is the absolute path name of the device (special file) and is typically */dev/device*. The *device* name can contain any printable non-blank character except it cannot contain the @ character or start with a decimal digit (`0 ... 9`), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")". The device is created in disabled mode as must be explicitly enabled via the **ENable Device** command.

Full NQS manager privileges are required to use this command.

Create Device_queue

Create DEVICE_queue *queue* **PRiority** = *n* [**Device** = *device*]
[**Device** = (*device* [, *device* . . .])]
[**PIpeonly**]

queue A named device *queue*.

n An integer. See text.

device A valid *device* name.

Define a device queue named *queue* with inter-queue priority *n* (0..63). The *queue* name can contain any printable non-blank character except it cannot contain the @ character or start with a decimal digit (0 . . . 9), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")".

If **PIpeonly** is specified, then requests may enter this *queue* only if their source is a pipe queue.

After **Device** appears a list of one or more *devices* that may service this *queue*. (See the "Queue Types" section on page 4-58 for more information.) All devices named in the set must have been previously defined by the **Create DEVICE** command.

The queue is created in a disabled and stopped state, and must be explicitly enabled and started by the **ENable Queue** and **STArt Queue** commands.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Create Pipe_queue

Create Pipe_queue *queue* **PRiority** = *n* **Server** = (*server*)
[**Destination** = *destination*]
[**Destination** = (*destination* [, *destination* . . .])]
[**PIpeonly**] \ [**Run_limit** = *n*]

queue A named pipe *queue*.

n An integer. See text.

server The name of the *server* program associated with this pipe queue.

destination The name of a valid *destination* queue.

Define a pipe queue named *queue* with inter-queue priority *n* (0..63) and associate it with a *server*. This is done by specifying an absolute path name to the program binary (*server*) and any arguments required by the program. The *queue* name can contain any printable non-blank character except it cannot contain the @ character or start with a decimal digit (`0 . . . 9`), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")".

After **Destination** appears a list of one or more *destination* queues that requests from this pipe *queue* may be sent to. The syntax of a pipe queue destination (*destination*) if specified, must conform to one of the following four forms:

local_queue_name
local_queue_name@local_machine_name
remote_queue_name@remote_machine_name
remote_queue_name@[remote_machine_mid]

where the *local_machine_name* or *remote_machine_name* must be defined in the network host table of the local system. The destination syntax form:

remote_queue_name@[remote_machine_mid]

allows for the explicit specification of the destination machine by its machine-id, entered as in integer:

```
example_queue_name@[123]
```

QMGR *(cont.)*

QMGR *(cont.)*

If **PIpeline** is specified, then requests may enter this *queue* only if their source is a pipe queue. **Run_limit** sets a ceiling on the maximum number of requests allowed to run in the pipe queue at any given time. The default run-limit is one. (See the "Queue Types" section on page 4-58 for more information.) The queue is created in a disabled and stopped state, and must be explicitly enabled and started by the **ENable Queue** and **STArt Queue** commands.

Full NQS manager privileges are required to use this command.

Delete Destination

DElete DEStination = *destination queue*
DElete DEStination = (*destination* [, *destination* . . .]) *queue*

queue A named pipe *queue*.

destination The name of a valid *destination* queue.

Delete the mappings from the pipe *queue* to the *destination* queue(s). All requests from the named *queue* being transferred to a deleted *destination* complete normally. If all *destinations* for a pipe *queue* are deleted in this manner, then the pipe *queue* is effectively stopped, though its actual status will remain unchanged. Thus, the addition of a new destination for a pipe queue that has been effectively stopped in the manner described, will immediately cause the queue to start running again.

The syntax of a pipe queue destination (*destination*) if specified, must conform to one of the following four forms:

local_queue_name
local_queue_name@local_machine_name
remote_queue_name@remote_machine_name
remote_queue_name@[remote_machine_mid]

where the *local_machine_name* or *remote_machine_name* must be defined in the network host table of the local system. The destination syntax form:

remote_queue_name@[remote_machine_mid]

allows for the explicit specification of the destination machine by its machine-id, entered as in integer:

```
example_queue_name@[123]
```

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Delete Device

DElete DEVice *device*

device A valid *device* name.

Delete the specified *device*. A device must first be disabled to delete it from the device set (see the **Disable Device** description on page 4-25). Similarly, the device cannot be servicing an NQS request if it is to be deleted. Upon deletion, the device is removed from all queue device mapping sets.

Full NQS manager privileges are required to use this command.

Delete Device

DElete DEVice = *device queue*
DElete DEVice = (*device* [, *device* . . .]) *queue*

device A valid *device* name.

queue A named device *queue*.

Delete the mappings from the device *queue* to the *device(s)*. All requests from the named device *queue* running on any of the named *devices* are allowed to complete normally. If ALL queue-to-device mappings for the named device *queue* are removed by this command, then the *queue* is effectively stopped, since no devices are accessible to handle requests for the affected queue.

The actual queue status remains unchanged for a *device queue* with all devices deleted, so that the first addition of an enabled device to the device set for a *queue* that was effectively stopped in the manner described, immediately causes the *queue* to start running again.

Full NQS manager privileges are required to use this command.

Delete Forms

DElete Forms *form-name . . .*

form-name A valid form name.

The specified *form-name(s)* are deleted from the list of valid forms.

Full NQS manager privileges are required to use this command.

Delete Groups

DElete Groups = *group queue*
DElete Groups = (*group* \[, *group . . .*]) *queue*

group The login *group* name of the requester.

queue A named device, batch, or pipe *queue*

The specified *group(s)* are deleted from the access list for *queue*. This command can only delete groups that have been added with the **ADd Groups** command. It is not permitted to delete a group from a queue where access is unrestricted. There are two ways to specify a group:

group name
[group id]

For example, if there is a group archers on your machine with group-id 10001, you may specify it with archers or [10001]. If the *group* name does not appear in the system group file, the latter form must be used. In order for a user to have access to a *queue*, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited. (See the **Set No_Access** description on page 4-40.)
2. The user's login group has access.
3. The user's account has access. (See the **Delete Users** description on page 4-24.)

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* **QMGR** *(cont.)*

Delete Managers

DElete Managers *manager* . . .

manager A valid *manager* account name specification.

The specified *manager(s)* are deleted from the access list of authorized NQS managers. Attempts to exclude the system root account from the NQS manager set will be silently ignored. A *manager* specification consists of an account name specification, followed by a colon, followed by either the letter *m* or the letter *o*. There are four ways to specify an account name:

local_account_name
[*local_user_id*]
[*remote_user_id*]@*remote_machine_name*
[*remote_user_id*]@[*remote_machine_mid*]

Each of the following four examples demonstrates the use of the above four parse constructs for an NQS *manager* name.

```
kingsbur                ; Local account of "kingsbur"
[149]                  ; Local account (user-id = 149)
[149]@Hal9000         ; Remote account (user-id = 149)
                        ; on remote machine: "Hal9000"
[149]@[9000]          ; Remote account (user-id = 149)
                      ; on remote machine with machine
                                        ; id = 9000.
```

If the account name specification is followed by *:m*, it is understood that the account is currently permitted to use all **qmgr** commands. If the account name specification is followed by *:o*, it is understood that the account is currently permitted to use only those commands appropriate for an operator to use. The root account always has full privileges.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Delete Queue

DElete Queue *queue*

queue A named device, batch, or pipe *queue*

The *queue* is deleted. To delete a *queue*, no requests may be present in the *queue* and the *queue* MUST be disabled (see the **Disable Queue** description on page 4-25). Any queue-to-device mappings are updated accordingly.

Full NQS manager privileges are required to use this command.

Delete Users

DElete Users = *user queue*
DElete Users = (*user* [, *user* . . .]) *queue*

user The valid name of an NQS *user*.

queue A named device, batch, or pipe *queue*

The specified *user(s)* are deleted from the access list for *queue*. This command can only delete users who have been added with the **ADd Users** command. It is not permitted to delete a user from a queue where access is unrestricted. There are two ways to specify a user:

user name
[*user id*]

For example, if there is a user robin on your machine with user-id 1001, you may specify the user with robin or [1001]. If the group name does not appear in the system group file, the latter form must be used. In order for a user to have access to a queue, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited. (See the **Set No_Access** description on page 4-40.)
2. The user's login group has access. (See the **Delete Groups** description on page 4-22.)
3. The user's account has access.

Full NQS manager privileges are required to use this command.

Disable Device

DIsable Device *device*

device A valid *device* name.

The current request will complete. After that, the *device* is prevented from handling any more requests until it is enabled (see the **Enable Device** description on page 4-25). If the disabled *device* was the last enabled device in a queue-to-device mapping, then the device queue is effectively stopped, since no more enabled devices appear in the device set for the particular queue. However, the actual queue status remains unchanged for such a queue, so that the first addition of an enabled device to the device set for a queue that was effectively stopped in the manner described, immediately causes the queue to start running again.

NQS operator privileges are required to use this command.

Disable Queue

DIsable Queue *queue*

queue A named device, batch, or pipe *queue*.

Prevent any more requests from being placed in this *queue*.

NQS operator privileges are required to use this command.

Enable Device

ENable Device *device*

device A valid *device* name.

If the *device* is already enabled, then this is a no-op. Otherwise, the *device* becomes available to handle requests.

NQS operator privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Enable Queue

ENable Queue *queue*

queue A named device, batch, or pipe *queue*

If the *queue* is already enabled, then this is a no-op. Otherwise, the *queue* is enabled to accept new requests.

NQS operator privileges are required to use this command.

Exit

EXit

Exit from the NQS manager subsystem (the **qmgr** command). The NQS manager program is also exited if an end-of-file is encountered on the standard input file. Thus, typing ^D or **EXit** will cause the NQS manager program to exit.

Help

Help [*command*]

command A valid NQS *command* specification.

Get help information. **Help** without an argument displays information about what commands are available. **Help** with an argument displays information about that command. The command may be partially specified as long as it is unique. A more complete help request yields more detailed information.

The **Help** *command* provides information that is often more extensive than the command descriptions in this manual page! Use it.

Lock Local_daemon

Lock Local_daemon

Lock the NQS local daemon into memory. See the **plock(2)** description in the UNIX documentation.

NQS operator privileges are required to use this command.

Purge Queue

Purge Queue *queue*

queue A named device, batch, or pipe *queue*.

All queued requests are purged (dropped) from the *queue* and are irretrievably lost. Running requests in the *queue* are allowed to complete.

NQS operator privileges are required to use this command.

Set Corefile_limit

SEt CORefile_limit = (*limit*) *queue*

limit The maximum value supported for this process type. The *limit* is specified as a single integer less than 100,000,000 with an optional fractional part.

queue A named batch *queue*.

Set a per-process maximum core file size *limit* for a batch *queue* against which the per-process maximum core file size limit for a request may be compared. If the local host does not support per-process core file size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum core file size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process core file size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit.

If core file size limits are not supported by the host UNIX operating system, then an appropriate error diagnostic is displayed, upon the use of this command. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Data_limit

SEt DAta_limit = (*limit*) *queue*

limit The maximum value supported for this process type. The *limit* is specified as a single integer less than 100,000,000 with an optional fractional part.

queue A named batch *queue*.

Set a per-process maximum data segment size *limit* for a batch *queue* against which the per-process maximum data segment size limit for a request may be compared. If the local host does not support per-process data segment size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum data segment size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum data segment size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit.

Depending on the underlying UNIX implementation upon which NQS is running, this limit may, or may not be configurable. If data-segment size limits are not

supported by the host UNIX operating system, then an appropriate error diagnostic is displayed, upon the use of this command. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Debug

SEt DEBug *level*

level A valid debug *level* number.

Set the debug *level*. The following values are valid:

0	No debug
1	Minimum debug
2	Maximum debug

Full NQS manager privileges are required to use this command.

Set Default Batch_request Priority

SEt DEFault Batch_request Priority *priority*

priority A value defining batch queue processing priority.

Set the default intra-queue *priority* for batch queues. This is NOT the UNIX execution time priority. This is the priority used if the user does not specify an intra-queue priority parameter on the **qsub** command.

Full NQS manager privileges are required to use this command.

Set Default Batch_request Queue

SEt DEFault Batch_request Queue *queue*

queue A named batch queue.

Set the default batch *queue*. This is the queue used if the user does not specify a queue parameter on the **qsub** command.

Full NQS manager privileges are required to use this command.

Set Default Destination_retry Time

SEt DEFault DEStination_retry Time *retry_time*

retry_time A number of hours.

Set the default number of hours that can elapse during which time a pipe queue destination can be unreachable before being marked as completely failed.

Full NQS manager privileges are required to use this command.

Set Default Destination_retry Wait

SEt DEFault DEStination_retry Wait *interval*

interval A number of minutes.

Set the default number of minutes to wait before retrying a pipe queue destination that was unreachable at the time of the last attempt. When a pipe queue destination fails to accept a request because of a network failure, or remote server failure, then the request is not transferred, and the destination is disabled for the number of minutes as determined by the *interval* parameter. At the end of this time, the destination is re-enabled, and retried as appropriate. To prevent an infinite number of retries, the **SEt DEFault DEStination_retry Time** command is used to prevent a pipe queue destination from being endlessly retried forever.

Full NQS manager privileges are required to use this command.

Set Default Device_request Priority

SEt DEFault DEVice_request Priority *priority*

priority An integer (0..63) corresponding to device request *priority* (63 = highest).

Set the default intra-device-queue *priority*. The priority selected is termed the intra-queue priority because the selected priority determines the relative ordering of requests within the selected device queue (not the execution-time priority) of the request. This is the priority used if the user does not specify an intra-queue priority parameter with the **qpr** command.

Full NQS manager privileges are required to use this command.

Set Default Print_request Forms

SEt DEFault Print_request Forms *form-name*

form-name The valid name of a forms that is to be used as default.

Set the default print forms to *form-name*. This is the forms used if the user does not specify a forms parameter on the **qpr** command.

Full NQS manager privileges are required to use this command.

Set Default Print_request Queue

SEt DEFault Print_request Queue *queue*

queue A named print queue.

Set the default print *queue*. This is the queue used if the user does not specify a queue parameter on the **qpr** command.

Full NQS manager privileges are required to use this command.

Set Destination

SEt DEStination = *destination queue*
SEt DEStination = (*destination* [, *destination* . . .]) *queue*

destination A valid *destination* for a pipe queue.

queue A named pipe *queue*.

Associate one or more *destination* queues with a particular pipe queue. All previous destinations in the pipe queue destination set are discarded. All machine-names appearing in any queue destination names must be defined in the local system's network host table.The syntax of a pipe queue destination (*destination*) must conform to one of the following four forms:

local_queue_name
local_queue_name@local_machine_name
remote_queue_name@remote_machine_name
remote_queue_name@[remote_machine_mid]

where the *local_machine_name* or *remote_machine_name* must be defined in the network host table of the local system. The destination syntax form:

remote_queue_name@[remote_machine_mid]

allows for the explicit specification of the destination machine by its machine-id, entered as an integer:

```
example_queue_name@[123]
```

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Set Device

SEt DEVICE = *device queue*
SEt DEVICE = (*device* [, *device* . . .]) *queue*

device A valid *device* name.

queue A named device queue.

Associate one or more *devices* with a device set for a particular *queue*. Use of the **SEt DEVICE** command causes all previous *devices* in the device set for the named *queue* to be discarded. The device set for the named *queue* is then set to include only the specified *devices*.

Full NQS manager privileges are required to use this command.

Set Device_server

SEt DEVICE_server = (*server*) *device*

server The pathname to a valid *server*.

device A valid *device* name.

Associate a *server* with a *device*. *Server* should consist of the absolute path name to the program binary followed by any arguments required by the program. The text within the enclosing parentheses defines the *server* command line that will be used to execute requests handled by the named *device*.

Full NQS manager privileges are required to use this command.

Set Forms

SEt Forms *form-name . . .*

form-name The valid name of a forms.

Specify the valid *form-name(s)* for the device forms list. Other valid forms may be added to this list (see the **Add Forms** description on page 4-11). The *form-name* can contain any printable non-blank character except it cannot contain the @ character or start with a decimal digit (0 . . . 9), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")".

Full NQS manager privileges are required to use this command.

Set Forms

SEt Forms = *form-name device*

form-name The valid name of a *forms.*

device A valid *device* name.

Set the *form-name* for a *device.* The *form-name* can contain any printable non-blank character except it cannot contain the @ character or start with a decimal digit (0 . . . 9), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")".

NQS operator privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Set Lifetime

SEt LIfetime *lifetime*

lifetime A number of hours.

Set the *pipe queue* request *lifetime* in hours. This value determines the number of hours that any request can reside within a *pipe queue*, from the time at which the request was first placed within the queue. If any request resides within a *pipe queue* for a duration of time equal to or greater than this limit (for example, if the pipe queue is stopped), then the request is deleted, and mail is sent informing the owner of the request that the request has been deleted.

The *lifetime* parameter exists to prevent requests from filling up all available queue space because of network failures, or other causes that would prevent requests from being delivered to their respective destinations. This value should be typically configured to describe a fairly long time interval (such as 72 hours). Alternatively, the *lifetime* parameter can be assigned a value of 0, which indicates that all requests residing within a pipe queue have an infinite lifetime.

Full NQS manager privileges are required to use this command.

Set Log_file

SEt LOg_file *filename*

filename A valid file name.

Specify the name of a new log file for NQS messages. If the specified file is successfully opened for append, then NQS writes a message on the old logfile stating that a switch is being made to the new logfile. Otherwise, an error message is written on the current log file and the current logfile is not changed.

Full NQS manager privileges are required to use this command.

Set Mail

SEt MAIl *userid*

userid The *userid* of the user that is to receive NQS mail messages.

Specify the *userid* used to send NQS mail. This command is used to set the account name that will appear in the From: portion of mail sent by NQS to notify users (when appropriate), of certain events concerning their NQS request(s).

The mail *account name* can contain any printable non-blank character except it cannot contain the @ character or start with a decimal digit (`0 ... 9`), and it cannot contain any of the **qmgr** command punctuation characters of ",", "=", " (", and ")".

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* **QMGR** *(cont.)*

Set Managers

SEt MANagers *manager* . . . :{m,o}

manager The name of a valid *user* that has an NQS account.

The list of authorized NQS managers is set to the specified *manager(s)*. A *manager* specification consists of an account name specification, followed by a colon, followed by either the letter *m* or the letter *o*. There are four ways to specify an account name:

local_account_name
[*local_user_id*]
[*remote_user_id*]@*remote_machine_name*
[*remote_user_id*]@[*remote_machine_mid*]

If the account name specification is followed by **:m**, then the account is designated as an NQS *manager* account, capable of using all **qmgr** commands. If the account name specification is followed by **:o**, then the account is designated as an NQS *operator* account, capable of only using those commands appropriate for an NQS operator. The *root* account always has full privileges. Each of the following four examples demonstrates the use of the above four parse constructs for an NQS manager.

```
kingsbur              ; Local account of "kingsbur"
[149]                ; Local account (user-id = 149)
[149]@Hal9000       ; Remote account (user-id = 149)
                      ; on remote machine: "Hal9000"
[149]@[9000]        ; Remote account (user-id = 149)
                    ; on remote machine with machine
                                        ; id = 9000.
```

An **:m** or **:o** suffix at the end of the *manager* name specified for this command is REQUIRED. This suffix indicates the NQS **qmgr** privileges to be assigned to the corresponding account. An **:m** suffix indicates that the account should be granted full NQS *manager* privileges. The owner of such an account will be able to use all of the NQS **qmgr** commands. An **:o** suffix indicates that the account will only be granted NQS operator privileges. Such an account will have the privilege to execute many **qmgr** commands, but will NOT have the privilege to create new queues, devices, forms, managers, or to delete any of the same.

Attempts to exclude the system root account from the NQS manager set as a fully privileged manager account (privilege=**m**), will be silently ignored. Also see the **Add Managers** description on page 4-13.

Full NQS manager privileges are required to use this command.

Set Maximum Copies

SEt MAXimum Copies *copies*

copies An integer value.

Set the maximum number of print *copies*. NQS will refuse to print more than this many copies of a print file.

Full NQS manager privileges are required to use this command.

Set Maximum Open_retries

SEt MAXimum Open_retries *retries*

retries An integer value.

Specify the maximum number of *retries* that NQS will try to open a device prior to spawning a device server to handle an NQS device request. If an error occurs trying to open a device, then this limit prevents NQS from trying forever to open a failed device. If this limit is exceeded trying to open a device, then the device is marked as failed.

Full NQS manager privileges are required to use this command.

Set Maximum Print_size

SEt MAXimum Print_size *size*

size A byte value corresponding to maximum print file size.

Specify the maximum *size* of an NQS print file in bytes. NQS will refuse to print files larger than this size.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Set Network Client

SEt NEtwork Client = (*client*)

client The valid pathname to the *client* associated with a pipe queue. The text within the enclosing parentheses defines the command line that will be used to execute the client.

Specify the network client to be used. *Client* should consist of the absolute path name of the client followed by any arguments required by the client. The network *client* stages out batch request output files and empties network queues.

Full NQS manager privileges are required to use this command.

Set Network Daemon

SEt NEtwork Daemon = (*daemon*)

daemon The valid pathname to the *daemon* associated with the network. The text within the enclosing parentheses defines the command line that will be used to execute the daemon.

Specify the network daemon to be used. The network daemon listens for messages from remote clients. *Daemon* should consist of the absolute path name of the daemon followed by any arguments required by the daemon.

Full NQS manager privileges are required to use this command.

Set Network Server

SEt NEtwork Server = (*server*)

server The valid pathname to the *server* associated with the network. The text within the enclosing parentheses defines the command line that will be used to execute the server.

Specify the network server to be used. The network server processes submission requests from remote pipe clients. *Server* should consist of the absolute path name of the server followed by any arguments required by the server.

Full NQS manager privileges are required to use this command.

Set Nice_value_limit

SEt NIce_value_limit = *nice-value queue*

nice-value An integer preceded by an optional negative sign.

queue A named *batch* queue.

Set the UNIX *nice-value* limit for a batch *queue*, against which the nice-value for a request may be compared. If a request already in the queue has asked for treatment more favorable than the new *nice-value*, then it will be given a grandfather clause.

A request specifying a *nice-value* may only enter a batch queue if the queue's nice value is numerically less than (more willing to allow access to the CPU) or equal to the request's nice value. Requests with numerically higher execution priorities than the *nice-value* are rejected.

The *nice-value* assigned to a particular batch request is determined at the time that the request is queued. If the *nice-value* limit for the batch queue is later raised (for example, the new *nice-value* is numerically greater than the old *nice-value*), no requests in the queue will be affected. However, if the new *nice-value* limit is numerically greater than the *nice-value* requested by a previously queued request, then a warning diagnostic is displayed.

Unless a process of the request is running with an effective user-id of root, it will be unable to lower its nice value (raise its execution priority) from the value determined when the request was queued.

Nice-value is an integer preceded by an optional negative sign. It must be remembered that *nice-values* in the range [-20 to-1] actually specify a HIGHER execution priority than the default of 0, while values in the range [1 to 19] ([1 to 20] on Berkeley systems), specify a LOWER execution priority than the default value. On System V based UNIX implementations, a specification of 20 is silently interpreted as 19, since only Berkeley UNIX implementations are capable of using the highest value of 20.

Full NQS manager privileges are required to use this command.

Set No_Access

SEt NO_Access *queue*

queue A named device, batch, or pipe queue.

Specify that no one will be allowed to place requests in *queue*. Requests in the queue will be allowed to remain there. In order to enforce access restrictions on a queue where access is currently unrestricted, start with this command and then add groups or users as needed. Root is an exception; requests submitted by root are always allowed into a queue, even if root is not explicitly given access. If you wish to deny access to root, see the **Disable Queue** description on page 4-25.

In order for a user to have access to a queue, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited.
2. The user's login group has access. (See the **Delete Groups** description on page 4-22.)
3. The user's account has access. (See the **Delete Users** description on page 4-24.)

Full NQS manager privileges are required to use this command.

Set No_Default Batch_request Queue

SEt NO_Default Batch_request Queue

Indicate that there is to be no default batch request queue.

Full NQS manager privileges are required to use this command.

Set No_Default Print_request Forms

SEt NO_Default Print_request Forms

Indicate that there are no default print request forms defined for the local NQS system. Print requests executed on the local system that do not explicitly define any required forms, will run on the first available device for the device queue in which they are entered.

Full NQS manager privileges are required to use this command.

Set No_Default Print_request Queue

SEt NO_Default Print_request Queue

Indicate that there is to be no default print request queue.

Full NQS manager privileges are required to use this command.

Set No_Network_daemon

SEt NO_Network_daemon

Indicate that there is to be no network daemon.

Full NQS manager privileges are required to use this command.

Set Open_wait

SEt Open_wait *interval*

interval A number of seconds.

Specify the number of seconds to wait between failed device open attempts.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)*

QMGR *(cont.)*

Set Per_Process Cpu_limit

SEt PER_Process Cpu_limit = (*limit*) *queue*

limit — A value indicating *limit* time.

queue — A named batch queue.

Set a per-process maximum CPU time *limit* for a batch *queue* against which the per-process maximum CPU time limit for a request may be compared. If the local host does not support per-process CPU time limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum CPU time limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum CPU time limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Per_Process Memory_limit

SEt PER_Process Memory_limit = (*limit*) *queue*

limit — A value indicating *limit* memory size.

queue — A named *batch* queue.

Set a per-process maximum memory size *limit* for a batch *queue* against which the per-process maximum memory size limit for a request may be compared. If the local host does not support per-process memory size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum memory size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. However, if the new memory usage limit value is less than the memory usage limit requested by a previously queued request, then a warning diagnostic is displayed.

A request specifying a per-process maximum memory size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. The memory consumed by ANY process comprising the running batch request cannot exceed the PER-PROCESS maximum determined when the request was queued. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Per_Process Permfile_limit

SEt PER_Process Permfile_limit = (*limit*) *queue*

limit A value indicating *limit* file size.

queue A named batch queue.

Set a per-process maximum permanent file size *limit* for a batch *queue* against which the per-process maximum permanent file size limit for a request may be compared. If the local host does not support per-process permanent file size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum permanent file size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. However, if the new permanent-file size limit value is less than the permanent-file size usage limit requested by a previously queued request, then a warning diagnostic is displayed.

A request specifying a per-process maximum permanent file size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. The size of any permanent-file created by ANY process comprising the running batch request cannot exceed the PER-PROCESS maximum permanent file size as determined when the request was queued. For the syntax of *limit*, see the "Limits" section on page 4-59.

For iPSC systems, there is a lower limit for file sizes. You cannot specify a limit size less than 2048 bytes (2 kilobytes).

Full NQS manager privileges are required to use this command.

Set Per_Process Tempfile_limit

SEt PER_Process Tempfile_limit = (*limit*) *queue*

limit A value indicating *limit* file size.

queue A named batch queue.

Set a per-process maximum temporary file size *limit* for a batch *queue* against which the per-process maximum temporary file size limit for a request may be compared. If the local host does not support per-process temporary file size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum temporary file size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum temporary file size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a batch request which asks for a larger temporary-file size limit than the destination batch queue has defined as allowable, then the request is rejected.

The temporary-file size limit assigned to a particular batch request is determined at the time that the request is queued. If the temporary-file size limit for the containing batch queue is later reduced, no requests in the queue will be affected. However, if the new temporary-file size limit value is less than the temporary-file size usage limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Per_Request Cpu_limit

SEt PER_Request Cpu_limit = (*limit*) *queue*

limit A value indicating *limit* time.

queue A named *batch* queue.

Set a per-request maximum CPU time *limit* for a batch *queue* against which the per-request maximum CPU time limit for a request may be compared. If the local host does not support per-request CPU time limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum CPU time limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum CPU time limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a batch request which asks for more CPU-time than the destination batch queue has defined as allowable, then the request is rejected.

The maximum CPU-time to be given to a particular batch request at execution time is determined at the time that the request is queued. If the CPU-time limit for the containing batch queue is later reduced, no requests in the queue will be affected. However, if the new CPU-time limit value is less than the CPU-time limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)*

QMGR *(cont.)*

Set Per_Request Memory_limit

SEt PER_Request Memory_limit = (*limit*) *queue*

limit A value indicating *limit* memory size.

queue A named batch queue.

Set a per-request maximum memory size *limit* for a batch *queue* against which the per-request maximum memory size limit for a request may be compared. If the local host does not support per-request memory size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum memory size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum memory size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a batch request which asks for a larger maximum memory usage limit than the destination batch queue has defined as allowable, then the request is rejected.

The maximum memory usage limit assigned to a particular batch request is determined at the time that the request is queued. If the memory usage limit for the containing batch queue is later reduced, no requests in the queue will be affected. However, if the new memory usage limit value is less than the memory usage limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Per_Request Permfile_limit

SEt PER_Request Permfile_limit = (*limit*) *queue*

limit A value indicating *limit* file space.

queue A named batch queue.

Set a per-request maximum permanent file space *limit* for a batch *queue* against which the per-request maximum permanent file space limit for a request may be compared. If the local host does not support per-request permanent file space limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum permanent file space limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum permanent file space limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a batch request which asks for a larger permanent-file space usage limit than the destination batch queue has defined as allowable, then the request is rejected.

The permanent-file space usage limit assigned to a particular batch request is determined at the time that the request is queued. If the permanent-file space usage limit for the containing batch queue is later reduced, no requests in the queue will be affected. However, if the new permanent-file space usage limit value is less than the permanent-file space usage limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS *manager* privileges are required to use this command.

Set Per_Request Tempfile_limit

SEt PER_Request Tempfile_limit = (*limit*) *queue*

limit A value indicating *limit* file space.

queue A named *batch* queue.

Set a per-request maximum temporary file space *limit* for a batch *queue* against which the per-request maximum temporary file space limit for a request may be compared. If the local host does not support per-request temporary file space limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum temporary file space limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum temporary file space limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a batch request which asks for a larger temporary-file size limit than the destination batch queue has defined as allowable, then the request is rejected.

The temporary-file size limit assigned to a particular batch request is determined at the time that the request is queued. If the temporary-file size limit for the containing batch queue is later reduced, no requests in the queue will be affected. However, if the new temporary-file size limit value is less than the temporary-file size usage limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Set Pipe_client

SEt PIpe_client = (*client*) *queue*

client The valid pathname to the *client* associated with a pipe queue.

queue A named pipe queue. The text placed within the parentheses defines the fully qualified pathname of the client, and client arguments that are to be used as the command line for a pipe queue client invocation by NQS.

Associate a pipe *client* with a pipe *queue*. *Client* should consist of the absolute path name to the program binary followed by any arguments required by the program. The named queue cannot refer to a batch or device queue.

Full NQS manager privileges are required to use this command.

Set Priority

SEt PRiority = *priority queue*

priority An integer (0..63) corresponding to queue *priority* (63 = highest).

queue A named device, batch, or pipe queue. The queue named as a parameter of the command must already exist.

Specify the inter-queue *priority* of a *queue*.

Full NQS manager privileges are required to use this command.

Set Run_limit

SEt Run_limit = *run-limit queue*

run-limit Maximum number of requests that can run at once.

queue A named batch or pipe queue. The queue named as a parameter of the command must already exist.

Change the *run-limit* of an NQS batch or pipe *queue*. The *run-limit* determines the maximum number of requests that will be allowed to run in the queue at any given time. A *run-limit* cannot be applied to a device queue.

NQS operator privileges are required to use this command.

Set Shell_strategy Fixed

SEt SHell_strategy FIxed = (*shell*)

shell The valid pathname to a command interpreter.

Specify that *shell* should be used to execute all batch requests. *Shell* must be the absolute path name of a command interpreter. This command exists for NQS installations that use only one type of shell (e.g. Bourne, C-shell, etc.).

Full NQS manager privileges are required to use this command.

Set Shell_strategy Free

SEt SHell_strategy FRee

Specify that the **FRee** shell strategy should be used to execute all batch requests. The free shell strategy aims at duplicating the shell choice that would have been made if the batch request script had been executed interactively. This will result in an extra shell being spawned for all batch request executions, EXACTLY as would happen if the user were running their batch request script interactively.

Under this strategy, the user's *login shell* is allowed to determine the shell to be used to execute the batch request. The user's login shell is the shell named within the user's entry in the password file (see the **passwd**(4) description in the UNIX documentation).

Full NQS manager privileges are required to use this command.

Set Shell_strategy Login

SEt SHell_strategy Login

Specify that the **Login** shell strategy should be used to execute all batch requests. Under the login shell strategy, the user's login shell is used to execute the batch request. The *login shell* is the shell named in the password file (see the **passwd**(4) description in the UNIX documentation). This means that the user's login shell will be spawned to run ALL batch request scripts, without examining the batch shell script to determine which type of script it is.

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Set Stack_limit

SEt STack_limit = (*limit*) *queue*

limit A value indicating *limit* size.

queue A named batch queue.

Set a per-process maximum stack segment size *limit* for a batch *queue* against which the per-process maximum stack segment size limit for a request may be compared. If the local host does not support per-process stack segment size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum stack segment size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum stack segment size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a request which asks for a larger stack-segment size limit than the destination

queue has defined as allowable, then the request is rejected. Stack-segment limits cannot be applied to pipe queues. Any attempt to set a stack-segment limit for a pipe queue will fail, with an appropriate error diagnostic being displayed.

The maximum stack-segment size limit assigned to a particular request is determined at the time that the request is queued. If the stack-segment size limit for the containing queue is later reduced, no requests in the queue will be affected. However, if the new stack-segment size limit value is less than the stack-segment size limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Set Unrestricted_access

SEt UNrestricted_access *queue*

queue A named device, batch, or pipe queue.

Specify that no requests will be turned away from *queue* on the grounds of queue access restrictions. In order to enforce access restrictions on a queue where access is currently unrestricted, see the **Set No_Access** description on page 4-40.

In order for a user to have access to a queue, it is only necessary for ONE of the following to be true:

1. Queue access is unlimited.
2. The user's login group has access. (See the **Add Groups** description on page 4-12.)
3. The user's account has access. (See the **Add Users** description on page 4-14.)

Full NQS manager privileges are required to use this command.

QMGR *(cont.)* QMGR *(cont.)*

Set Working_set_limit

SEt Working_set_limit = (*limit*) *queue*

limit A value indicating *limit* size.

queue A named batch queue.

Set a per-process maximum working set size *limit* for a batch *queue* against which the per-process maximum working set size limit for a request may be compared. If the local host does not support per-process working set size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum working set size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum working set size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. If an attempt is made to queue a request which asks for a larger working set limit than the destination queue has defined as allowable, then the request is rejected. Working set limits cannot be applied to pipe queues. Any attempt to set a working-set-limit for a pipe queue will fail, with an appropriate error diagnostic being displayed.

The working set limit assigned to a particular request is determined at the time that the request is queued. If the working set limit for the containing queue is later reduced, no requests in the queue will be affected. However, if the new working set limit is less than the working set limit requested by a previously queued request, then a warning diagnostic is displayed. For the syntax of *limit*, see the "Limits" section on page 4-59.

Full NQS manager privileges are required to use this command.

Show All

SHOw All

Display the standard amount of information about devices, forms, limits supported, managers, parameters, and queues.

Show Device

SHOw Device [*device-name*]

device-name The valid name of an NQS device.

If no *device-name* is specified, display general status information on all NQS devices on this host.

If a *device-name* is specified, more detailed information will be displayed on that device. If a *device-name* is specified, then status information concerning the named device is displayed including any queue to device mappings for the named device.

Show Forms

SHOw Forms

Display the list of valid forms included in the device forms list.

Show Limits_supported

SHOw LImits_supported

Display the list of NQS resource limit types which are meaningful on this machine. If a limit type is meaningful on a machine, then the corresponding **qmgr** commands will allow the association of a limit of that type with any batch queue on that machine. Note that users may request resource limits that are NOT meaningful on the machine where **qsub** is invoked. If the request is to be executed on a remote machine where the limit is meaningful, then NQS will honor it. Otherwise the unsupported limit is simply ignored. For a more complete discussion of the limits supported by NQS, see the "Limits" section on page 4-59.

QMGR *(cont.)* QMGR *(cont.)*

Show Long Queue

SHOw LOng Queue [*queue-name* [*user-name*]]

queue-name The valid name of an NQS queue.

user-name The valid name of an NQS user.

Display in long format the status of all NQS queues on this host.

If no queue name is specified, then status information is displayed for all NQS queues. If a *queue-name* is specified, output will be limited to that queue. The specific status information for the named queue is displayed, including the ordering of requests within the queue. If a *user-name* is specified, output will downplay any requests not belonging to that user.

Show Managers

SHOw Managers

Display the list of authorized NQS managers.

Show Parameters

SHOw Parameters

Display the current general NQS parameters.

Show Queue

SHOw Queue [*queue-name* [*user-name*]]

queue-name The valid name of an NQS queue.

user-name The valid name of an NQS user.

Display the status (in short format) of all NQS queues on this host. If no *queue-name* is specified, then status information is displayed for all NQS queues. If a *queue-name* is specified, output will be limited to that queue. If a *user-name* is specified, output will downplay any requests not belonging to that user.

QMGR *(cont.)* QMGR *(cont.)*

Shutdown

SHUtdown [*seconds*]

seconds A real value. If *seconds* is not specified, then a default value of twenty (20) seconds is used.

Shut down NQS on the local host. A SIGTERM signal is sent to each process of each request presently running. After the specified number of *seconds* of real time have elapsed, a SIGKILL signal is sent to all remaining processes for each request. If a *seconds* value is not specified, then the delay is sixty seconds. Unlike **ABort Queue**, **SHUtdown** requeues all of the requests it kills, provided that the initial SIGTERM signal is caught or ignored by the running request.

NQS operator privileges are required to use this command.

Start Queue

STArt Queue *queue*

queue A named device, batch, or pipe queue.

If the *queue* is already started, then nothing happens. Otherwise, the *queue* is started and requests in the *queue* are eligible for selection.

NQS operator privileges are required to use this command.

Stop Queue

STOp Queue *queue*

queue A named device, batch, or pipe queue.

Any requests in the *queue* that are currently running are allowed to complete. All other requests are frozen in the *queue*. New requests can still be submitted to the *queue*, but will be frozen like the other requests in the *queue*. No requests in the named *queue* will be run until the *queue* has been explicitly started again by the **STArt Queue** command.

NQS operator privileges are required to use this command.

Unlock Local_daemon

Unlock Local_daemon

Remove a lock that has been keeping the NQS local daemon in memory. See the **plock(2)** description in the UNIX documentation for more information.

NQS operator privileges are required to use this command.

QMGR *(cont.)*

QMGR *(cont.)*

Queue Types

NQS supports four different queue types, that serve to provide four very different functions. These four queue types are known as *batch, device, pipe,* and *network.*

The queue type of *batch* can only be used to execute NQS batch requests. Only NQS batch requests created by the **qsub** command (see the **QSUB** description on page 4-71) can be placed in a batch queue.

The queue type of *device* can only be used to execute NQS device requests. Only NQS device requests created by the **qpr** command (see the **QPR** description on page 4-62) can be placed in a device queue.

Queues of type *pipe,* are used to send NQS requests to other pipe queues, or to request destination queues of type batch or device, as appropriate for the request type. In general, pipe queues in combination with network queues, act as the mechanism that NQS uses to transport both batch and device requests to distant queues on other remote machines. It is also perfectly legal for a pipe queue to transport requests to queues on the same machine.

When a pipe queue is defined, it is given a destination set, which defines the set of possible destination queues for requests entered in that pipe queue. In this manner, it is possible for a batch or device request to pass through many pipe queues on its way to its ultimate destination, which must eventually be a queue of type batch or device (matching the request type).

Each pipe queue has an associated server. For each request handled by a pipe queue, the associated server is spawned which must select a queue destination for the request being handled, based on the characteristics of the request, and upon the characteristics of each queue in the destination set defined for the pipe queue.

Since a different server can be configured for each pipe queue, and batch and device queues can be endowed with the *pipeonly* attribute that will only admit requests queued via another pipe queue, it is possible for respective NQS installations to use pipe queues as a request class mechanism, placing requests that ask for different resource allocations in different queues, each of which can have different associated limits and priorities.

It is also completely possible for a pipe client (pipe queue server) when handling a request, to discover that no destination queue will accept the request, for various reasons which can include insufficient resource limits to execute the request, or a lack of a corresponding account or privilege for queueing at a remote queue. In such circumstances, the request will be deleted, and the user will be notified by mail (see the **mail** and **mailx** descriptions in the UNIX documentation).

The queue type of *network* as alluded to earlier, is implicitly used by pipe queues to pass NQS requests between machines, and is also used to handle queued file transfer operations.

QMGR *(cont.)*

QMGR *(cont.)*

Queue Access

NQS supports queue access restrictions. For each queue of queue type other than network, access may be either unrestricted or restricted. If access is unrestricted, any request may enter the queue. If access is restricted, a request can only enter the queue if the requester or the requester's login group has been given access. Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Limits

NQS supports many batch request resource limit types that can be applied to an NQS batch queue. The configurability of these limits allows an NQS manager to set batch queue-specific resource limits which all batch requests in the queue must adhere to.

The syntax of a *limit* in commands of the form

SEt Some_limit = (*limit*) *queue*

is quite flexible.

For finite CPU time limits, the acceptable syntax is as follows:

```
[[hours :] minutes : ] seconds [.milliseconds]
```

Whitespace can appear anywhere between the principal tokens, with the exception that no whitespace can appear around the decimal point.

Example time limit-values are:

```
1234 : 58 : 21.29     - 1234 hrs 58 mins 21.290 secs
12345                 - 12345 seconds
121.1                 - 121.100 seconds
59:01                 - 59 minutes and 1 second
```

For all other finite limits (with the exclusion of the *nice-value*), the acceptable syntax is:

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

QMGR *(cont.)*

QMGR *(cont.)*

where the *integer* and *fraction* tokens represent strings of up to eight decimal digits, denoting the obvious values. In both cases, the *units* of allocation may also be specified as one of the case insensitive strings:

b	- bytes
w	- words
kb	- kilobytes (2^{10} bytes)
kw	- kilowords (2^{10} words)
mb	- megabytes (2^{20} bytes)
mw	- megawords (2^{20} words)
gb	- gigabytes (2^{30} bytes)
gw	- gigawords (2^{30} words)

In the absence of any *units* specification, the units of *bytes* are assumed.

For all limit types with the exception of the *nice-value*, it is possible to state that no limit should be applied. This is done by specifying a *limit* of unlimited, or any initial substring thereof.

The complications caused by batch request resource limits first show up when queueing a batch request in a batch queue. This operation is described in the following paragraphs.

If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type. (See the **QLIMIT** description on page 4-7 to find out what limits are supported by a given machine.)

For each remaining finite limit that can be supported by the underlying UNIX implementation that is not a CPU *time-limit*, or UNIX *nice-value*, the *limit-value* is internally converted to the units of bytes or words, whichever is more appropriate for the underlying machine architecture.

As an example, a *per-process memory size limit value* of 321 megabytes would be interpreted as 321 x 2^{20} bytes, provided that the underlying machine architecture was capable of directly addressing single bytes. Thus the original limit coefficient of 321 would become 321 x 2^{20}. On a machine that was only capable of addressing words, the appropriate conversion of 321 x 2^{20} *bytes / #of-bytes-per-word* would be performed.

If the result of such a conversion would cause overflow when the coefficient was represented as a *signed-long integer* on the supporting hardware, then the coefficient is replaced with the coefficient of: of 2^{N-1} where N is equal to the number of bits of precision in a signed long integer. For typical 32-bit machines, this *default extreme limit* would therefore be 2^{31-1} bytes. For word addressable machines in the supercomputer class supporting 64-bit long integers, the *default extreme limit* would be 2^{63-1} words.

QMGR *(cont.)* QMGR *(cont.)*

Lastly, some implementations of UNIX reserve coefficients of the form: 2^{N-1} as synonymous with infinity, meaning no limit is to be applied. For such UNIX implementations, NQS further decrements the *default extreme limit* so as to not imply infinity.

The identical internal conversion process as described in the preceding paragraphs is also performed for all *finite limit-values* specified with a particular batch request.

After each applicable request *limit* has been converted as described above, the resulting limit is then compared against the corresponding limit as configured for the destination batch queue. If the corresponding batch queue limit for all batch request limits is defined as unlimited, or is greater than or equal to the corresponding batch request limit, then the request can be successfully queued, provided that no other anomalous conditions occur. For requests that ask for a limit of infinity, the corresponding queue limit must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the **qsub** command, or by the indirect placement of a batch request into a batch queue via a pipe queue. It is impossible for a batch request to be queued in an NQS batch queue if any of these resource limit checks fail.

Finally, if a request fails to specify a limit for a resource limit type that is supported on the execution machine, then the corresponding limit as configured for the destination queue becomes the *limit* for the request.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent **qmgr** commands that alter the limits of the containing batch queue.

See Also

passwd, plock, qdel, qdev, qlimit, qpr, qstat, and **qsub**

QPR QPR

Submit a hardcopy print request to NQS.

Synopsis

qpr [*flags*] [*files*]

Description of Parameters

flags Any of the flags for the **qpr** command. A terse list of the **qpr** flags follows. Refer to the "Discussion" section for a more complete description of the **qpr** flags.

-a	run request after stated time
-f	limit devices to those with stated forms
-mb	send mail to originator when request begins
-me	send mail to originator when request ends
-mu	redirect any mail to stated user
-n	copies to be printed
-p	priority value
-q	submit device request to stated queue
-r	assign stated name to this request
-z	submit request silently

files The valid names of any files that are to be queued for printing.

Discussion

The **qpr** command places the named files in a Network Queueing System (NQS) queue to be printed by a device such as a line printer or laser printer. If no files are specified, **qpr** will read from the standard input.

In the absence of the **-z** flag, **qpr** will print a request-id on the standard output, upon successful queueing of a request. This *request-id* can be compared with what is reported by **qdev** and **qstat** to find out what happened to a request, and given as an argument to **qdel** to delete a request. A *request-id* is always of the form: *seqno.hostname* where *seqno* refers to the sequence number assigned to the request by NQS, and *hostname* refers to the name of originating local machine. This identifier is used throughout NQS to uniquely identify the request, no matter where it is in the network.

The following options to **qpr** may appear in any order and may be intermixed with file names.

QPR *(cont.)* QPR *(cont.)*

-a flag

-a *date-time* Submit at the specified date and/or time. In the absence of this flag, **qpr** will submit the request immediately.

If a *date-time* specification is comprised of two or more tokens separated by whitespace characters, then the date-time specification must be placed within double quotes as in: **-a** `"July, 4, 2026 12:31-EDT"`, or otherwise escaped such that the shell will interpret the entire *date-time* specification as a single lexical token.

The syntax accepted for the *date-time* parameter is relatively flexible. Unspecified date and time values default to an appropriate value (e.g. if no date is specified, then the current month, day, and year are assumed).

A date can be specified as a month and day (current year assumed). The year can also be explicitly specified. It is also possible to specify the date as a weekday name (e.g. Tues), or as one of the strings today or tomorrow. Weekday names and month names can be abbreviated by any three character (or longer) prefix of the actual name. An optional period can follow an abbreviated month or day name.

Time of day specifications can be given using a twenty-four hour clock, or am and pm specifications may be used alternatively. In the absence of a meridian specification, a twenty-four hour clock is assumed.

It should be noted that the time of day specification is interpreted using the precise meridian definitions whereby 12am refers to the twenty-four hour clock time of 0:00:00, 12m refers to noon, and 12-pm refers to 24:00:00. Alternatively, the phrases midnight and noon are accepted as time of day specifications, where midnight refers to the time of 24:00:00.

A timezone may also appear at any point in the *date-time* specification. Thus, it is legal to say: April 1, 1987 13:01-PDT. In the absence of a timezone specification, the local timezone is assumed, with daylight savings time being inferred when appropriate, based on the date specified.

All alphabetic comparisons are performed in a case insensitive fashion such that both WeD and weD refer to the day of Wednesday.

Some valid *date-time* examples are:

01-Jan-1986 12am, PDT
Tuesday, 23:00:00
11pm tues.
tomorrow 23-MST

QPR *(cont.)* QPR *(cont.)*

-f flag

-f *form-name* Limit the set of acceptable devices to those devices which are loaded with the forms: *form-name*. In the absence of this flag, **qpr** will submit the request only to a device that is loaded with the default forms. If there is no default forms defined, the request will be submitted to the appropriate output device without regard to the forms configured for the device.

In any case, only those devices associated with the chosen queue will be considered.

-mb flag

-mb Send mail to the user on the originating machine when the request begins execution. If the **-mu** flag is also present, then mail is sent to the user specified for the **-mu** flag instead of to the invoking user.

-me flag

-me Send mail to the invoker on the originating machine when the request has ended execution. If the **-mu** flag is also present, then mail is sent to the user specified for the **-mu** flag instead of to the invoking user.

-mu flag

-mu *user-name* Specify that any mail concerning the request should be delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no @ characters), or as *user@machine*. In the absence of this flag, any mail concerning the request will be sent to the invoker on the originating machine.

-n flag

-n *number-of-copies*
Print *number-of-copies* copies. The default is one.

QPR *(cont.)* QPR *(cont.)*

-p flag

-p *priority* Assign an intra-queue priority to this request. The specified *priority* must be an integer, and must be in the range [0..63], inclusive. A value of 63 defines the highest intra-queue request priority, while a value of 0 defines the lowest. This priority does not determine the execution priority of the request. This priority is only used to determine the relative ordering of requests within a queue.

When a request is added to a queue, it is placed at a specific position within the queue such that it appears ahead of all existing requests whose priority is less than the priority of the new request. Similarly, all requests with a higher priority will remain ahead of the new request when the queueing process is complete. When the priority of the new request is equal to the priority of an existing request, the existing request takes precedence over the new request.

If no intra-queue priority is chosen by the user, then NQS assigns a default value.

-q flag

-q *queue-name* Specify the queue to which the device request is to be submitted. If no **-q** *queue-name* specification is given, then the user's environment variable set is searched for the variable: *QPR_QUEUE*. If this environment variable is found, then the character string value for *QPR_QUEUE* is presumed to name the queue to which the request should be submitted. If the *QPR_QUEUE* environment variable is not found, then the request will be submitted to the default device request queue, if defined by the local System Administrator. Otherwise, the request cannot be queued, and an appropriate error message is displayed to this effect.

QPR *(cont.)* QPR *(cont.)*

-r flag

-r *request-name* Assign a name to this request. In the absence of an explicit **-r** *request-name* specification, the *request-name* defaults to the name of the first print file (leading path name removed) specified on the command line. If no print files were specified, then the default *request-name* assigned to the request is *stdin*.

In all cases, if the *request-name* is found to begin with a digit, then the character "R" is prepended to prevent a *request-name* from beginning with a digit. All *request-names* are truncated to a maximum length of 15 characters.

Be sure not to confuse *request-name* with *request-id*.

-z flag

-z Submit the request silently. If the request is submitted successfully, nothing will be written to *stdout* or *stderr*.

QPR *(cont.)*

QPR *(cont.)*

Queue Access

NQS supports queue access restrictions. For each queue of queue type other than network, access may be either unrestricted or restricted. If access is unrestricted, any request may enter the queue. If access is restricted, a request can only enter the queue if the requester or the requester's login group has been given access to that queue (see the **QMGR** description on page 4-9). Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Use **qstat** to determine who has access to a particular queue.

See Also

mail, qdel, qdev, qlimit, qmgr, qstat, and **qsub**

QSTAT

QSTAT

Display status of NQS requests and queues.

Synopsis

qstat [**-a**] [**-U**] [**-b**] [**-d**] [**-p**] [**-f**] [**-l**] [**-n**]
[**-s** *state -rqht-*] [**-h** *host-name*] [**-u** *user-name*]
[**-T** *user-target-name*] [*queue-name ...*]
[*queue-name@host-name ...*]

Description of Parameters

-a Displays all requests. The **-U** (unrestricted) option is synonymous.

-U Unrestricted (same as the **-a** option). Displays all requests.

-b Displays batch queues.

-d Displays device queues.

-p Displays pipe queues.

-f Queues are shown in a full format. The **-l** (long) option displays in the same format.

-l Queues are shown in a full format. The **-f** (full) option displays in the same format.

-n The queue header and trailer are not displayed.

-s *state -rqht-* Queue state.

-h *host-name ...* The named host or hosts to which this command applies. In the absence of a named host, the local host is assumed.

-u *user-name* Shows only those requests belonging to *user-name*.

-T *user-target-name*
Shows only those requests belonging to *user-target-name*.

QSTAT *(cont.)*

QSTAT *(cont.)*

queue-name ... The name of a queue for which the status is being requested. The local host is assumed if it is not indicated with the **-h** *host-name* specifier.

queue-name@host-name ...
The name of a queue for which the status is being requested. The *host-name* host is indicated.

Discussion

The **qstat** command displays the status of Network Queueing System (NQS) requests and queues.

If no objects are specified, then the current state of each NQS request on the local host is displayed. Otherwise, information is displayed for the specified object only. Each entry displays information about a given request. Ordinarily, **qstat** shows only those requests belonging to the invoker.

If information about the queues is requested with the **-b**, **-d**, or **-p** options, but no queues are specified, then the current state of each NQS queue on the local host is displayed. Otherwise, information is displayed for the specified queues only. Queues may be specified either as *queue-name* or *queue-name@host-name*. In the absence of a *host-name* specifier, the local host is assumed.

For each selected queue, **qstat** displays information about the queue itself. The following flags are available:

-a Displays all requests. The **-U** (unrestricted) option is synonymous.

-b Displays batch queues.

-d Displays device queues.

-p Displays pipe queues.

-f Queues are shown in a full format. The **-l** (long) option displays in the same format.

-n The queue header and trailer are not displayed.

-u *user-name* Shows only those requests belonging to *user-name*.

When a queue is being examined, the queue name, host machine, priority, number of requests in a given state, resource limits, and access are displayed.

QSTAT *(cont.)*

QSTAT *(cont.)*

Queue State

The general state of a queue is defined by two principal properties of the queue.

The first property determines whether or not requests can be submitted to the queue. If they can, then the queue is said to be enabled. Otherwise the queue is said to be disabled.

The second principal property of a queue determines if requests which are ready to run, but are not now presently running, will be allowed to run upon the completion of any currently running requests, and whether any requests are presently running in the queue.

If queued requests not already running are blocked from running, and no requests are presently executing in the queue, then the queue is said to be stopped. If the same situation exists with the difference that at least one request is running, then the queue is said to be stopping, where the requests presently executing will be allowed to complete execution, but no new requests will be spawned.

One of the words AVAILABLE, STOPPED, DISABLED, UNAVAIL, or NQS DOWN will appear in the queue status field to indicate the respective queue states of:

AVAILABLE= enabled and started,
STOPPED = enabled and stopped,
DISABLED = disabled and running or
UNAVAIL = disabled and stopped.

Requests can only be submitted to the queue if the queue is enabled, and the local NQS daemon is present.

If the NQS daemon for the local host upon which the queue resides is not running, the status displays NQS DOWN.

Caveats

NQS is not finished, and continues to undergo development. Some of the request states shown above may or may not be supported in your version of NQS.

See Also

qdel, qdev, qlimit, qmgr, qpr, and **qsub**

QSUB QSUB

Submit an NQS batch request.

Synopsis

qsub [*flags*] [*script-file*]

Description of Parameters

flags Any of the flags implemented for the **qsub** command. What follows is a terse definition of the flags implemented by the **qsub** command (see the "Long Description" section on page 4-72 for the complete definition and syntax used for each of these flags).

-a	run request after stated time
-e	direct stderr output to stated destination
-eo	direct stderr output to the stdout destination
-ke	keep stderr output on the execution machine
-ko	keep stdout output on the execution machine
-lc	establish per-process corefile size limit
-ld	establish per-process data-segment size limits
-lf	establish per-process permanent-file size limits
-lF	establish per-request permanent-file space limits
-lm	establish per-process memory size limits
-lM	establish per-request memory space limits
-ln	establish per-process nice execution value limit
-ls	establish per-process stack-segment size limits
-lt	establish per-process CPU time limits
-lT	establish per-request CPU time limits
-lv	establish per-process temporary-file size limits
-lV	establish per-request temporary-file space limits
-lw	establish per-process working set limit
-mb	send mail when the request begins execution
-me	send mail when the request ends execution
-mu	send mail for the request to the stated user
-nr	declare that batch request is not restartable
-o	direct stdout output to the stated destination
-p	specify intra-queue request priority
-q	queue request in the stated queue
-r	assign stated request name to the request
-re	remotely access the stderr output file
-ro	remotely access the stdout output file

QSUB *(cont.)*

QSUB *(cont.)*

-s	specify shell to interpret the batch request script
-x	export all environment variables with request
-z	submit the request silently

script-file An optional batch file that contains embedded default flags that set default characteristics for the batch request.

Discussion

The **qsub** command submits a batch request to the Network Queueing System (NQS).

If no *script-file* is specified, then the set of commands to be executed as a batch request is taken directly from the standard input file (*stdin*). In all cases however, the *script-file* is spooled, so that later changes will not affect previously queued batch requests.

All of the flags that can be specified on the command line can also be specified within the first comment block inside the batch request script file as embedded default flags. Such flags appearing in the batch request script file set default characteristics for the batch request. If the same flag is specified on the command line, then the command line flag (and any associated value) takes precedence over the embedded flag. See the "Long Description" section on page 4-72 for more information on embedded default flags.

Long Description

As described above, it is possible to specify default flags within the batch request script file that configure the default behavior of the batch request. The algorithm used to scan for such embedded default flags within an NQS batch request script file is as follows:

1. Read the first line of the script file.

2. If the current line contains only whitespace characters, or the first non-whitespace character of the line is : (colon), then goto step 7.

3. If the first non-whitespace character of the current line is not a # (pound) character, then goto step 8.

4. If the second non-whitespace character in the current line is *not* the @ (at) character, or the character immediately following the second non-whitespace character in the current line is not a $ (dollar sign) character, or if the second non-whitespace character is not a Q character followed immediately by the string "SUB", then goto step 7.

QSUB *(cont.)*

QSUB *(cont.)*

5. If no - is present as the character immediately following the "@$" sequence or the "QSUB" sequence, then goto step 8.

6. Process the embedded flag, stopping the parsing process upon reaching the end of the line, or upon reaching the first unquoted # character.

7. Read the next script file line. Go to step 2.

8. End. No more embedded flags will be recognized.

Here is an example of the use of embedded flags within the script file.

```
#
#  Batch request script example:
#
#  @$-a "11:30pm EDT" -lt "21:10, 20:00"
#       # Run request after 11:30 EDT by default,
#       # and set a maximum per-process CPU time
#       # limit of 21 minutes and ten seconds.
#       # Send a warning signal when any process
#       # of the running batch request consumes
#       # more than 20 minutes of CPU time.
#  QSUB-lT 1:45:00
#       # Set a maximum per-request CPU time limit
#       # of one hour, and 45 minutes.  (The
#       # implementation of CPU time limits is
#       # completely dependent upon the UNIX
#       # implementation at the execution
#       # machine.)
#  QSUB-mb -me
# Send mail at beginning and end of
#       # request execution.
#  @$-q batch1 # Queue request to queue: batch1 by
#       # default.
#  @$   # No more embedded flags.
#
make all
```

The following paragraphs give the detailed descriptions of the *flags* supported by the **qsub** command.

QSUB *(cont.)* QSUB *(cont.)*

-a flag

-a *date-time* Do not run the batch request before the specified date and/or time. If a *date-time* specification has two or more tokens separated by whitespace characters, then the *date-time* specification must be placed within double quotes as in: **-a** `"July, 4, 2026 12:31-EDT"`, or otherwise escaped such that **qsub** and the shell will interpret the entire date-time specification as a single character string. This restriction also applies when an embedded default **-a** flag with accompanying *date-time* specification appears within the batch request script file.

The syntax accepted for the *date-time* parameter is relatively flexible. Unspecified date and time values default to an appropriate value (e.g. if no date is specified, then the current month, day, and year are assumed).

A date may be specified as a month and day (current year assumed), or the year can also be explicitly specified. It is also possible to specify the date as a weekday name (e.g. Tues), or as one of the strings: today, or tomorrow. Weekday names and month names can be abbreviated by any three character (or longer) prefix of the actual name. An optional period can follow an abbreviated month or day name.

Time of day specifications can be given using a twenty-four hour clock, or am and pm specifications may be used alternatively. In the absence of a meridian specification, a twenty-four hour clock is assumed.

It should be noted that the time of day specification is interpreted using the precise meridian definitions whereby "12am" refers to the twenty-four hour clock time of 0:00:00, "12m" refers to noon, and "12-pm" refers to 24:00:00. Alternatively, the phrases midnight and noon are accepted as time of day specifications, where midnight refers to the time of 24:00:00.

A time zone may also appear at any point in the *date-time* specification. Thus, it is legal to say: April 1, 1987 13:01-PDT. In the absence of a timezone specification, the local timezone is assumed, with daylight savings time being inferred when appropriate, based on the date specified.

All alphabetic comparisons are performed in a case insensitive fashion such that both WeD and weD refer to the day of Wednesday.

Some valid date-time examples are:

```
01-Jan-1986 12am, PDT
Tuesday, 23:00:00
11pm tues.
tomorrow 23-MST
```

QSUB *(cont.)* QSUB *(cont.)*

-e flag

-e [*machine:*][[/]*path/*] *stderr-filename*

Direct output generated by the batch request which is sent to the *stderr* file to the named [*machine:*][[/]*path/*] *stderr-filename*. The brackets [and] enclose optional portions of the *stderr* destination *machine* , *path* , and *stderr-filename*.

If no explicit *machine* destination is specified, then the destination machine defaults to the machine that originated the batch request, or to the machine that will eventually run the request, depending on the respective absence, or presence of the **-ke** flag.

If no machine destination is specified, and the path/filename does not begin with a /, then the current working directory is prepended to create a fully qualified path name, provided that the **-ke** (keep stderr) flag is also absent. In all other cases, any partial path/filename is interpreted relative to the user's home directory on the *stderr* destination machine.

This flag cannot be specified when the **-eo** flag option is also present.

If the **-eo** and **-e** [*machine:*][[/]*path/*] *stderr-filename* flag options are not present, then all *stderr* output for the batch request is sent to the file whose name consists of the first seven characters of the *request-name* followed by the characters: .e, followed by the request sequence number portion of the *request-id* discussed below. In the absence of the **-ke** flag, this default *stderr* output file will be placed on the machine that originated the batch request in the current working directory, as defined when the batch request was first submitted. Otherwise, the file will be placed in the user's home directory on the execution machine.

-eo flag

-eo

Direct all output that would normally be sent to the *stderr* file to the *stdout* file for the batch request. This flag cannot be specified when the **-e** [*machine:*][[/]*path/*] *stderr-filename* flag option is also present.

QSUB *(cont.)*

QSUB *(cont.)*

-ke flag

-ke

In the absence of an explicit *machine* destination for the *stderr* file produced by a batch request, the *machine* destination chosen for the *stderr* output file is the machine that originated the batch request. In some cases however, this behavior may be undesirable, and so the **-ke** flag can be specified which instructs NQS to leave any *stderr* output file produced by the request on the machine that actually executed the batch request.

This flag is meaningless if the **-eo** flag is specified, and cannot be specified if an explicit machine destination is given for the *stderr* parameter of the **-e** flag.

-ko flag

-ko

In the absence of an explicit machine destination for the *stdout* file produced by a batch request, the machine destination chosen for the *stdout* output file is the machine that originated the batch request. In some cases however, this behavior may be undesirable, and so the **-ko** flag can be specified which instructs NQS to leave any *stdout* output file produced by the request on the machine that actually executed the batch request.

This flag cannot be specified if an explicit machine destination is given for the *stdout* parameter of the **-o** flag.

QSUB *(cont.)* QSUB *(cont.)*

-lc flag

-lc *per-process corefile size limit*

Set a *per-process* maximum *corefile size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to exit creating a core file whose size would exceed the maximum *per-process corefile size limit* for the request, then the core file image of the aborting process will be reduced to the necessary size by an algorithm dependent upon the underlying UNIX implementation.

Not all UNIX implementations support *per-process corefile size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process corefile size limit*.

QSUB *(cont.)* QSUB *(cont.)*

-ld flag

-ld *per-process data-segment size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *data-segment size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process data-segment size-limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process data-segment* warning size limits. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-ld** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-process data-segment size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process data-segment size limit*.

QSUB *(cont.)* QSUB *(cont.)*

-lf flag

-lf *per-process permanent-file size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *permanent-file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a permanent file such that the file size would increase beyond the maximum *per-process permanent-file size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning permanent-file size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lf** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-process permanent-file size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

At the time of this writing, the author was unaware of any UNIX implementation that made a distinction at the kernel level, between permanent, and temporary files. While it is certainly possible to construct a pseudo-temporary file by first creating it, and then unlinking its pathname, the disk space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from. Furthermore, such a file will be subject to the same quota enforcement mechanisms, if any are provided by the underlying UNIX implementation, that all other UNIX files are created under.

For all UNIX implementations that do not support a distinction between permanent, and temporary files at the kernel level, this limit is interpreted as a *per-process file size limit*, with the word *permanent* removed from the definition.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process permanent-file size limit*.

QSUB *(cont.)* QSUB *(cont.)*

-lF flag

-lF *per-request permanent-file space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *permanent-file space limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a permanent file such that the file space consumed by all permanent files opened for writing by all of the processes in the batch request, would increase beyond the maximum *per-request permanent-file space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning permanent-file space limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lF** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-request permanent-file space limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

At the time of this writing, the author was unaware of any UNIX implementation that made a distinction at the kernel level, between permanent, and temporary files. While it is certainly possible to construct a pseudo-temporary file by first creating it, and then unlinking its pathname, the disk space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from. Furthermore, such a file will be subject to the same quota enforcement mechanisms, if any are provided by the underlying UNIX implementation, that all other UNIX files are created under.

For all UNIX implementations that do not support a distinction between permanent, and temporary files at the kernel level, this limit is interpreted as a *per-request file space limit*, with the word *permanent* removed from the definition.

QSUB *(cont.)*

QSUB *(cont.)*

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request permanent-file space limit*.

-lm flag

-lm *per-process memory size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning m*emory size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process memory size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning memory size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lm** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-process memory size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process memory size limit*.

QSUB *(cont.)* QSUB *(cont.)*

-lM flag

-lM *per-request memory space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *memory space limit* for all processes that constitute the running batch request. If the sum of all memory consumed by all of the processes comprising the running request exceeds the maximum *per-request memory space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning memory size limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default -lM flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-request memory space limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request memory space limit*.

QSUB *(cont.)*

QSUB *(cont.)*

-ln flag

-ln *per-process nice value limit*

Set a *per-process nice value* for all processes comprising the running batch request.

At present, all UNIX implementations support the use of an integer called the *nice* value, which determines the execution-time priority of a process relative to all other processes in the system. By letting the user set a limit on the *nice* value for all processes comprising the running request, a user can cause a request to consume less (or more) of the CPU resource presented by the execution machine.

This is particularly useful when a user wishes to execute a CPU intensive batch request on a machine running interactive processes. By setting a low execution-time priority, a user can make a long running batch request give way to more interactive processes during the daytime, while the coming of the nighttime hours with typically smaller process loads will allow such a request to gain more and more of the CPU resource. In this way, long running batch requests can be polite to their more transient, interactive neighbor processes.

The only quirk associated with this flag results from the peculiar choice of nice values, implemented by the standard UNIX implementations. In general, increasingly negative nice values cause the relative execution priority of a process to increase, while increasingly positive nice values causes the relative priority to decrease! Thus, a *nice value* limit specification of: -ln -10 is greedier than a *nice value* limit specification of: -ln 0.

Since varying UNIX implementations often support a different finite range of *nice values*, NQS allows the specification of *nice values* that can eventually turn out to be outside the limits for the UNIX implementation running at the execution machine. In such cases, NQS will simply bind the specified *nice value* limit to within the necessary range as appropriate.

Lastly, any *nice value* specified by the use of this flag must be acceptable to the batch queue in which the request is ultimately placed (see the "Limits" section on page 4-98 for more information).

QSUB *(cont.)*

QSUB *(cont.)*

-ls flag

-ls *per-process stack-segment size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *stack-segment size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process stack-segment size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning stack-segment size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-ls** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-process stack-segment size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process stack-segment size limit*.

QSUB *(cont.)*

QSUB *(cont.)*

-lt flag

-lt *per-process CPU time limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *CPU time limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process CPU time limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process CPU warning time limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lt** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-process CPU time limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process CPU time limit*.

QSUB *(cont.)* QSUB *(cont.)*

-lT flag

-lT *per-request CPU time limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *CPU time limit* for all of the processes that constitute the running batch request. If the sum of the CPU times consumed by all of the processes in the batch request exceeds the maximum *per-request CPU time limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request CPU warning time limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lT** flag with its associated limit value(s) appears within the batch request script file.

Not all UNIX implementations support *per-request CPU time limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request CPU time limit*.

QSUB *(cont.)* QSUB *(cont.)*

-lv flag

-lv *per-process temporary file size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *temporary* (volatile) *file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a *temporary* file such that the file size would increase beyond the maximum *per-process temporary-file size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning temporary-file size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lv** flag with its associated limit value(s) appears within the batch request script file.

At the time of this writing, no UNIX operating system known to the author supported a distinction at the kernel level between permanent and temporary files. Certainly, a pseudo-temporary file can be constructed by creating it, and then unlinking its pathname. However, the file space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from.

Until a mechanism is implemented in the kernel that knows about permanent and temporary files, this limit cannot be supported in the sense most useful for batch requests, namely the strict enforcement of disk quotas for permanent versus temporary files.

Until such a time, this limit will simply be ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process temporary-file size limit.*

QSUB *(cont.)* QSUB *(cont.)*

-lV flag

-lV *per-request temporary file space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *temporary (volatile) file space limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a *temporary* file such that the file space consumed by all temporary files opened for writing by all of the processes in the batch request would increase beyond the maximum *per-request temporary-file space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning temporary-file space limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that **qsub** and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **-lV** flag with its associated limit value(s) appears within the batch request script file.

At the time of this writing, no UNIX operating system known to the author supported a distinction at the kernel level between permanent and temporary files. Certainly, a pseudo-temporary file can be constructed by creating it, and then unlinking its pathname. However, the file space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from.

Until a mechanism is implemented in the kernel that knows about permanent and temporary files, this limit cannot be supported in the sense most useful for batch requests, namely the strict enforcement of disk quotas for permanent versus temporary files.

Until such a time, this limit will simply be ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *temporary-file space limit*.

QSUB *(cont.)* QSUB *(cont.)*

-lw flag

-lw *per-process working set size limit*

Set a *per-process* maximum *working set size limit* for all processes that constitute the running batch request.

Not all UNIX implementations support *per-process working set size limits*, and such a limit only makes sense in the context of a paged virtual memory machine. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the "Limits" section on page 4-98 for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process working set size limit*.

-mb flag

-mb Send mail to the user on the originating machine when the request begins execution. If the **-mu** flag is also present, then mail is sent to the user specified for the **-mu** flag instead of to the invoking user.

-me flag

-me Send mail to the user on the originating machine when the request has ended execution. If the **-mu** flag is also present, then mail is sent to the user specified for the **-mu** flag instead of to the invoking user.

-mu flag

-mu *user-name* Specify that any mail concerning the request should be delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no @ characters), or as *user@machine*. In the absence of this flag, any mail concerning the request will be sent to the invoker on the originating machine.

QSUB *(cont.)* QSUB *(cont.)*

-nr flag

-nr Declare that the request is non-restartable. If this flag is specified, then the request will not be restarted by NQS upon system boot if the request was running at the time of an NQS shutdown or system crash.

By default, NQS assumes that all requests are restartable, with the caveat that it is the responsibility of the user to ensure that the request will execute correctly if restarted, by the use of appropriate programming techniques.

Requests that are not running are always preserved across host crashes and NQS shutdowns for later requeueing, with or without this flag.

When NQS is shutdown via an operator command to the **qmgr** NQS control program, a SIGTERM signal is sent to all processes comprising all running NQS requests on the local host, and all queued NQS requests are barred from beginning execution. After a finite number of seconds have elapsed (typically 60, but this value can be overridden by the operator), all remaining processes comprising all remaining running NQS requests are killed by the signal: SIGKILL.

For an NQS request to be properly restarted after an NQS shutdown, the **-nr** flag must not be specified, and the spawned batch request shell must ignore SIGTERM signals (which is done by default). The spawned batch request shell must also not exit before the final SIGKILL arrives. Since the batch request shell is simply spawning commands and programs, waiting for their completion, this implies that the commands and programs being executed by the batch request shell must also be immune to SIGTERM signals, saving state as appropriate before being killed by the final SIGKILL signal.

See the "Caveats" section on page 4-101 for more discussion concerning the restartability of NQS batch requests.

QSUB *(cont.)* QSUB *(cont.)*

-o flag

-o [*machine:*][[/]*path/*] *stdout-filename*

Direct output generated by the batch request which is sent to the *stdout* file to the named [*machine:*][[/]*path/*] *stdout-filename*.

The brackets "[" and "]" enclose optional portions of the *stdout* destination *machine*, *path*, and *stdout-filename*.

If no explicit *machine* destination is specified, then the destination machine defaults to the machine that originated the batch request, or to the machine that will eventually run the request, depending on the respective absence, or presence of the **-ko** flag.

If no *machine* destination is specified, and the path/filename does not begin with a /, then the current working directory is prepended to create a fully qualified path name, provided that the **-ko** (keep stdout) flag is also absent. In all other cases, any partial path/filename is interpreted relative to the user's home directory on the *stdout* destination machine.

If no **-o** [*machine:*][[/]*path/*] *stdout-filename* flag is specified, then all *stdout* output for the batch request is sent to the file whose name consists of the first seven characters of the *request-name* followed by the characters: .o, followed by the request sequence number portion of the *request-id* discussed below. In the absence of the **-ko** flag, this default *stdout* output file will be placed on the machine that originated the batch request in the current working directory, as defined when the batch request was first submitted. Otherwise, the file will be placed in the user's home directory on the execution machine.

QSUB *(cont.)* QSUB *(cont.)*

-p flag

-p *priority* Explicitly assign an *intra-queue* priority to the request. The specified *priority* must be an integer, and must be in the range [0..63], inclusive. A value of 63 defines the highest *intra-queue* request priority, while a value of 0 defines the lowest. This priority does not determine the execution priority of the request. This priority is only used to determine the relative ordering of requests within a queue.

When a request is added to a queue, it is placed at a specific position within the queue such that it appears ahead of all existing requests whose priority is less than the priority of the new request. Similarly, all requests with a higher priority will remain ahead of the new request when the queueing process is complete. When the priority of the new request is equal to the priority of an existing request, the existing request takes precedence over the new request.

If no *intra-queue* priority is chosen by the user, then NQS assigns a default value.

-q flag

-q *queue-name* Specify the queue to which the batch request is to be submitted. If no **-q** *queue-name* specification is given, then the user's environment variable set is searched for the variable: *QSUB_QUEUE*. If this environment variable is found, then the character string value for *QSUB_QUEUE* is presumed to name the queue to which the request should be submitted. If the *QSUB_QUEUE* environment variable is not found, then the request will be submitted to the default batch request queue, if defined by the local system administrator. Otherwise, the request cannot be queued, and an appropriate error message is displayed to this effect.

-r flag

-r *request-name* Assign the specified *request-name* to the request. In the absence of an explicit **-r** *request-name* specification, the *request-name* defaults to the name of the script file (leading path name removed) given on the command line. If no script file was given, then the default *request-name* assigned to the request is *stdin*.

In all cases, if the *request-name* is found to begin with a digit, then the character R is prepended to prevent a *request-name* from beginning with a digit. All *request-names* are truncated to a maximum length of 15 characters.

QSUB *(cont.)*

QSUB *(cont.)*

-re flag

-re

By default, all output generated by a batch request sent to the *stderr* file is temporarily into a file residing in a protected directory on the machine that executes the request. When the batch request completes execution, this file is then spooled to its final destination, possibly on a remote machine.

This default spooling of the *stderr* output file is done to reduce the network traffic costs incurred by the submitter (owner) of a batch request which is to return its *stderr* output to a remote machine upon completion. In some cases, this behavior is not desired. If it is necessary to override this behavior, then the **-re** flag can be specified which says that *stderr* output produced by the request is to be immediately written to the final destination file, as output is generated, no matter what the networking cost.

Circumstances may not allow a given NQS implementation to support this flag, in which case it will be ignored, and the *stderr* output file will simply be spooled as it ordinarily would without this flag.

-ro flag

-ro

By default, all output generated by a batch request sent to the *stdout* file is temporarily spooled into a file residing in a protected directory on the machine that executes the request. When the batch request completes execution, this file is then spooled to its final destination, possibly on a remote machine.

This default spooling of the *stdout* output file is done to reduce the network traffic costs incurred by the submitter (owner) of a batch request which is to return its *stdout* output to a remote machine upon completion. In some cases, this behavior is not desired. If it is necessary to override this behavior, then the **-ro** flag can be specified which says that *stdout* output produced by the request is to be immediately written to the final destination file, as output is generated, no matter what the networking cost.

Circumstances may not allow a given NQS implementation to support this flag, in which case it will be ignored, and the *stdout* output file will simply be spooled as it ordinarily would without this flag.

QSUB *(cont.)*

QSUB *(cont.)*

-s flag

-s *shell-name* Specify the absolute path name of the shell which will be used to interpret the batch request script. This flag unconditionally overrides any shell strategy configured on the execution machine for selecting which shell to spawn in order to interpret the batch request script.

In the absence of this flag, the NQS system at the execution machine will use one of three (3) distinct shell choice strategies for the execution of the batch request. Any one of the three strategies can be configured by a system administrator for each NQS machine.

The three shell strategies are called *fixed*, *free*, and *login*. These shell strategies respectively cause the configured fixed shell to be executed to interpret all batch requests, cause the user's login shell as defined in the password file to be executed which in turn chooses and spawns the appropriate shell for interpreting the batch request script, or cause only the user's login shell to be executed to interpret the script.

A shell strategy of fixed means that the same shell (as configured by the System Administrator), will be used to execute all batch requests.

A shell strategy of free will run the batch request script exactly as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of fixed and login exist for host systems that are short on available free processes. In these two strategies, a single shell is executed, and that same shell is the shell that executes all of the commands in the batch request script.

The shell strategy configured for a particular NQS system can be determined by the **qlimit** command.

QSUB *(cont.)*

QSUB *(cont.)*

-x flag

-x Export all environment variables. When a batch request is submitted, the current values of the environment variables: *HOME, SHELL, PATH, LOGNAME* (not all systems), *USER* (not all systems), *MAIL*, and *TZ* are saved for later recreation when the batch request is spawned, as the respective environment variables: *QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_USER, QSUB_MAIL*, and *QSUB_TZ*. Unless the **-x** flag is specified, no other environment variables will be exported from the originating host for the batch request. If the **-x** flag option is specified, then all remaining environment variables whose names do not conflict with the automatically exported variables, are also exported with the request. These additional environment variables will be recreated under the same name when the batch request is spawned.

-z flag

-z Submit the batch request silently. If the request is submitted successfully, then no messages are displayed indicating this fact. Error messages will, however, always be displayed.

If the batch request is successfully submitted and the **-z** flag has not been specified, the *request-id* of the request is displayed to the user. A *request-id* is always of the form: *seqno.hostname* where *seqno* refers to the sequence number assigned to the request by NQS, and *hostname* refers to the name of originating local machine. This identifier is used throughout NQS to uniquely identify the request, no matter where it is in the network.

The following events take place in the following order when an NQS batch request is spawned:

1. The process that will become the head of the process group for all processes comprising the batch request is created by NQS.
2. Resource limits are enforced.
3. The real and effective group-id of the process is set to the group-id as defined in the local password file for the request owner.
4. The real and effective user-id of the process is set to the real user-id of the batch request owner.
5. The user file creation mask is set to the value that the user had on the originating machine when the batch request was first submitted.

QSUB *(cont.)*

QSUB *(cont.)*

6. If the user explicitly specified a shell by use of the -s flag (discussed above), then that user-specified shell is chosen as the shell that will be used to execute the batch request script. Otherwise, a shell is chosen based upon the shell strategy as configured for the local NQS system (see the earlier discussion of the -s flag for a description of the possible *shell strategies* that can be configured for an NQS system).

7. The environment variables of *HOME, SHELL, PATH, LOGNAME* (not all systems), *USER* (not all systems), and *MAIL* are set from the user's password file entry, as though the user had logged directly into the execution machine.

8. The environment string: "ENVIRONMENT=BATCH" is added to the environment so that shell scripts (and the user's *.profile* (Bourne shell) or *.cshrc* and *.login* (C-shell scripts), can test for batch request execution when appropriate, and not (for example) perform any setting of terminal characteristics, since a batch request is not connected to an input terminal.

9. The environment variables of *QSUB_WORKDIR, QSUB_HOST, QSUB_REQNAME*, and *QSUB_REQID* are added to the environment. These environment variables equate to the obvious respective strings of the working directory at the time that the request was submitted, the name of the originating host, the name of the request, and the request *request-id.*

10. All of the remaining environment variables saved for recreation when the batch request is spawned are added at this point to the environment. When a batch request is initially submitted, the current values of the environment variables: *HOME, SHELL, PATH, LOGNAME* (not all systems), *USER* (not all systems), *MAIL*, and *TZ* are saved for later recreation when the batch request is spawned. When recreated however, these variables are added to the environment under the respective names: *QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_USER, QSUB_MAIL*, and *QSUB_TZ*, to avoid the obvious conflict with the local version of these environment variables. Additionally, all environment variables exported from the originating host by the -x option are added to the environment at this time.

11. The current working directory is then set to the user's home directory on the execution machine, and the chosen shell is executed to execute the batch request script with the environment as constructed in the steps outlined above.

In all cases, the chosen shell is executed as though it were the login shell. If the Bourne shell is chosen to execute the script, then the *.profile* file is read. If the C-shell is chosen, then the *.cshrc* and *.login* scripts are read.

If the user did not specify a specific shell for the batch request, then NQS chooses which shell should be used to execute the shell script, based on the shell strategy as configured by the system administrator (see the earlier discussion of the -s flag).

QSUB *(cont.)*

QSUB *(cont.)*

In such a case, a free shell strategy instructs NQS to execute the login shell for the user (as configured in the password file). The login shell is in turn instructed to examine the shell script file, and fork another shell of the appropriate type to interpret the shell script, behaving exactly as an interactive invocation of the script.

Otherwise no additional shell is spawned, and the chosen fixed or login shell sequentially executes the commands contained in the shell script file until completion of the batch request.

Queue Types

NQS supports four different queue types that serve to provide four very different functions. These four queue types are known as batch, device, pipe, and network.

The queue type of batch can only be used to execute NQS batch requests. Only NQS batch requests created by the **qsub** command can be placed in a batch queue.

The queue type of device can only be used to execute NQS device requests. Only NQS device requests created by the **qpr** command can be placed in a device queue.

Queues of type pipe are used to send NQS requests to other pipe queues, or to request destination queues of type batch or device, as appropriate for the request type. In general, pipe queues, in combination with network queues, act as the mechanism that NQS uses to transport both batch and device requests to distant queues on other remote machines. It is also perfectly legal for a pipe queue to transport requests to queues on the same machine.

When a pipe queue is defined, it is given a destination set which defines the set of possible destination queues for requests entered in that pipe queue. In this manner, it is possible for a batch or device request to pass through many pipe queues on its way to its ultimate destination, which must eventually be a queue of type batch or device (matching the request type).

Each pipe queue has an associated server. For each request handled by a pipe queue, the associated server is spawned which must select a queue destination for the request being handled, based on the characteristics of the request, and upon the characteristics of each queue in the destination set defined for the pipe queue.

Since a different server can be configured for each pipe queue, and batch and device queues can be endowed with the *pipeonly* attribute that will only admit requests queued via another pipe queue, it is possible for respective NQS installations to use pipe queues as a request class mechanism, placing requests that ask for different resource allocations in different queues, each of which can have different associated limits and priorities.

QSUB *(cont.)*

QSUB *(cont.)*

It is also completely possible for a pipe queue server, when handling a request, to discover that no destination queue will accept the request, for various reasons which can include insufficient resource limits to execute the request, or a lack of a corresponding account or privilege for queueing at a remote queue. In such circumstances, the request will be deleted, and the user will be notified by mail (see **mail**(1)).

The queue type of network, as alluded to earlier, is implicitly used by pipe queues to pass NQS requests between machines, and is also used to handle queued file transfer operations.

Queue Access

NQS supports queue access restrictions. For each queue of queue type other than network, access may be either unrestricted or restricted. If access is unrestricted, any request may enter the queue. If access is restricted, a request can only enter the queue if the requester or the requester's login group has been given access to that queue (see the **QMGR** description on page 4-9). Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Use **qstat** to determine who has access to a particular queue.

Limits

NQS supports many batch request resource limit types that can be applied to an NQS batch queue. The configurability of these limits allows an NQS manager to set batch queue-specific resource limits which all batch requests in the queue must adhere to.

The syntax of a *limit* in commands of the form

SEt Some_limit = (*limit*) *queue*

is quite flexible.

For finite CPU time limits, the acceptable syntax is as follows:

```
[[hours :] minutes : ] seconds [.milliseconds]
```

Whitespace can appear anywhere between the principal tokens, with the exception that no whitespace can appear around the decimal point.

QSUB *(cont.)*

QSUB *(cont.)*

Example time limit-values are:

```
1234 : 58 : 21.29 - 1234 hrs 58 mins 21.290 secs
12345             - 12345 seconds
121.1             - 121.100 seconds
59:01             - 59 minutes and 1 second
```

For all other finite limits (with the exclusion of the *nice-value*), the acceptable syntax is:

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

where the *integer* and *fraction* tokens represent strings of up to eight decimal digits, denoting the obvious values. In both cases, the *units* of allocation may also be specified as one of the case insensitive strings:

b	- bytes
w	- words
kb	- kilobytes (2^{10} bytes)
kw	- kilowords (2^{10} words)
mb	- megabytes (2^{20} bytes)
mw	- megawords (2^{20} words)
gb	- gigabytes (2^{30} bytes)
gw	- gigawords (2^{30} words)

In the absence of any *units* specification, the units of *bytes* are assumed.

For all limit types with the exception of the *nice-value*, it is possible to state that no limit should be applied. This is done by specifying a *limit* of unlimited, or any initial substring thereof.

The complications caused by batch request resource limits first show up when queueing a batch request in a batch queue. This operation is described in the following paragraphs.

If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type. (See the **QLIMIT** description on page 4-7 to find out what limits are supported by a given machine.)

For each remaining finite limit that can be supported by the underlying UNIX implementation that is not a CPU *time-limit*, or UNIX *nice-value*, the *limit-value* is internally converted to the units of bytes or words, whichever is more appropriate for the underlying machine architecture.

QSUB *(cont.)*

QSUB *(cont.)*

As an example, a *per-process memory size limit value* of 321 megabytes would be interpreted as 321 x 2^{20} bytes, provided that the underlying machine architecture was capable of directly addressing single bytes. Thus the original limit coefficient of 321 would become 321 x 2^{20}. On a machine that was only capable of addressing words, the appropriate conversion of 321 x 2^{20} *bytes / #of-bytes-per-word* would be performed.

If the result of such a conversion would cause overflow when the coefficient was represented as a *signed-long integer* on the supporting hardware, then the coefficient is replaced with the coefficient of: of 2^{N-1} where N is equal to the number of bits of precision in a signed long integer. For typical 32-bit machines, this *default extreme limit* would therefore be 2^{31-1} bytes. For word addressable machines in the supercomputer class supporting 64-bit long integers, the *default extreme limit* would be 2^{63-1} words.

Lastly, some implementations of UNIX reserve coefficients of the form: 2^{N-1} as synonymous with infinity, meaning no limit is to be applied. For such UNIX implementations, NQS further decrements the *default extreme limit* so as to not imply infinity.

The identical internal conversion process as described in the preceding paragraphs is also performed for all *finite limit-values* configured for a particular batch queue using the **qmgr** program.

After all of the applicable *limit-values* have been converted as described above, each such resulting *limit-value* is then compared against the corresponding *limit-value* as configured for the destination batch queue. If, for every type of limit, the batch queue *limit-value* is greater than or equal to the corresponding batch request *limit-value*, then the request can be successfully queued, provided that no other anomalous conditions occur. For request *infinity limit-values*, the corresponding queue *limit-value* must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the **qsub** command, or by the indirect placement of a batch request into a batch queue via a pipe queue. It is impossible for a batch request to be queued in an NQS batch queue if any of these resource limit checks fail.

Finally, if a request fails to specify a *limit-value* for a resource limit type that is supported on the execution machine, then the corresponding *limit-value* configured for the destination queue becomes the *limit-value* for the unspecified request limit.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent **qmgr** commands that alter the limits of the containing batch queue.

QSUB *(cont.)*

QSUB *(cont.)*

Caveats

When an NQS batch request is spawned, a new *process-group* is established such that all processes of the request exist in the same *process-group*. If the **qdel** command is used to send a signal to an NQS batch request, the signal is sent to all processes of the request in the created *process-group*. However, should one or more processes of the request choose to successfully execute a **setpgrp** (2) system call, then such processes will not receive any signals sent by the **qdel** command. This can lead to rogue requests whose constituent processes must be killed by other means such as the **kill**(1) command. However, NQS takes advantage of any UNIX implementations that provide a mechanism of locking a process, and all of its subsequent children in a particular *process-group*. For such UNIX implementations, this problem does not occur.

It is extremely wise for all processes of an NQS request to catch any SIGTERM signals. By default, the receipt of a SIGTERM signal causes the receiving process to die. NQS sends a SIGTERM signal to all processes in the established *process-group* for a batch request as a notification that the request should be prepared to be killed, either because of an *abort queue* command issued by an operator using the **qmgr** program, or because it is necessary to shutdown NQS and all running requests as part of a general shutdown procedure of the local host.

It must be understood that the spawned shell ignores SIGTERM signals. If the current immediate child of the shell does not ignore or catch SIGTERM signals, then it will be killed by the receipt of such, and the shell will go on to execute the next command from the script (if there is one). In any case, the shell will not be killed by the SIGTERM signal, though the executing command will have been killed.

After receiving a SIGTERM signal delivered from NQS, a process of a batch request typically has sixty seconds to get its house in order before receiving a SIGKILL signal (though the sixty second duration can be changed by the operator).

All batch requests terminated because of an operator NQS shutdown request that did not specify the **-nr** flag are considered restartable by NQS, and are requeued (provided that the batch request shell process is still present at the time of the SIGKILL signal broadcast as discussed above), so that when NQS is rebooted, such batch requests will be respawned to continue execution. It is however, up to the user to make the request restartable by the appropriate programming techniques. NQS simply spawns the request again as though it were being spawned for the first time.

Upon completion of a batch request, a mail message can be sent to the submitter (see the discussion of the **-me** flag above). In many instances, the completion code of the spawned Bourne or C-Shell will be displayed. This is merely the value returned by the shell through the **exit** (2) system call.

Lastly, there is no good way to echo commands executed by unmodified versions of the Bourne and C shells. While the C-shell can be spawned in such a fashion as to echo the commands it executes, it is often very difficult to tell an echoed command from genuine output produced by the batch request, because no magic character such as a '$' is displayed in front of the echoed command. The Bourne shell does not support any echo option whatsoever.

Thus, one of the better ways to write the shell script for a batch request is to place appropriate lines in the shell script of the form:

```
echo "explanatory-message"
```

where the echoed message should be a meaningful message chosen by the user.

Limitations and Implementation Notes

Network queues have not yet been implemented.

In the present implementation, it is not possible to see the *stderr* or *stdout* files produced by the batch request while the request is running, unless the **-re** and **-ro** flags have been respectively specified.

Lastly, the strange `@$` syntax used to introduce embedded argument flags was chosen because it rarely conflicts with anything else present in a shell script file. NQS users with better minds will (rightly) suggest improved alternatives to this convention.

See Also

mail, qdel, qdev, qlimit, qmgr, qpr, qstat, kill, setpgrp(), and **signal**()

RECONFIGURING NQS A

INTRODUCTION

NQS is initially installed and configured with the iPSC system software. After the initial configuration steps, the NQS system is functional only on the SRM and only jobs submitted from the SRM can be processed. The procedures in this appendix must be used in order to install NQS on Sun workstations and to configure them to work with the iPSC system.

SETTING UP THE SRM FOR NQS

You should have already performed the automatic NQS configuration steps for the SRM described in the *iPSC®/2 and iPSC®/860 Release 3.3 Software Release Notes*. The procedures described in the *Release Notes* are repeated here for reference.

Perform the following steps at the iPSC SRM:

1. Login as root on the SRM
2. Change directories to the source directory, */usr/src/nqs* as follows:

```
% cd /usr/src/nqs
```

3. Make sure that your umask is set to 0 by entering the following:

```
% umask 0
```

4. Compile NQS code as follows:

```
% make
```

5. Install the NQS executable code by entering the following:

```
% make install
```

Before the NQS daemon can be started, the host machine must be in the NQS data base. Once the NQS daemon is running, the **qmgr** command is functional and you can use the **qmgr** command to create a queue for job requests.

If you want your machine to accept requests from other machines on the network, special care is required. The **NMAPMGR** utility is used to set up and define machine ID's for each remote machine that is to have access to the NQS queue structure. Perform the following steps to set up the SRM.

1. Confirm that the SRM has not already been set up for NQS by entering the following:

```
% ls /etc/nmap
```

If this directory contains any files (and any files named *machines* in particular), then the machine database has already been created for this machine.

CAUTION

Do not create another machine database if one already exits. Only one machine database can exist at a machine, so a new one will overwrite any existing database.

2. Invoke the **NMAPMGR** utility as follows:

```
% nmapmgr
NMPAPMGR>
```

If the machine database already exists for this machine, skip the next step.

3. If it has not been done already, create a machine database on the SRM as follows:

```
NMPAPMGR> create
```

NOTE

The **NMAPMGR create** subcommand must be issued only once at each machine. If the NQS database has already been created for this machine, do not reissue the **create** subcommand.

4. Add the SRM to the newly created machine database. As the NQS manager, one of your jobs is to assign principal names and machine IDs to the machines in this NQS environment. If your SRM has a principal name of **snoopy** and a machine ID of **0**, enter the following subcommand:

```
NAMAPMGR> add mid 0 snoopy
```

5. For each machine in the network that will be submitting requests, add that machine to the NQS machine database. For example, if you are adding a machine named **sun4_112** with machine ID **112** to the NQS system, enter the following subcommand:

```
NAMAPMGR> add mid 112 sun4_112
```

   The machine name should be the primary name as it appears in the host database (*/etc/hosts*).

6. Exit the **NMAPMGR** utility, as follows:

```
NMAPMGR> exit
```

7. Change directories to the **qmgr** *lib* directory:

```
% cd /usr/lib/nqs
```

8. Start the NQS daemon:

```
% ./nqsdaemon > logfile
```

   After starting the daemon, **cat** the logfile. It should be empty. If there are messages in the logfile that indicate that something went wrong when trying to start the daemons, kill the daemons and try again. The daemons for NQS appear in the process list as `nqsdaemon`, `logdaemon`, and `netdaemon`.

9. Invoke **qmgr**, and add the managers for each user you wish to be a queue manager, as follows:

```
% qmgr
MGR>: add manager username:m
```

   Note that you will substitute the actual user name for `username` in the above example. If you only want to grant operator privileges to a new user, add *:o* instead of *:m* to the end of the command line. Refer to the **Add Managers** description on page 4-13 for detailed information on using this subcommand.

10. Create new batch queues to support NQS operations. If you want to create batch queue (for example) named ***new_q*** with the lowest possible priority, you need to perform a minimum of three commands. Create, start, and enable the example batch queue as follows:

```
MGR>: create batch_queue new_q pri=63
MGR>: start queue new_q
MGR>: enable queue new_q
```

11. If you want to create other queues, now is a good time. For example, a pipe queue with the original batch queue as the destination might be set up as follows:

```
MGR>: create pipe_queue new_p_q pri=10
                        server=(/usr/lib/nqs/pqs) dest=new_q
MGR>: start queue new_p_q
MGR>: enable queue new_p_q
```

The new pipe queue has a higher priority (i.e., lower number) than the original batch queue, and it uses a server program at */usr/lib/nqs/pqs*. You could have also used the **Add Destination** subcommand to add the ***new_q*** queue as a destination to another, existing pipe queue.

12. Now exit the **qmgr** environment as follows:

```
MGR>: exit
```

Your are now ready to submit jobs to the NQS system.

SETTING UP A SUN WORKSTATION FOR NQS

The following procedures give the steps you can perform at your Sun workstation to make NQS and the SRM batch queues available to this remote machine. Note that there may be several ways of accomplishing the desired results. The following procedures describe a quick and normally trouble free way of setting up a remote machine for NQS. In the following examples, the remote machine is a Sun 4 workstation.

1. Copy the source files to the workstation, or mount your SRM NQS source directory as an NFS mounted file system. Be sure to include the *../h* directory when copying. The following command will build a **tar** file named *nqs.tar*:

```
% make archive
```

This command creates an archive file of the source files (called *nqs.tar*) in the top level source directory.

2. Once the sources are on the Sun workstation, change directories to the top level source directory on the Sun workstation, and remove the *.o* and *.a* files from the *proto* directory as follows:

```
% make clobber
```

NOTE

The *.a* and *.o* files will only exist if the directory is mounted across the network and **make** has already been run for the SRM. The **make archive** command does not copy the *.a* or *.o* files.

3. Next, save copies of the make files in the *proto* directory as follows:

```
% cd proto
% cp Makefile makef
% cp Makefile.s4 makef.s4
```

4. Move the *Makefile.s4* file to *Makefile* as follows:

```
% mv Makefile.s4 Makefile
```

5. Change directories to the top level source directory, set the umask to 0, and compile the NQS code as follows:

```
% cd ..
% umask 0
% make
```

6. Now login as root, change to the top of the make directory (*make_top* in this example) in use at this Sun workstation, and install NQS as follows:

```
% cd make_top
% make install
```

The following procedures replace the actions that are performed by **`make startup`**. By performing the startup steps by hand, you retain direct control over machine IDs and queue creation. These steps are necessary if requests are to be sent to the SRM.

Perform the following procedures at the remote machine to add each remote machine to the NQS environment:

1. Login as root.

2. Change directories to the NQS command target directory. This directory is identified by the NQS_USREXE symbol in *proto/Makefile.S4* (located at */usr/local/bin* by default).

3. Invoke **NMAPMGR**.

4. Enter **Create** to create the machine database at the remote machine.

5. Use the **Add Mid** subcommand to add the unique machine ID (mid) and name of the remote machine to the database maintained by the remote machine.

NOTE

The machine ID and name at the remote machine database must be identical to the one maintained by the SRM for this machine.

6. Exit the **NMAPMGR** utility.

7. Change directories to the NQS library directory. The path to this directory is defined by the NQS_LIBEXE symbol in *proto/Makefile.S4* (default path is */usr/local/lib/nqs*).

8. Start the nqsdaemon at the remote machine, then **cat** the logfile to ensure the daemon started normally, as follows:

```
% nqsdaemon > logfile
% cat logfile
```

9. Invoke **qmgr**, then use the **Add Manager** subcommand to add the user(s) of the remote machine to the NQS environment as a manager (**:m**) or an operator (**:o**). You must have at least one manager defined at the remote machine, so it is wise to add yourself as a manager.

10. Create, enable, and start the queue structures that are needed at the remote machine end to give the remote machine access to NQS resources. At a minimum, a pipe queue is needed to pass requests to the appropriate batch or device queue in the network.

11. Use the **Set Default Batch_request Queue** subcommand to define the default batch queue used with the **qsub** command. The default batch queue will be used if **qsub** is invoked and the **-q** switch is not used to specify a batch queue.

12. Use the **Set Destination** subcommand where required to add the remote machine as a destination to existing pipe queues. The destination can also be specified when creating the pipe queue.

13. Exit **qmgr**.

The following example session shows the operations performed at a remote machine to add that machine to the NQS environment.

```
% /usr/local/bin/nmapmgr
NMAPMGR>: create
NMAP_SUCCESS:  Successful completion.

NMAPMGR>: add mid 1 io
NMAP_SUCCESS:  Successful completion.

NMAPMGR>: exit

% cd /usr/local/lib/nqs
% nqsdaemon > logfile
% cat logfile
% qmgr

Mgr: add manager sean:m
Mgr: NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.
```

```
Mgr: create batch bq pri=10
Queue bq created.
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: enable queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: start queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: set default batch queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: create pipe pq pri=10 server=(/usr/local/lib/nqs/pipeclient) dest=bq
Queue pq created.
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: enable queue pq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: start queue bq
NQS manager[TCML_COMPLETE  ]: Transaction complete at local host.

Mgr: show long queue
bq@io;  type=BATCH;  [ENABLED, INACTIVE];  pri=10
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
  Run_limit = 1;
  Cumulative system space time = 0.000000 seconds
  Cumulative user space time = 0.000000 seconds
  Unrestricted access
  Per-process core file size limit = 1 megabytes <DEFAULT>
  Per-process data size limit = 1 megabytes <DEFAULT>
  Per-process permanent file size limit = 1 megabytes <DEFAULT>
  Per-process execution nice value = 0 <DEFAULT>
  Per-process stack size limit = 1 megabytes <DEFAULT>
  Per-process CPU time limit = 36000.0 <DEFAULT>
  Per-process working set limit = 1 megabytes <DEFAULT>
```

```
pq@io;  type=PIPE;  [ENABLED, INACTIVE];  pri=10
  0 depart;   0 route;   0 queued;   0 wait;   0 hold;   0 arrive;
  Run_limit = 1;
  Cumulative system space time = 0.000000 seconds
  Cumulative user space time = 0.000000 seconds
  Unrestricted access
  Queue server: /usr/local/lib/nqs/pipeclient
  Destset = {bq@io};

Mgr: exit
%
```

The SRM and remote machines are now ready for operation in the NQS environment. There are additional tasks and housekeeping functions that must be performed to ensure proper network operation, but the NQS environment is minimally functional at this point. The next section describes some of the additional tasks that need to be performed, and also some of the consequences of an improper setup.

ADDITIONAL NQS SETUP CONCERNS

This section includes additional tasks that need to be performed to make a fully functional NQS, and a number of troubleshooting topics that are related to NQS setup.

Using UNIX Mail Facilities

NQS uses the UNIX mail facilities extensively. Any error messages returned by NQS are typically returned to the originating machine using UNIX mail facilities. Refer to the "Setting Up the SRM Mail Service" section on page 3-5 for more information on setting up the UNIX mail facilities for NQS operations.

NQS Local/Remote Machine Identity

The name that is returned by the **/usr/bin/hostname** command is the name that should be the first name entry on the line for this machine in the */etc/hosts* file. This is also the name that should be in the NQS name map database. The **NMAPMGR Add Mid** command must be invoked using this name. There are then three places to verify consistent naming: in the files created by the **NMAPMGR** command, in the */etc/hosts* (or network equivalent) file, and the name reported by the **/usr/bin/hostname** command.

If this local machine is sending or receiving requests to/from other machines on the network, the following additional requirements apply:

1. When the local machine does a name look up in its */etc/hosts* file (or network equivalent) for the remote machine, the name it gets should match both the remote machine's notion of its own name (as reported by the **/usr/bin/hostname** command at the remote machine), and the local machine's NQS database name for the remote machine (as defined in the database files created by the **NMAPMGR** command).

2. Similarly, the remote machine's view of what the local machine's name is must match the local machine's view of its own name.

3. The **NMAPMGR** numbering scheme on the two machines must be identical. In other words, if the local machine thinks it has mid (machine ID) 1 and thinks that the remote machine is mid 2, then the remote machine should agree. The remote machine should call the local machine mid 1 and itself mid 2.

Checking Queue Configuration

Once a queue is created, it will not accept requests until it is both started and enabled. These are two separate operations done from the **qmgr** facility. Enabling a queue makes it possible for the **qsub** command to place requests in the queue. Starting the queue makes is possible for requests to actually be serviced; otherwise they stay in the queue.

With pipe queues, you must check the following additional things:

1. You should check to ensure the destination queue exists, is enabled and is running.

2. You should check to ensure the specified server really exists and is in the (absolute) pathname that was given as the destination parameter in the **qmgr** command. This information can be verified by using **qmgr**. Execute the **Show Long Queue** command. This command will dump all of the data associated with a queue.

3. As mentioned before, the name and the machine ID of the destination machine must be the same as what the destination machine thinks.

Local/Remote *.rhosts* Files

Any user who wishes to send requests over the network to remote machines must have a file called *.rhosts* in the home directory of each machine in the network that may be either a sending or destination machine. The file in the local machine must contain a line for each machine that the local machine will be communicating with. That line consists of a machine name (as known to NQS) and a user name. For example, if user `sean` on machine `multnomah` wants to send a job to machine `pacificr`, then in the home directory of each machine, there should be a *.rhosts* file containing at least the following two lines:

```
multnomah sean
pacificr sean
```

As mentioned in the prior two sections, the machine names (`multnomah` and `pacificr`) must correspond exactly to the names maintained in the NQS databases, and to the names each machine recognizes as its own.

Pipe Queue Servers

When configuring a pipe queue, there must be an associated server program. This server program is designated with the **Server=(*server*)** clause in the **Create Pipe** subcommand in the **qmgr** facility. The purpose of the server is to handle routing of requests that are put on the pipe queue. There is a program that comes with the NQS software package that can be specified as the pipe server. The executable file is called *pipeclient*. This file is in the NQS library directory (*NQS_LIBEXE* in the makefile, */usr/lib/nqs* on the SRM).

The **qmgr** facility will not report an error if the file name is misspelled, if the file is in a different directory, or even if the file doesn't exist at all, so use care when specifying the pipe queue server file. If you incorrectly specify the pipe queue server file, jobs put in the pipe queue will remain there and not be processed.

NQS Troubleshooting Topics

The following paragraphs each describe a possible problem and the cause or suggested corrective action.

NQS doesn't compile.

Make sure you are using the correct *Makefile*. For the SRM, use the original *Makefile*. For a 4.3BSD machine move the file *./proto/Makefile.s4* to *Makefile*. See the "SETTING UP A SUN WORKSTATION FOR NQS" section on page A-4 for detailed information. For other machines, you will have to modify the makefile so the proper preprocessor constants are passed to the compile command.

The compiler works on the makefile, but you can't copy the resulting files.

Check the permissions of the created executable and object files in the *./proto* directory. If the files are not readable by root, check the umask.

You can't add a machine ID to the NQS database.

If you are having trouble with the **NMAPMGR** utility when adding machine IDs, make sure the data base was created with the **Create** command of the **NMAPMGR** utility. As `root`, you can remove all the files in the directory */etc/nmap*, restart the **NMAPMGR** utility, and **Create** a new database.

The *nqsdaemon* or the **qmgr** facility do not startup.

Make sure the **NMAPMGR** database contains an entry for this (local) machine.

Unable to submit a job.
: Make sure the specified job file and destination queue exits. Also make sure the queue is running (started) and enabled.

A job is on the queue forever with `State=queued`.
: Make sure the queue is running. If this is a pipe queue, double check to make sure that the pipe server program is properly identified.

A job is on the queue forever with `State=routing`.
: Make sure the destination queue is valid and enabled. If the destination queue is on another machine, make sure the identities of the sending and receiving machines are consistent throughout.

The job disappears from the queue but no results ever come back.
: Check that the *.rhosts* files are properly set up in the two (sending and destination) machines.

USING THE NMAPMGR UTILITY

The **NMAPMGR** utility is a tool that an NQS manager must use when setting up machines in the NQS environment. NQS refers to a database that holds the names and machine ID numbers of every machine in the NQS environment. If a machine does not have an entry in this NQS database, it is unable to use the NQS resources connected to the network. The NQS database is located in directory */etc/nmap* of each machine in the NQS environment.

Each machine in the NQS environment maintains its own version of the NQS database. Each machine in the system must also have a unique machine ID number and a unique machine name. Once a machine is identified by a specific ID and name, that ID and name must be used everywhere in the NQS environment to identify that machine. For example, a remote machine identified as `johnj` and ID # 112 in the SRM database should have that exact identity in its own database and in the databases of every other NQS machine that may have to communicate with it.

The **NMAPMGR** utility must be used to create the NQS databases on each machine, and also to change the contents of the databases. There is a limited set of subcommands available to you once you invoke the **NMAPMGR** utility. The following subsections give reference information on the subcommands.

NMAPMGR Command Reference

The **NMAPMGR** subcommands are used only when configuring (or reconfiguring) the NQS environment. These subcommands allow you to add or delete machines, users, or groups to/from the database that describes the NQS environment.

NMAPMGR NMAPMGR

NQS network mapping manager program.

Synopsis

nmapmgr

Description

The **NMAPMGR** utility is a program that is used by the System Administrator when initially configuring NQS, or when changing the NQS configuration. The NMAPMGR utility creates, views, and/or changes the network mapping for the NQS environment.

Add Gid

Add Gid *from_mid from_gid to_gid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

from_gid The group ID number (decimal integer) of an existing group for this machine.

to_gid The group ID number (decimal integer) of the group that is being added to this machine.

This subcommand adds the mapping for a group ID number (*to_gid*) to the machine identified by the *from_mid* parameter. The *from_gid* parameter implies an existing GID for every machine in the NQS database. The default UID and GID are stored in the file */usr/lib/nqs/sitename*. You must be logged in as root in order to use this subcommand.

Add Mid

Add Mid *mid principal-name*

mid The machine ID (mid) for the new machine.

principal-name The principal name that is to be used by the new machine.

Create a new machine mapping with the specified machine ID (*mid*) and assign the principal name (*principal-name*) to this new machine. You must be logged in as `root` in order to use this subcommand.

NMAPMGR *(cont.)* NMAPMGR *(cont.)*

Add Name

Add Name *name to_mid*

name The new name that is being added to the machine identified by *to_mid.*

to_mid The machine ID number (decimal integer) to which the new machine name will be added.

Add new mapping for *name* to the machine identified by *to_mid.* You must be logged in as `root` in order to use this subcommand.

Add Uid

Add Uid *from_mid from_uid to_uid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

from_uid The user ID number (decimal integer) of an existing user of this machine.

to_uid The user ID number (decimal integer) of the user that is being added to this machine.

This subcommand adds the mapping for a user ID number (*to_uid*) to the machine identified by the *from_mid* parameter. The *from_uid* parameter implies an existing UID for every machine in the NQS database. The default UID and GID are stored in the file */usr/lib/nqs/sitename*. You must be logged in as `root` in order to use this subcommand.

Change Name

CHange Name *mid new-name*

mid The machine ID (*mid*) for the machine to which this command applies.

new-name The new principal name that is to be used by this machine.

Change the principal name of the machine identified by (*mid*) to the name identified as *new-name*. You must be logged in as `root` in order to use this subcommand.

Create

CReate

This subcommand creates the NQS mapping database on this machine. No **NMAPMGR** subcommands will work until this database has been created. You must be logged in as `root` in order to use this subcommand.

Delete Defgid

Delete DEFGid *from_mid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

Delete and disable the default group ID mapping for the machine identified as *from_mid*. You must be logged in as `root` in order to use this subcommand.

Delete Defuid

Delete DEFUid *from_mid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

Delete and disable the default user ID mapping for the machine identified as *from_mid*. You must be logged in as `root` in order to use this subcommand.

Delete Gid

Delete Gid *from_mid from_gid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

from_gid The group ID number (decimal integer) of the existing group for this machine.

This subcommand deletes a group ID number (*from_gid*) from the machine identified by the *from_mid* parameter. You must be logged in as `root` in order to use this subcommand.

Delete Mid

Delete Mid *mid*

mid The machine ID (*mid*) of the machine to which this subcommand applies.

This subcommand deletes all mappings associated with the machine identified as *mid*. This means that all user ID, group ID, name-to-mid, and mid-to-name mappings for the specified machine ID will be deleted when this subcommand executes. You must be logged in as `root` in order to use this subcommand.

Delete Name

Delete Name *name*

name The principal name of the machine to which this subcommand applies.

Delete the name to machine ID mapping for the machine with principal name *name*. You must be logged in as `root` in order to use this subcommand.

Delete Uid

Delete Uid *from_mid from_uid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

from_uid The user ID for which the machine-to-user mapping is being requested.

This subcommand deletes a user ID number (*from_uid*) from the machine identified by the *from_mid* parameter. You must be logged in as `root` in order to use this subcommand.

NMAPMGR *(cont.)* NMAPMGR *(cont.)*

Disable Mid

Disable Mid *from_mid wakeuptime*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

wakeuptime The time (a standard UNIX long integer) after which this subcommand is no longer in effect.

This subcommand disables user and group mapping between the machine identified as *from_mid* and the local machine until the *wakeuptime* is reached. You must be logged in as `root` in order to use this subcommand.

Enable Mid

Enable Mid *from_mid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

This subcommand enables user and group mapping to the machine identified as *from_mid*. You must be logged in as `root` in order to use this subcommand.

Exit

Exit

Leave the **NMAPMGR** command environment. Issuing the **Quit** subcommand or pressing the **<CTRL-D>** keys will have the same effect.

NMAPMGR *(cont.)* NMAPMGR *(cont.)*

Get Gid

Get Gid *from_mid from_gid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

from_gid The group ID for which the machine-to-group mapping is being requested.

Show the machine-to-group mapping from the machine identified as *from_mid* to the group identified as *from_gid*.

Get Mid

Get Mid *name*

name The principal name of the machine to which this subcommand applies.

Determine the machine ID of the machine whose principal name is *name*.

Get Name

Get Name *mid*

mid The machine ID number (decimal integer) to which this subcommand applies.

Return the principal host name for the machine specified as *mid*.

Get Uid

Get Uid *from_mid from_uid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

from_uid The user ID for which the machine-to-user mapping is being requested.

Show the machine-to-user mapping from the machine identified as *from_mid* to the user identified as *from_uid*.

Help

Help

Typing ***help*** while the **NMAPMGR** utility is active results in the following display:

```
Commands:
  Add Uid <from_mid> <from_uid> <to_uid>
  Add Gid <from_mid> <from_gid> <to_gid>
  Add Name <name> <to_mid>
  Add Mid <mid> <principal-name>
  CHange Name <mid> <new-name>
  CReate
  Delete Uid <from_mid> <from_uid>
  Delete Gid <from_mid> <from_gid>
  Delete DEFUid <from_mid>
  Delete DEFGid <from_mid>
  Delete Name <name>
  Delete Mid <mid>
  Exit
  Get Uid <from_mid> <from_uid>
  Get Gid <from_mid> <from_gid>
  Get Mid <name>
  Get Name <mid>
  Help
  Quit
  Set Nfds <#-of-file-descrs>
  Set DEFUid <from_mid> <defuid>
  Set DEFGid <from_mid> <defgid>
 ^D (to exit nmapmgr.)
```

Quit

Quit

Leave the **NMAPMGR** command environment. Issuing the **Exit** subcommand or pressing the **<CTRL-D>** keys will have the same effect.

NMAPMGR *(cont.)* NMAPMGR *(cont.)*

Set Defgid

Set DEFGid *from_mid defgid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

defgid The group ID that is to be used as the default group ID for this machine.

This command enables and sets the default group *defgid* ID mapping for the *from_mid* machine. Any default group ID mapping for this machine is replaced by this command. You must be logged in as `root` in order to use this subcommand.

Set Defuid

Set DEFUid *from_mid defuid*

from_mid The machine ID number (decimal integer) to which this subcommand applies.

defuid The user ID that is to be used as the default user ID for this machine.

This command enables and sets the default user ID (*defuid*) mapping for the *from_mid* machine. Any default user ID mapping for this machine is replaced by this command. You must be logged in as `root` in order to use this subcommand.

Set Nfds

Set Nfds *#-of-file-descrs*

#-of-file-descrs The number of file descriptors (0, 1, or 2) used to control file usage by the **NMAPMGR** command.

This is an internal command that affects the processing time of the **NMAPMGR** command.

NMAPMGR Error Messages

The following error messages may be returned to indicate the success or failure of any of the **NMAPMGR** subcommands.

`Integer overflow.`
The maximum supported integer size was exceeded. For the machine ID, the mid cannot exceed 13 decimal digits in size.

`Invalid NMAP return code.`
The value returned by one of the **NMAPMGR** procedures was not one of the valid values.

`Missing arguments.`
One or more of the arguments to the **NMAPMGR** procedure was missing. All arguments to **NMAPMGR** procedures are required. There are no optional arguments.

`Missing <mid>.`
The machine ID argument required by the **NMAPMGR** procedure was missing.

`Missing <name>.`
The principal name of the machine required by the **NMAPMGR** procedure was missing.

`Missing verb modifier.`
One of the command verbs (Add, Change, Delete, Get, or Set) was not followed by a modifier such as Mid or Uid.

`NMAP_DEFMAP: Successful completion using default mapping.`
No mapping existed for the specified mid/uid(gid) pair, but the default uid(gid) mapping was enabled, and so the default uid(gid) mapping for the specified machine has been returned.

`NMAP_EBADNAME: Null name or name too long.`
A null name or a name too long for mapping was specified.

`NMAP_ECONFLICT: Already exists.`
A mapping already existed for the source target which defined a mapping result target different from the new target.

`NMAP_ENOMAP: No such mapping.`
No mapping exists for the specified machine ID and remote user ID or remote group ID pair.

`NMAP_ENOMID: No such machine.`
There is no such machine ID in the current database.

`NMAP_ENOPRIV: No privilege for operation.`
The current user has insufficient privilege to perform the prior operation on the database.

`NMAP_EUNEXPECT: Fatal error in mapping software.`
This message is returned if an unanticipated error occurred in the mapping software.

`NMAP_SUCCESS: Successful completion.`
Return message if the prior operation was successful.

`NMAPMGR command line too long.`
The command line exceeded the maximum line size supported for **NMAPMGR**.

`NMAPMGR error reading command input.`
An error was detected in the command line.

`No such mid.`
No mapping was found for the stated machine ID.

`Unrecognized command verb: ...`
The stated command verb was not one of the recognized command verbs (Add, Change, Create, Delete, Exit, Get, Help, Quit, or Set).

`Unrecognized command verb modifier.`
One of the command verbs (Add, Change, Delete, Get, or Set) was not followed by a recognized modifier such as Mid or Uid.

`Unsigned integer number expected.`
Signed integers are not supported.

GLOSSARY

This Glossary contains definitions of terms that apply to the NQS utility and the hardware and software environment in which it operates.

arriving request state
Requests in this state are in the process of being queued from another (possibly remote) *pipe queue*. After leaving this state, the request will enter the *waiting*, *queued*, *running*, or *routing* state.

batch queue A *batch queue* holds requests for scheduled, perhaps delayed, processing by various subsystems in the NPSN.

daemon A *daemon* is a process which is designed to run continuously, providing some service when needed.

device An NQS *device* is a site at which a shared serial-access resource such as a printer is offered.

device queue A *device queue* holds requests for resources such as printers and Computer Output Microfilm (COM) units.

forms A *forms* type is a set of parameters (for example, margins and font specifications for a printer) that determine operating characteristics for a *device*.

grandfathered limits
Requests that have been queued are allowed to complete regardless of their limits.

local daemon The *local daemon* processes requests executing on the local machine. It manages the batch queues and schedules all requests for execution. It also detects and forwards any requests that are to be executed on a remote machine.

log daemon The *log daemon* records a log of NQS requests and activities.

manager An NQS *manager* identifies a person who is capable of changing any NQS characteristic on the local machine.

network daemon
The *network daemon* handles all remote requests. It is essentially the same as the *local daemon* that forwards requests, but it receives its requests through Berkley sockets from remote machines.

network queue A *network queue* is a specialized form of a *pipe queue* that passes NQS requests between machines, and also performs queued file transfer operations.

operator An NQS *operator* identifies a person who can execute only the *operator commands* as a proper subset of all the commands provided by the *qmgr*(1m) utility.

pipe queue A *pipe queue* is a queue which can pass queued requests on to other *pipe queues* , *batch queues* , or *device queues.*

queued request state
The request has been accepted in a queue, and is completely ready to enter the *running* or *routing* request state.

request An NQS request is a *request* by a user or user program to perform a function that requires a delay in servicing (e.g., after a certain time). Examples of such functions are the scheduling of a shared serial-access resource (e.g., a printer), and the scheduling of batch job requests.

routing request state
A request residing in a *pipe queue* is being routed and delivered to another queue destination.

running request state
A request residing in a *batch* or *device queue* is currently being executed.

server A *server* is a program that is always spawned to handle a request that is given to a device for execution.

waiting request state
The request is waiting for some finite time interval to pass. After leaving this state, the request will enter the *queued, running,* or *routing* state.

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W