User's Guide

1 Introduction

CODINE (Computing in Distributed Networked Environments) is a *load management* tool for heterogeneous, distributed computing environments.CODINE provides an effective method for distributing the batch workload among multiple computational servers. In doing so, it increases the productivity of all of the machines and simultaneously increases the number of jobs that can be completed in a given time period. Also, by increasing the productivity of the workstations, the need for outside computational resources is reduced.

CODINE provides the user with the means to submit computationally demanding task to the CODINE system for transparent distribution of the associated workload. In addition to batch jobs, interactive jobs and parallel jobs can be submitted to CODINE. Checkpointing programs are also supported. Checkpointing jobs migrate from workstation to workstation without user intervention on load demand. Comprehensive tools are provided for the monitoring and controlling of CODINE jobs.

Please refer to the CODINE Quick Start Guide for an overview on the CODINE system, its features and components. The CODINE Quick Start Guide also contains a quick installation procedure for a small sample CODINE configuration and a glossary of terms commonly used in the CODINE manual set.

The CODINE User's Guide gives an introduction for the user to CODINE. The reader is pointed to the CODINE Reference Manual for a detailed discussion of all available CODINE commands. Readers responsible for the cluster administration are pointed to the CODINE Installation and Administration Guide for a description of the CODINE cluster management facilities.

CODINE as well as UNIX Commands which can be found in manual pages or the corresponding reference manuals are typeset in *emphasized* font throughout the CODINE User's Guide. Command-line in- and output is printed in `teletype` font and newly introduced or defined terms are typeset in **boldface** font.

2 CODINE User Types and Operations

There are four user categories in CODINE:

- **Managers:**
  Managers have full capabilities to manipulate CODINE. By default, the superusers of any machine hosting a queue have manager privileges.
- **Operators:**
  The operators can perform the same commands as the manager with the exception of adding/deleting/modifying queues.
- **Owners:**
  The queue owners are allowed to suspend/enable the owned queues, but have no further management permissions.
- **Users:**
  Users have certain access permissions as described in “User Access Permissions” on page 188 but no cluster or queue management capabilities. The following table adjoins CODINE command capabilities to the different user categories:

Table 4: CODINE Command Capabilities and User Categories

Command	Manager	Operator	Owner	User
qacct	Full	Full	Own jobs only	Own jobs only
qalter	Full	Full	Own jobs only	Own jobs only
qconf	Full	No modifications to the system setup	Show configurations and access permissions only	Show configurations and access permissions only
qdel	Full	Full	Own jobs only	Own jobs only
qhold	Full	Full	Own jobs only	Own jobs only
qhost	Full	Full	Full	Full
qlogin	Full	Full	Full	Full
qmod	Full	Full	Own jobs and owned queues only	Own jobs only

Table 4: CODINE Command Capabilities and User Categories

Command	Manager	Operator	Owner	User
qmon	Full	No modifications to the system setup	No configuration changes	No configuration changes
qrexec	Full	Full	Full	Full
qselect	Full	Full	Full	Full
qsh	Full	Full	Full	Full
qstat	Full	Full	Full	Full
qsub	Full	Full	Full	Full

3 Navigating through the CODINE System

3.1 Overview on Host Functionality

The `Host Configuration` button in the *qmon* main menu allows you to retrieve an overview on the functionality which is associated with the hosts in your CODINE cluster. However, unless you do not have CODINE manager privileges, you may not apply any changes to the presented configuration.

The host configuration dialogues are described in the CODINE Installation and Administration Guide in section “CODINE Daemons and Hosts” on page 60.

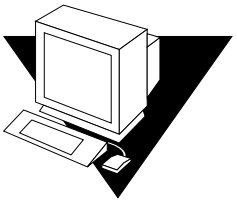
The subsequent sections provide the commands to retrieve this kind of information from the command-line.

3.1.1 The Master Host

The location of the master host should be transparent for the user as the master host may migrate between the current master host and one of shadow master hosts at any time. The file `<codine_root>/<cell>/common/act_qmaster` contains the name of the current master host for the CODINE cell `<cell>`.

3.1.2 Execution Hosts

To display information about the hosts being configured as execution hosts in your cluster please use the commands

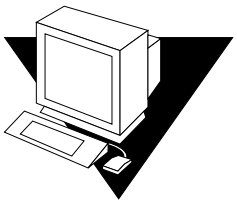


```
% qconf -sel
% qconf -se hostname
% qhost
```

The first command displays a list of the names of all hosts being currently configured as execution hosts. The second command displays detailed information about the specified execution host. The third command displays status and load information about the execution hosts. Please refer to the *host_conf* manual page for details on the information displayed via *qconf* and to the *qhost* manual page for details on its output and further options.

3.1.3 Administration Hosts

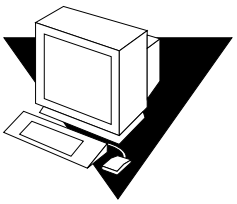
The list of hosts with administrative permission can be displayed with the command



```
% qconf -sh
```

3.1.4 Submit Hosts

The list of submit host can be displayed with the command



```
% qconf -ss
```

3.2 Queues and Queue Properties

In order to be able to optimally utilize the CODINE system at your site, you should become familiar with the queue structure and the properties of the queues which are configured for your CODINE system.

3.2.1 The Queue Control qmon Dialogue

The *qmon* queue control dialogue displayed and described in section “Controlling Queues with qmon” on page 258 provides a quick overview on the installed queues and their current status.

3.2.2 Show Properties with the qmon Object Browser

The *qmon* object browser can be used in combination with the queue control dialogue to display the pertinent queue property information. The object browser is opened upon clicking on the `Browser` icon button in the *qmon* main menu. By selecting the Queue button and moving the mouse pointer over a queue icon in the queue control dialogue, queue property information is displayed in a similar way as described in the *queue_conf* manual page

The following figure shows an object browser example display with a queue property print-out.

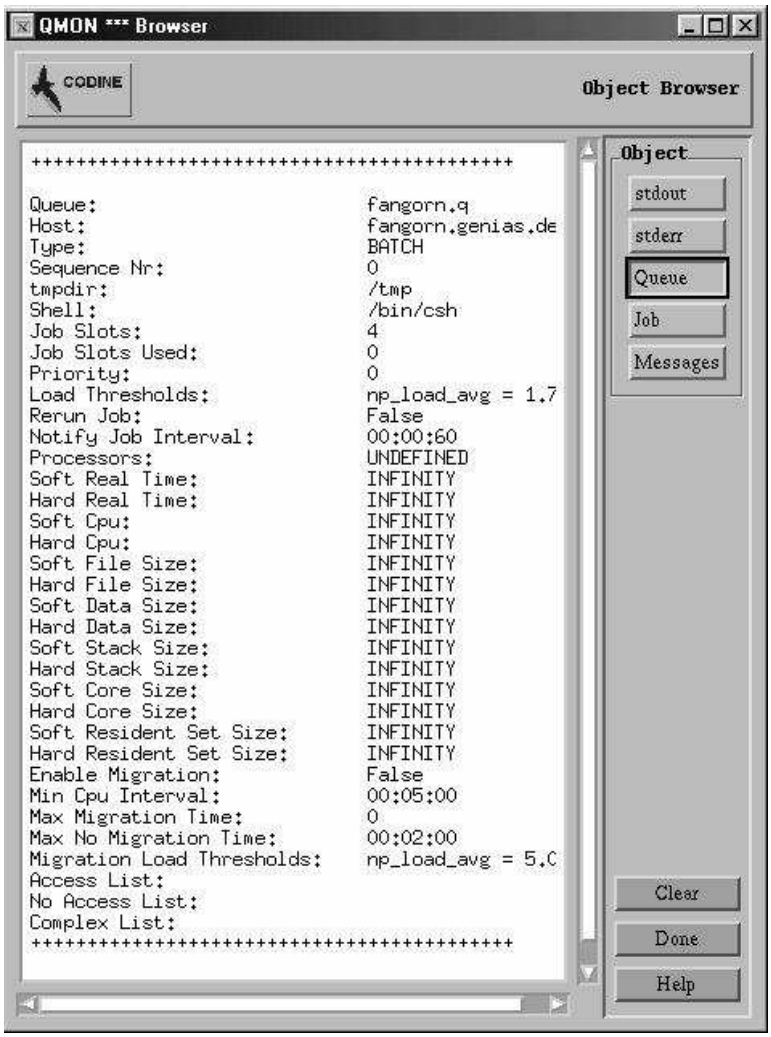
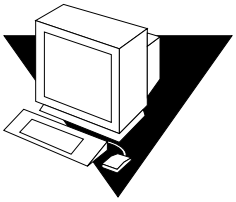


Figure 57: `Browser` queue output

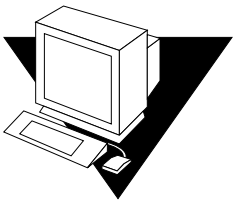
3.2.3 Queue Information from the Command-line

In order to display a list of currently configured queues use the



```
% qconf -sql
```

command. To display the properties of a particular queue please execute



```
% qconf -sq queue_name
```

A detailed description of each property can be found in the *queue_conf* manual page (see section 5 of the CODINE Reference Manual). Here is a short introduction to the most important parameters:

- **qname:**
  The queue name as requested.
- **hostname:**
  The host of the queue.
- **processors:**
  The processors of a multi processor system, to which the queue has access.
- **qtype:**
  The type of job which is allowed to run in this queue. Currently, this is either batch, interactive, checkpointing, parallel or any combination thereof or transfer alternatively
- **slots:**
  The number of jobs which may be executed concurrently in that queue.
- **owner_list:**
  The owners of the queue as explained in section “Managers,

Operators and Owners” on page 190

- **user_lists:**
  The user or group identifiers in the user access lists (see “User Access Permissions” on page 188) enlisted under this parameter may access the queue.
- **xuser_lists**
  The user or group identifiers in the user access lists (see “User Access Permissions” on page 188) enlisted under this parameter may *not* access the queue.
- **complex_list**
  The complexes enlisted under this parameter are associated with the queue and the attributes contained in these complexes contribute to the set of requestable attributes for the queue (see “Requestable Attributes” on page 184).
- **complex_values**
  Assigns capacities as provided for this queue for certain complex attributes (see “Requestable Attributes” on page 184).

3.3 Requestable Attributes

When submitting a CODINE job a requirement profile of the job can be specified. The user can specify attributes or characteristics of a host or queue which the job requires to run successfully. CODINE will map these job requirements onto the host and queue configurations of the CODINE cluster and will, therefore, find the suitable hosts for a job.

The attributes which can be used to specify the job requirements are either related to the CODINE cluster (e.g. space required on a network shared disk), to the hosts (e.g. operating system architecture), to the queues (e.g. permitted CPU time) or the attributes are derived from site policies such as the availability of installed software only on some hosts.

The available attributes include the queue property list (see “Queues and Queue Properties” on page 181), the list of global and host related attributes (see “Complex Types” on page 97 of the CODINE Installation and Administration Guide) as well as administrator defined attributes. For convenience, however, the CODINE administrator commonly chooses to define only a subset of all available attributes to be requestable.

The attributes being currently requestable are displayed in the `Requested Resources` sub-dialogue (see figure 58 on page 185) to the *qmon* Submit dialogue (please refer to section "Submit Batch Jobs" on page 192 for detailed information on how to submit jobs). They are enlisted in the `Available Resources` selection list.

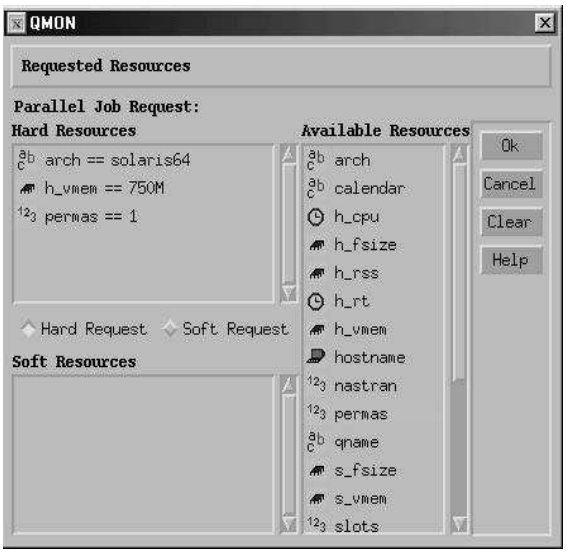
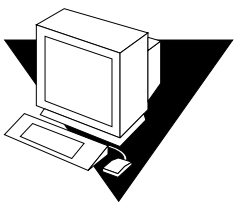


Figure 58: `Requested Resources` dialogue

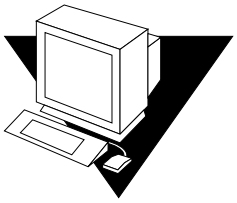
To display the list of requestable attributes from the command-line, you first have to display the list of currently configured **complexes** with the command



```
% qconf -scl
```

A so called complex contains the definition for a set of attributes. There are three standard complexes: `global` (for the cluster global attributes), `host` (for the host specific attributes and `queue` (for the queue property attributes). Any further complex names printed if the above command is executed refers to an administrator defined complex (see “The Complexes Concept” on page 95 in the CODINE Installation and Administration Guide or the complex format description in the section 5 of the CODINE Reference Manual for more information on complexes).

To display the attributes of a particular complex please execute



```
% qconf -sc complex_name[,...]
```

The output for the queue complex might for example look as shown in table 5 on page 186.

Table 5: “queue” complex

#name	shortcut	type	value	relop	requestable	consumable	default
#----------							
qname	q	STRING	NONE	==	YES	NO	NONE
hostname	h	HOST	unknown	==	YES	NO	NONE
tmpdir	tmp	STRING	NONE	==	NO	NO	NONE
calendar	c	STRING	NONE	==	YES	NO	NONE
priority	pr	INT	0	>=	NO	NO	0
seq_no	seq	INT	0	==	NO	NO	0
rerun	re	INT	0	==	NO	NO	0
s_rt	s_rt	TIME	0:0:0	<=	NO	NO	0:0:0
h_rt	h_rt	TIME	0:0:0	<=	YES	NO	0:0:0

Table 5: "queue" complex

s_cpu	s_cpu	TIME	0:0:0	<=	NO	NO	0:0:0
h_cpu	h_cpu	TIME	0:0:0	<=	YES	NO	0:0:0
s_data	s_data	MEMORY	0	<=	NO	NO	0
h_data	h_data	MEMORY	0	<=	YES	NO	0
s_stack	s_stack	MEMORY	0	<=	NO	NO	0
h_stack	h_stack	MEMORY	0	<=	NO	NO	0
s_core	s_core	MEMORY	0	<=	NO	NO	0
h_core	h_core	MEMORY	0	<=	NO	NO	0
s_rss	s_rss	MEMORY	0	<=	NO	NO	0
h_rss	h_rss	MEMORY	0	<=	YES	NO	0
min_cpu_interval	mci	TIME	0:0:0	<=	NO	NO	0:0:0
max_migr_time	mmt	TIME	0:0:0	<=	NO	NO	0:0:0
max_no_migr	mnm	TIME	0:0:0	<=	NO	NO	0:0:0

#--- # starts a comment but comments are not saved across edits ---

The column `name` is basically identical to the first column displayed by the *qconf -sq* command. The queue attributes cover most of the CODINE queue properties. The `shortcut` column contains administrator definable abbreviations for the full names in the first column. Either the full name or the shortcut can be supplied in the request option of a *qsub* command by the user.

The column `requestable` tells whether the Corresponding entry may be used in *qsub* or not. Thus the administrator can, for example, disallow the cluster's users to request certain machines/queues for their jobs directly, simply by setting the entries `qname` and/or `qhostname` to be not requestable. Doing this, implies that feasible user requests can be met in general by multiple queues, which enforces the load balancing capabilities of CODINE.

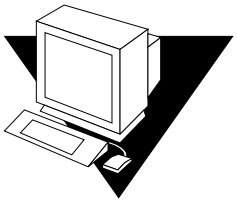
The column `relop` defines the relation operation used in order to compute whether a queue meets a user request or not. The comparison executed is

User_Request relop Queue/Host/...-Property

If the result of the comparison is false, the user's job cannot be run in the considered queue. Let, as an example, the queue *q1* be configured with a soft cpu time limit (see the *queue_conf* and the *setrlimit* manual pages for a description of user process limits) of 100 seconds while the queue *q2* is configured to provide 1000 seconds soft cpu time limit.

The columns `consumables` and `default` are meaningful for the administrator to declare so called consumable resources (see section "Consumable Resources" on page 105 of the CODINE Installation and Administration Guide). The user requests consumables just like any other attribute. The CODINE internal bookkeeping for the resources is however different.

Now, let a user submit the following request:



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

The *s_cpu=0:5:0* request (see the *qsub* manual page for details on the syntax) asks for a queue which at least grants for 5 minutes of soft limit cpu time. Therefore, only queues providing at least 5 minutes soft CPU runtime limit are setup properly to run the job.

☞ **CODINE will only consider workload information in the scheduling process if more than one queue is able to run a job.**

3.4 User Access Permissions

Access to queues and other CODINE facilities (e.g. parallel environment interfaces - see section „Parallel Jobs“ on page 219) can be restricted for certain users or user groups by the CODINE administrator.

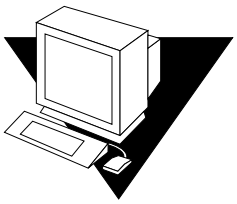
☞ **CODINE automatically takes into account the access restrictions configured by the cluster administration. The following sections are only important if you want to query your personal access permission.**

For the purpose of restricting access permissions, the administrator creates and maintains so called access lists (or in short **ACL**s). The ACLs contain arbitrary user and UNIX group names. The ACLs are then added to **access-allowed-** or **access-denied-lists** in the queue or in the parallel environment interface configurations (see *queue_conf* or *codine_pe* in CODINE Reference Manual section 5, respectively).

User‘s belonging to ACLs which are enlisted in access-allowed-lists have permission to access the queue or the parallel environment interface. User‘s being members of ACLs in access-denied-lists may not access the concerning resource.

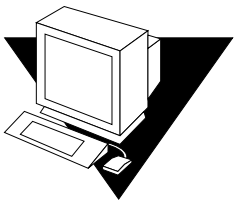
The `Userset Configuration` dialogue opened via the `User Configuration` icon button in the *qmon* main menu allows you to query for the ACLs you have access to via the `Userset Configuration` dialogue. Please refer to the section “Managing User Access” on page 126 of the CODINE Installation and Administration Guide for details.

From the command-line a list of the currently configured ACLs can be obtained by the command:



```
% qconf -sul
```

The entries in one or multiple access lists are printed with the command:



% qconf -su acl_name[,...]

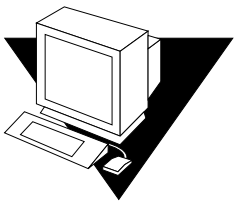
The ACLs consist of user account names and UNIX group names with the UNIX group names being identified by a prefixed "`@`" sign. This way you can determine to which ACLs your account belongs.

☞ **In case you have permission to switch your primary UNIX group with the *newgrp* command, your access permissions may change.**

You can now check for those queues or parallel environment interfaces to which you have access or to which access is denied for you. Please query the queue or parallel environment interface configuration as described in "Queues and Queue Properties" on page 181 and "Configuring PEs with qmon" on page 158 in the CODINE Installation and Administration Guide. The access-allowed-lists are named `user_lists`. The access-denied-list have the names `xuser_lists`. If your user account or primary UNIX group is associated with a access-allowed-list you are allowed to access the concerning resource. If you are associated with a access-denied-list you may not access the queue or parallel environment interface. If both lists are empty every user with a valid account can access the concerning resource.

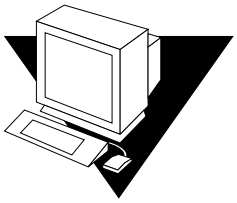
3.5 Managers, Operators and Owners

A list of CODINE managers can be obtained by



```
% qconf -sm
```

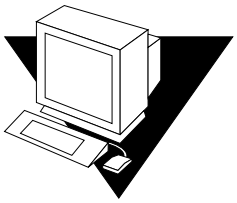
and a list of operators by



```
% qconf -so
```

☞ **The superuser of a CODINE administration host is considered as manager by default.**

The users, which are owners to a certain queue are contained in the queue configuration database as described in section “Queues and Queue Properties” on page 181. This database can be retrieved by executing



```
% qconf -sq queue_name
```

The concerning queue configuration entry is called `owners`.

4 Submit Batch Jobs

4.1 Shell Scripts

Shell scripts, also called batch jobs, are in principal a sequence of UNIX command-line instructions assembled in a file. Script files are made executable by the UNIX *chmod* command. If scripts are invoked, a proper command interpreter is started (e.g. *csh*, *tcsh*, *sh*, or *ksh*) and each instruction is interpreted as typed in manually by the user executing the script. Arbitrary UNIX commands, applications and other shell scripts can be invoked from within a shell script.

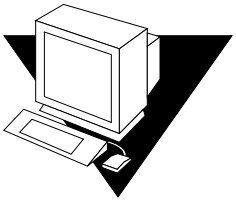
The appropriate command interpreter is either invoked as **login-shell** or not depending whether its name (*csh*, *tcsh*, *sh*, *ksh*, ...) is contained in the value list of the **login_shells** entry of the CODINE configuration in effect for the particular host and queue executing the job.

☞ **Note, that the CODINE configuration may be different for the various hosts and queues configured in your cluster. You can display the effective configurations via the *-sconf* and *-sq* options of the *qconf* command (refer to the CODINE Reference Manual for detailed information).**

If the command interpreter is invoked as login-shell, the environment of your job will be exactly the same as if you just have logged-in and executed the job-script. In case of using *csh* for example, `.login` and `.cshrc` will be executed in addition to the system default start-up resource files (e.g. something like `/etc/login`) while only `.cshrc` will be executed if *csh* is not invoked as login-shell. Refer to the manual page of the command interpreter of your choice for a description of the difference between being invoked as login-shell or not.

4.1.1 Example Script File

Below is the listing of a simple shell script, which first compiles the application `flow` from its Fortran77 source and then executes it.



```
#!/bin/csh
# This is a sample script file for compiling and
# running a sample FORTRAN program under CODINE.
cd TEST
# Now we need to compile the program 'flow.f' and
# name the executable 'flow'.
f77 flow.f -o flow
# Once it is compiled, we can run the program.
flow
# End of script file.
```

Your local system user's guide will provide detailed information about building and customizing shell scripts (you might also want to look at the *sh*, *ksh, csh* or *tcsh* manual page). In the following, the CODINE User's Guide will emphasize on specialities which are to be considered in order to prepare batch scripts for CODINE.

In general, all shell scripts that you can execute from your command prompt by hand can be submitted to CODINE as long as they do not require a terminal connection (except for the standard error and output devices, which are automatically redirected) and as long as they do not need interactive user intervention. Therefore, the script given above is ready to be submitted to CODINE and will perform the desired action.

4.2 Submitting CODINE jobs

4.2.1 Submitting jobs with qmon (Simple Example)

The *qmon* `Job Submission` dialogue is either invoked from the *qmon* main menu or from the *qmon* job control dialogue. Pressing the `Submit` icon button in the *qmon* main menu opens the dialogue as well as pushing the `Submit` button in the `Job Control` dialogue. The screen for entering `General` parameters looks as follows (see section „Submitting Jobs with qmon (Advanced Example)“ on page 200 for a discussion of the `Advanced` parameter screen).

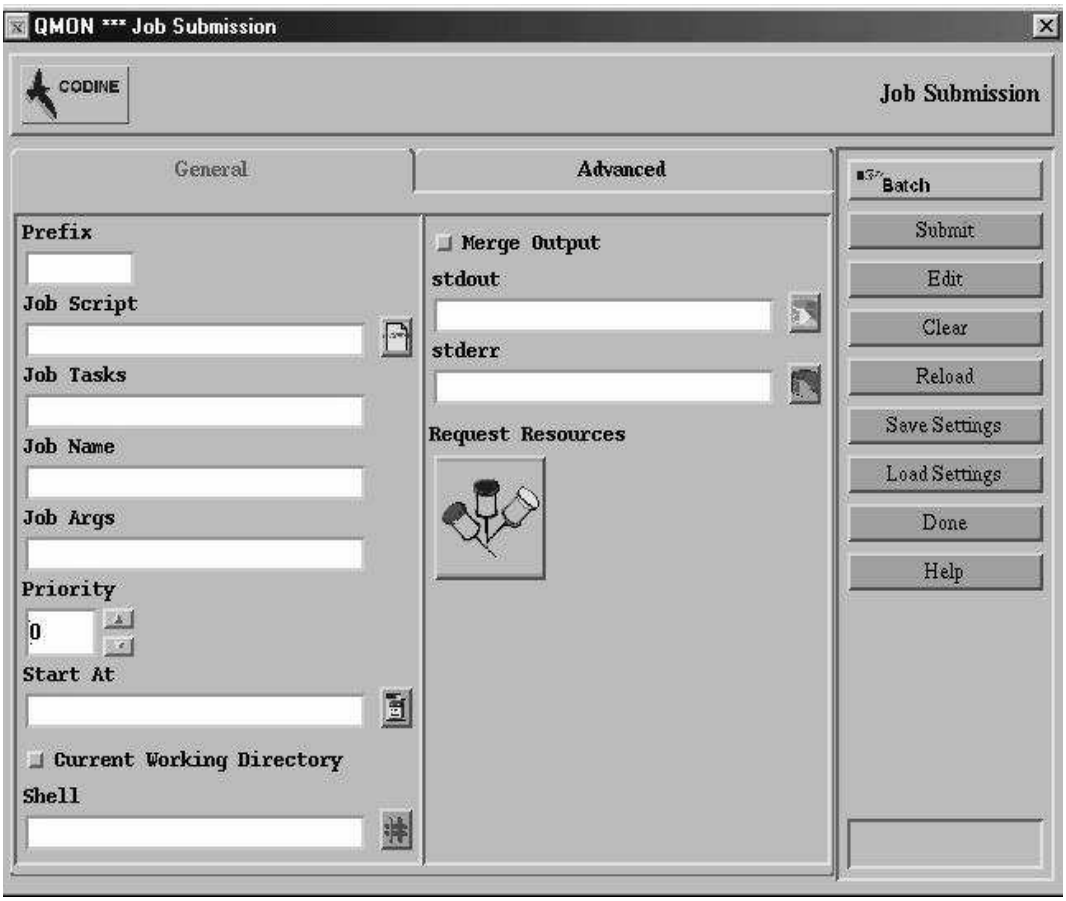


Figure 59: `Job Submission` dialogue

Throughout section 4 “Submit Batch Jobs” we will only deal with batch jobs. So please make sure that the default `Batch` icon is displayed on the top of the button column on the right side of the screen. If an `Interactive` icon is displayed instead, please click to the icon to change it back to the Batch icon. Please refer to section “Submit Interactive Jobs” on page 224 for detailed information on interactive jobs.

To submit a job you first have to select its script file. Use the file icon button on the right side of the `Job Script` input window to open the following file selection box and to select the job's script file.

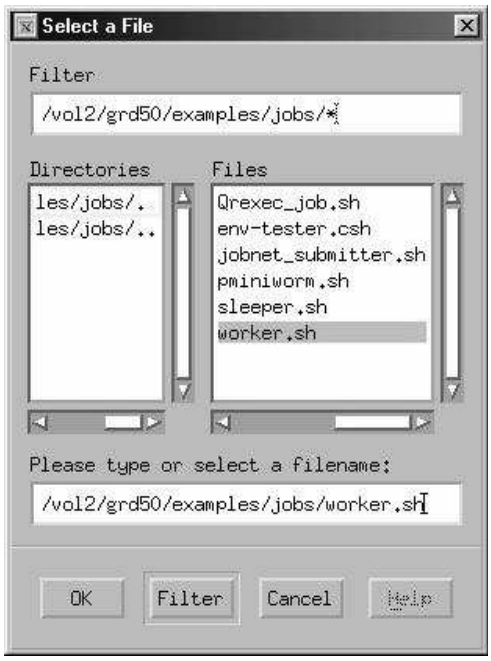


Figure 60: Job script selection box

Quitting the file selection dialogue with the `OK` button will transfer the selected file name to the `Job Submission` dialogue's `Job Script` input window. Now just click to the `Submit` button on the right side of the Job Submission screen to submit the job to the CODINE system.

☞ **To get immediate feedback from the job submission you either need to have the *qmon* `Job Control` dialogue open (see section „Monitoring and Controlling Jobs with qmon" on page 242) or you need the *qmon* `Object Browser` opened with the display messages facility activated (see section „Additional Information with the qmon Object Browser" on page 252).**

4.2.2 Submitting jobs with qmon (Extended Example)

The standard form of the `Job Submission` dialogue (see figure 71 on page 226) provides the means to configure the following parameters for a job:

- A prefix string which is used for script embedded CODINE submit options (please refer to section “Active CODINE Comments:” on page 207 for detailed information).
- The job script to be used. If the associated icon button is pushed, a file selection box is opened (see figure 60 on page 195)
- The task ID range for submitting array jobs (see “Array Jobs” on page 217).
- The name of the job (a default is set after a job script is selected).
- Arguments to the job script.
- The job’s initial priority value. Users without manager or operator permission may only lower their initial priority value.
- The time at which the job is to be considered eligible for execution. If the associated icon button is pushed, a helper dialogue for entering the correctly formatted time is opened (see figure 61 on page 197)
- A flag indicating whether the job is to be executed in the current working directory (for identical directory hierarchies between the submit and the potential execution hosts only).
- The command interpreter to be used to execute the job script (see “How a Command Interpreter Is Selected” on page 205). If the associated icon button is pushed. a helper dialogue for entering he command interpreter specifications of the job is opened (see figure 62 on page 198).
- A flag indicating whether the job’s standard output and standard error output are to be merged together into the standard output stream.
- The standard output redirection to be used (see “Output Redirection” on page 206). A default is used if nothing is specified. If the associated icon button is pushed, a helper dialogue for entering the output redirection alternatives (“Output redirection box” on page 198).

- ❏ The standard error output redirection to be used. Very similar to the standard output redirection.
- ❏ The resource requirements of the job (see “Resource Requirement Definition” on page 213). If resources are requested for a job, the icon button changes its color.
- ❏ A selection list button defining whether the job can be restarted after being aborted by a system crash or similar events and whether the restart behavior depends on the queue or is demanded by the job.
- ❏ A flag indicating whether the job is to be notified by SIGUSR1 or SIGUSR2 signals respectively if it is about to be suspended or cancelled.
- ❏ A flag indicating that either a user hold or a job dependency is to be assigned to the job. The job is not eligible for execution as long as any type of hold is assigned to it (see section „Monitoring and Controlling CODINE Jobs“ on page 242 for more information concerning holds). The input field attached to the Hold flag allows restricting the hold to only a specific range of task of an array job (see “Array Jobs” on page 217).
- ❏ A flag forcing the job to be either started immediately if possible or being rejected. Jobs are not queued, if this flag is selected.

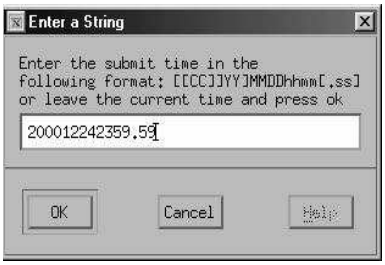


Figure 61: At time input box

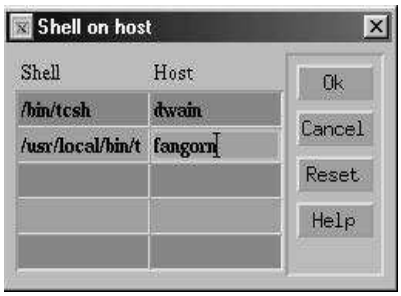


Figure 62: Shell selection box

Figure 63: Output redirection box

The buttons at the right side of the `Job Submission` screen allow you to initiate various actions:

- **Submit**
  Submit the job as specified in the dialogue
- **Edit**
  Edit the selected script file in an X-terminal either using *vi* or the editor as defined in the `$EDITOR` environment variable.
- **Clear**
  Clear all settings in the `Job Submission` dialogue including any specified resource requests.
- **Reload**
  Reload the specified script file, parse any script embedded options (see section „Active CODINE Comments:“ on page 207), parse default settings (see section „Default Requests“ on page 212) and discard intermediate manual changes to these settings. This action is the equivalent to a Clear action with subsequent specifications of the previous script file The option

will only show an effect if a script file is already selected.

- **Save Settings**
  Save the current settings to a file. A file selection box is opened to select the file. The saved files may either explicitly be loaded later-on (see below) or may be used as default requests (see section „Default Requests“ on page 212).
- **Load Settings**
  Load settings previously saved with the `Save Settings` button (see above). The loaded settings overwrite the current settings.
- **Done**
  Closes the `Job Submission` dialogue.
- **Help**
  Dialogue specific help.

Figure “Job submission example” on page 200 shows the submit dialogue with most of the parameters set. The job configured in the example has the script file `flow.sh` which has to reside in the working directory of *qmon*. The job is called `Flow` and the script file takes the single argument `big.data`. The job will be started with priority `-111` and is eligible for execution not before midnight of the 24th of December in the year 2000. The job will be executed in the submission working directory and will use the command interpreter *tcsh*. Finally standard output and standard error output will be merged into the file `flow.out` which will be created in the current working directory also.

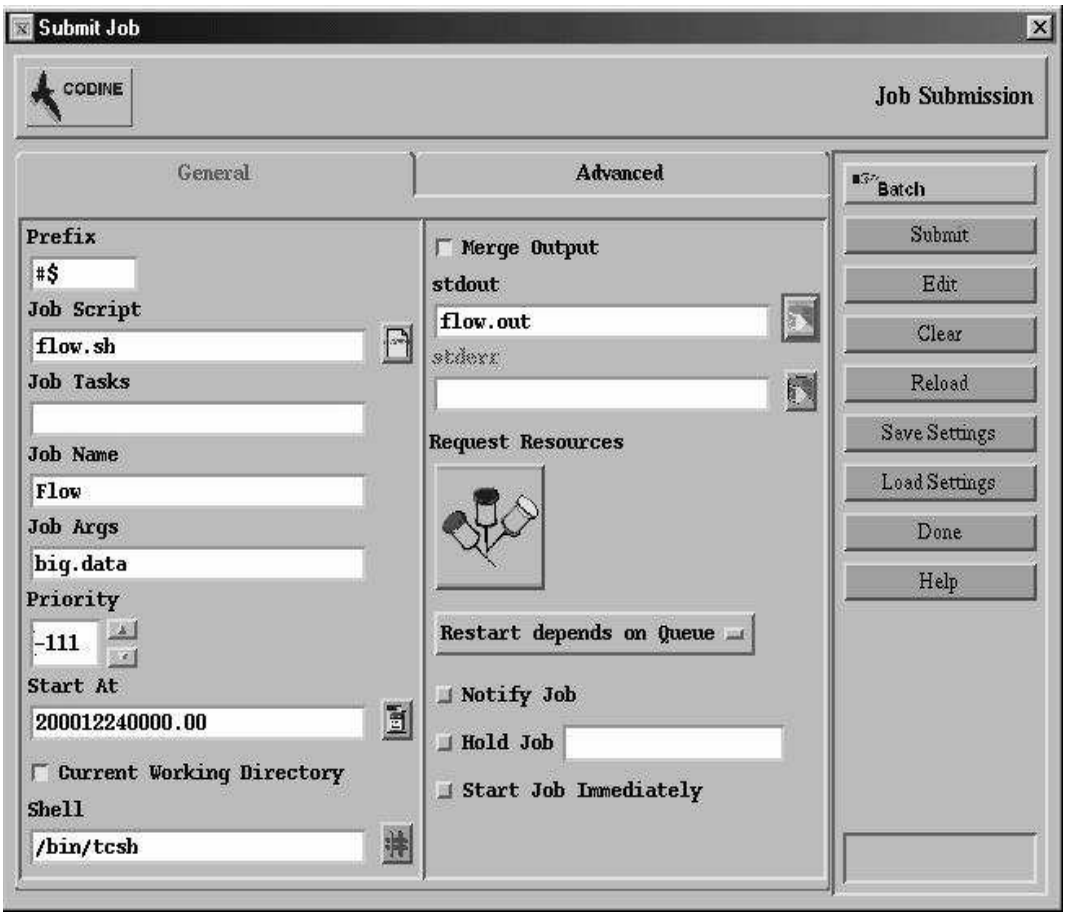


Figure 64: Job submission example

4.2.3 Submitting Jobs with qmon (Advanced Example)

The Advanced submission screen allows definition of the following additional parameters:

- A parallel environment interface to be used and the range of processes which is required (see section „Parallel Jobs“ on page 219).
- A set of environment variables which are to be set for the job before it is executed. If the associated icon button is pushed, a helper dialogue for the definition of the environment variables to

be exported is opened (see figure 65 on page 202). Environment variables can be taken from *qmon*'s runtime environment or arbitrary environment variable can be defined.

- A list of name/value pairs called `Context` (see figure 66 on page 203), which can be used to store and communicate job related information accessible anywhere from within a CODINE cluster. Context variables can be modified from the command-line via the *-ac/-dc/-sc* options to *qsub*, *qsh*, *qlogin* or *qalter* and can be retrieved via *qstat -j*.
- The checkpointing environment to be used in case of a job for which checkpointing is desirable and suitable (see section „Checkpointing Jobs“ on page 237).
- An account string to be associated with the job. The account string will be added to the accounting record kept for the job and can be used for later accounting analysis.
- The `Verify` flag, which determines the consistency checking mode for your job. To check for consistency of the job request CODINE assumes an empty and unloaded cluster and tries to find at least one queue in which the job could run. Possible checking modes are:
  - `Skip` - no consistency checking at all.
  - `Warning` - inconsistencies are reported, but the job is still accepted (may be desired if the cluster configuration is supposed to change after submission of the job).
  - `Error` - inconsistencies are reported and the job will be rejected if any are encountered.
  - `Just verify` - The job will not be submitted, but an extensive report is generated about the suitability of the job for each host and queue in the cluster.
- The events at which the user is notified via electronic mail. The events start/end/abortion/suspension of job are currently defined.
- A list of electronic mail addresses to which these notification mails are sent. If the associated icon button is pushed, a helper dialogue to define the mailing list is opened (see figure 67 on page 203).

- A list of queue names which are requested to be the mandatory selection for the execution of the job. The `Hard Queue List` is treated identical to a corresponding resource requirement as described in “Resource Requirement Definition” on page 213.
- A list of queue names which are requested to be a desirable selection for the execution of the job. The `Soft Queue List` is treated identical to a corresponding resource requirement as described in “Resource Requirement Definition” on page 213.
- A list of queue names which are eligible as so called *master queue* for a parallel job. A parallel job is started in the master queue. All other queues to which the job spawns parallel tasks are called *slave queues*.
- An argument list which is forwarded directly to the submission client of a foreign queuing system, in case the job is executed under the CODINE QSI (see section „The CODINE Queuing System Interface (QSI)“ on page 166 in the CODINE Installation and Administration Guide). The `Transfer QS Arguments` have no effect if the job executed within the CODINE system.
- An ID-list of jobs which need to be finished successfully before the job to be submitted can be started. The newly created job **depends** on successful completion of those jobs.

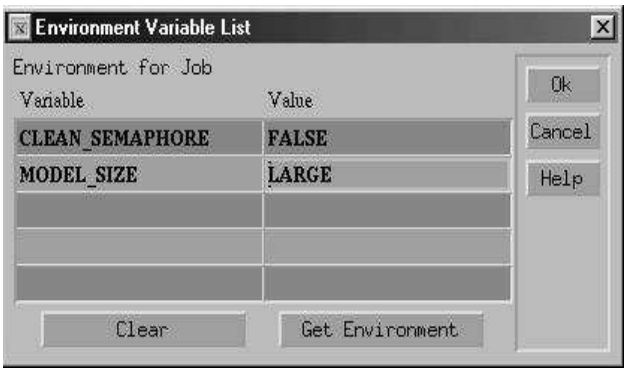


Figure 65: Job environment definition

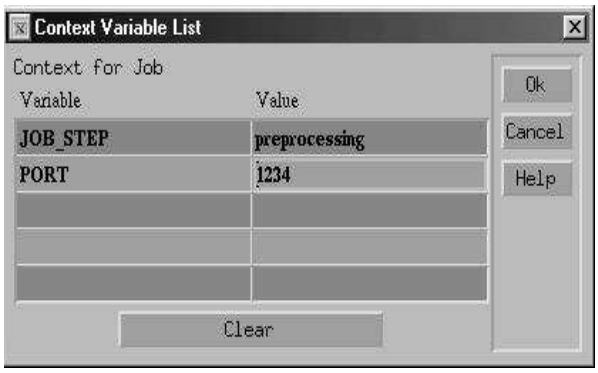


Figure 66: Job context definition

Figure 67: Mail address specification

Consequently, the job defined in figure 68 on page 205 has the following additional characteristics as compared to the job definition from section "Submitting jobs with qmon (Extended Example)" on page 196:

- The job requires the use of the parallel environment `mpi`. It needs at least 4 parallel processes to be created and can utilize up to 16 processes if available.
- Two environment variables are set and exported for the job.

- ❏ Two context variables are set.
- ❏ The account string `FLOW` is to be added to the job accounting record.
- ❏ The job is to be restarted if it fails in case of a system crash.
- ❏ Warnings should be printed if inconsistencies between the job request and the cluster configuration are detected
- ❏ Mail has to be sent to a list of two e-mail addresses as soon as the job starts and finishes.
- ❏ Preferably, the job should be executed in the queue `big_q`.

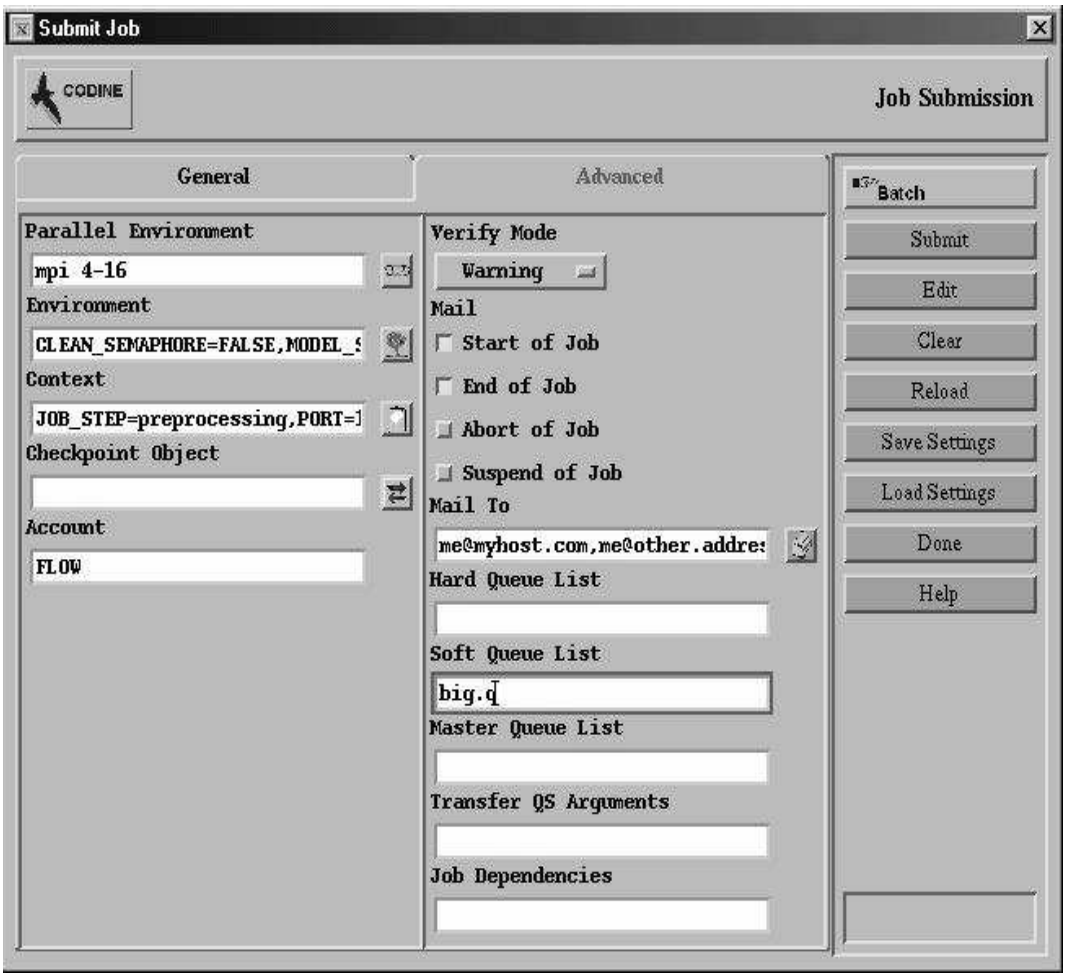


Figure 68: Advanced job submission example

4.2.4 Extensions to Regular Shell Scripts

There are some extensions to regular shell scripts, that will influence the behavior of the script if running under CODINE control. The extensions are:

- **How a Command Interpreter Is Selected**
  The command interpreter to be used to process the job script file can be specified at submit time (see for example page 198).

However, if nothing is specified, the configuration variable `shell_start_mode` determines how the command interpreter is selected:

- If `shell_start_mode` is set to `unix_behavior`, the first line of the script file if starting with a „#!“ sequence is evaluated to determine the command interpreter. If the first line has no „#!“ sequence, the Bourne-Shell *sh* is used by default.
- For all other settings of `shell_start_mode` the default command interpreter as configured with the `shell` parameter for the queue in which the job is started is used (see section „Queues and Queue Properties“ on page 181 and the *queue_conf* manual page).

❑ **Output Redirection**

Since batch jobs do not have a terminal connection their standard output and their standard error output has to be redirected into files. CODINE allows the user to define the location of the files to which the output is redirected, but uses defaults if nothing is specified.

The standard location for the files is in the current working directory where the jobs execute. The default standard output file name is `<Job_name>.o<Job_id>`, the default standard error output is redirected to `<Job_name>.e<Job_id>`. `<Job_name>` is either built from the script file name or can be defined by the user (see for example the *-N* option in the *qsub* manual page). `<Job_id>` is a unique identifier assigned to the job by CODINE.

In case of array job tasks (see section „Array Jobs“ on page 217), the task identifier is added to these filenames separated by a dot sign. Hence the resulting standard redirection paths are `<Job_name>.o<Job_id>.<Task_id>` and `<Job_name>.e<Job_id>.<Task_id>`.

In case the standard locations are not suitable, the user can specify output directions with *qmon* as shown in figure 68 and figure 63 or with the *-e* and *-o qsub* options. Standard output and standard error output can be merged into one file and the redirections can be specified on a per execution host basis. I.e., depending on the host on which the job is executed, the location of the output redirection files becomes different. To build custom but unique redirection file paths, pseudo environment

variables are available which can be used together with the *qsub -e* and *-o* option

- `$HOME` - home directory on execution machine.
- `$USER` - user ID of job owner.
- `$JOB_ID` - current job ID.
- `$JOB_NAME` - current job name (see -N option).
- `$HOSTNAME` - name of the execution host.
- `$TASK_ID` - array job task index number.

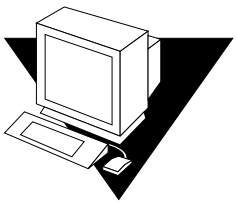
These variables are expanded during runtime of the job into the actual values and the redirection path is built with them.

See the *qsub* manual page in section 1 of the CODINE Reference Manual for further details.

❑ **Active CODINE Comments:**
Lines with a leading "#" sign are treated as comments in shell scripts. CODINE, however, recognizes special comment lines and uses them in a special way: the rest of such a script line will be treated as if it were part of the command line argument list of the CODINE submit command *qsub*. The *qsub* options supplied within these special comment lines are also interpreted by the *qmon* submit dialogue and the corresponding parameters are preset when a script file is selected.

The special comment lines per default are identified by the „#$" prefix string. The prefix string can be redefined with the *qsub -C* option.

The described mechanism is called script embedding of submit arguments. The following example script file makes use of script embedded command-line options.



```
#!/bin/csh
#Force csh if not CODINE default shell
#$ -S /bin/csh
# This is a sample script file for compiling and
# running a sample FORTRAN program under CODINE.
# We want CODINE to send mail when the job begins
# and when it ends.
#$ -M EmailAddress
#$ -m b,e
# We want to name the file for the standard output
# and standard error.
#$ -o flow.out -j y
# Change to the directory where the files are located.
cd TEST
# Now we need to compile the program 'flow.f' and
# name the executable 'flow'.
f77 flow.f -o flow
# Once it is compiled, we can run the program.
flow
# End of script file.
```

- **Environment Variables:**
  When a CODINE job is run, a number of variables are preset into the job's environment, as listed below
  - ARC: The CODINE architecture name of the node on which the job is running. The name is compiled-in into the *cod_execd* binary.
  - CODINE_ROOT: The CODINE root directory as set for *cod_execd* before start-up or the default `/usr/CODINE`.
  - COD_CELL: The CODINE cell in which the job executes.
  - COD_O_HOME: The home directory path of the job owner on the host from which the job was submitted.
  - COD_O_HOST: The host from which the job was submitted.

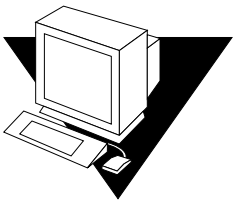
- COD_O_LOGNAME: The login name of the job owner on the host from which the job was submitted.
- COD_O_MAIL: The content of the MAIL environment variable in the context of the job submission command.
- COD_O_PATH: The content of the PATH environment variable in the context of the job submission command.
- COD_O_SHEL: The content of the SHELL environment variable in the context of the job submission command.
- COD_O_TZ: The content of the TZ environment variable in the context of the job submission command.
- COD_O_WORKDIR: The working directory of the job submission command.
- COD_CKPT_ENV: Specifies the checkpointing environment (as selected with the *qsub -ckpt* option) under which a checkpointing job executes.
- COD_CKPT_DIR: Only set for checkpointing jobs. Contains path ckpt_dir (see the *checkpoint* manual page) of the checkpoint interface.
- COD_STDERR_PATH: the pathname of the file to which the standard error stream of the job is diverted. Commonly used for enhancing the output with error messages from prolog, epilog, parallel environment start/stop or checkpointing scripts.
- COD_STDOUT_PATH: the pathname of the file to which the standard output stream of the job is diverted. Commonly used for enhancing the output with messages from prolog, epilog, parallel environment start/stop or checkpointing scripts.
- COD_TASK_ID: The task identifier in the array job represented by this task.
- ENVIRONMENT: Always set to BATCH. This variable indicates, that the script is run in batch mode.
- HOME: The user's home directory path from the *passwd* file.
- HOSTNAME: The hostname of the node on which the job is running.
- JOB_ID: A unique identifier assigned by the *cod_qmaster* when the job was submitted. The job ID is a decimal integer

in the range to 99999.

- JOB_NAME: The job name, built from the *qsub* script filename, a period, and the digits of the job ID. This default may be overwritten by *qsub -N*.
- LAST_HOST: The name of the preceding host in case of migration of a checkpointing job.
- LOGNAME: The user's login name from the *passwd* file.
- NHOSTS: The number of hosts in use by a parallel job.
- NQUEUES: The number of queues allocated for the job (always 1 for serial jobs)
- NSLOTS: The number of queue slots in use by a parallel job.
- PATH: A default shell search path of: `/usr/local/bin:/usr/ucb:/bin:/usr/bin`
- `PE`: The parallel environment under which the job executes (for parallel jobs only).
- `PE_HOSTFILE`: The path of a file containing the definition of the virtual parallel machine assigned to a parallel job by CODINE. See the description of the **$pe_hostfile** parameter in *codine_pe* for details on the format of this file. The environment variable is only available for parallel jobs.
- QUEUE: The name of the queue in which the job is running.
- REQUEST: The request name of the job, which is either the job script filename or is explicitly assigned to the job via the *qsub -N* option.
- RESTARTED: Indicates, whether a checkpointing job has been restarted. If set (to value 1), the job has been interrupted at least once and is thus restarted.
- SHELL: The user's login shell from the *passwd* file. Note: This is not necessarily the shell in use for the job.
- TMPDIR: The absolute path to the job's temporary working directory.
- TMP: The same as TMPDIR; provided for compatibility with NQS.
- TZ: The time zone variable imported from *cod_execd* if set.
- USER: The user's login name from the *passwd* file.

4.2.5 Submitting Jobs from the Command-line

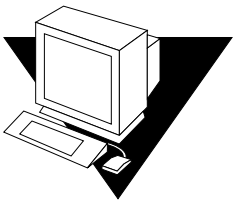
Jobs are submitted to CODINE from the command-line using the *qsub* command (see the corresponding CODINE Reference Manual section). A simple job as described in section "Submitting jobs with qmon (Simple Example)" on page 193 could be submitted to CODINE with the command



```
% qsub flow.sh
```

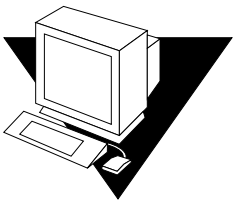
if the script file name is `flow.sh`.

As opposed to this, the submit command which would yield the equivalent to the *qmon* job submission described in section "Submitting jobs with qmon (Extended Example)" on page 196 would look as follows:



```
% qsub -N Flow -p -111 -a 200012240000.00 -cwd \
        -S /bin/tcsh -o flow.out -j y flow.sh big.data
```

Further command-line options can be added to constitute more complex requests. The job request from section "Submitting Jobs with qmon (Advanced Example)" on page 200, for example, would look as follows:



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

4.2.6 Default Requests

The last example in the above section demonstrates that advanced job requests may become rather complex and unhandy, in particular if similar requests need to be submitted frequently. To avoid the cumbersome and error prone task of entering such command-lines, the user can either embed *qsub* options in the script files (see “Active CODINE Comments:” on page 207) or can utilize so called **default requests**.

The cluster administration may setup a default request file for all CODINE users. The user, on the other hand, can create a private default request file located in the user‘s home directory as well as application specific default request files located in the working directories.

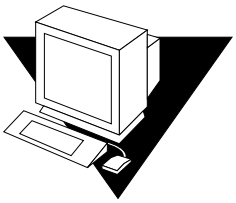
Default request files simply contain the *qsub* options to be applied by default to the CODINE jobs in a single or multiple lines. The location of the cluster global default request file is `<codine_root>/<cell>/common/cod_request`. The private general default request file is located under `$HOME/.cod_request`, while the application specific default request files are expected under `$cwd/.cod_request`.

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

- ❏ Global default request file.
- ❏ General private default request file.
- ❏ Application specific default request file.

- **Script embedding and the *qsub* command-line has higher precedence than the default request files. Thus, script embedding overwrites default request file settings, and the *qsub* command-line options my overwrite these settings again.**
- **The *qsub -clear* option can be used at any time in a default request file, in embedded script commands and in the *qsub* command-line to discard any previous settings.**

An example private default request file is presented below:



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

Unless overwritten, for all jobs of the given user the account string would be `myproject`, the jobs would execute in the current working directory, mail notification would be sent at the beginning and end of the jobs to `me@myhost.com`, the jobs are to be restarted after system crashes, the standard output and standard error output are to be merged and the *ksh* is to be used as command interpreter.

4.2.7 Resource Requirement Definition

In the examples so far the submit options used did not express any requirements for the hosts on which the jobs were to be executed. CODINE assumes that such jobs can be run on any host. In practice, however, most jobs require certain prerequisites to be satisfied on the executing host in order to be able to complete successfully. Such prerequisites are enough available memory, required software to be installed or a certain operating system architecture. Also, the cluster administration usually imposes restrictions on the usage of the machines in the cluster. The CPU time allowed to be consumed by the jobs is often restricted, for example.

CODINE provides the user with the means to find a suitable host for the user‘s job without a concise knowledge of the cluster‘s equipment and its utilization policies. All the user has to do is to specify the requirement of the user‘s jobs and let CODINE manage the task of finding a suitable and lightly loaded host.

Resource requirements are specified via the so called requestable attributes explained in section “Requestable Attributes” on page 184. A very convenient way of specifying the requirements of a job is provided by *qmon*. The `Requested Resources` dialogue, which is opened upon pushing the `Requested Resources` icon button in the `Job Submission` dialogue (see for example figure 68 on page 205) only displays those attributes in the `Available Resource` selection list which currently are eligible. By double-clicking to an attribute, the attribute is added to the `Hard` or `Soft` (see below) `Resources` list of the job and (except for `BOOLEAN` type attributes, which are just set to „`True`“) a helper dialogue is opened to guide the user in entering a value specification for the concerning attribute.

The example `Requested Resources` dialogue displayed below in figure 58 shows a resource profile for a job in which a `solaris64` host with an available `permas` license offering at least 750 Megabytes of memory is requested. If more than one queue fulfilling this specification is found, any defined soft resource requirements are taken into account (none in our example). However, if no queue satisfying both the hard and the soft requirements is found, any queue granting the hard requirements is considered to be suitable.

☞ **Only if more than one queue is suitable for a job, load criteria determine where to start the job.**

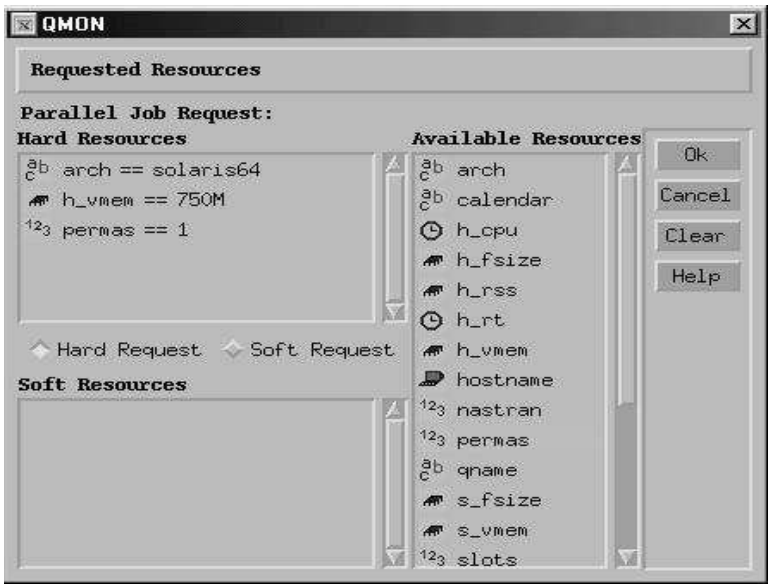
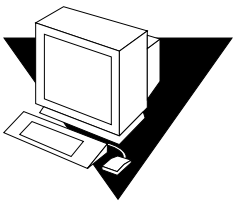


Figure 69: `Requested Resources` dialogue

☞ **The INTEGER attribute `permas` is introduced via an administrator extension to the "global" complex, the STRING attribute `arch` is imported from the "host" complex while the MEMORY attribute `h_vmem` is imported from the "queue" complex (see section „Requestable Attributes" on page 184)**

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



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

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

The notation *750M* for 750 Megabytes is an example for the CODINE quantity syntax. For those attributes requesting a memory consumption you can specify either integer decimal, floating point decimal, integer octal and integer hexadecimal numbers appended by the so called multipliers:

- **k**
  multiplies the value by 1000.
- **K**
  multiplies the value by 1024.
- **m**
  multiplies the value by 1000 times 1000.
- **M**
  multiplies the value by 1024 times 1024.

Octal constants are specified by a leading 0 (zero) and digits ranging from 0 to 7 only. Specifying a hexadecimal constant requires to prepend the number by 0x and to use digits ranging from 0 to 9, a to f and A to F. If no multipliers are appended the values are considered to count as bytes. If using floating point decimals, the resulting value will be truncated to an integer value.

For those attributes imposing a time limit one can specify the time values in terms of hours, minutes or seconds and any combination. The hours, minutes and seconds are specified in decimal digits separated by colons. A time of `3:5:11` is translated to 11111 seconds. If a specifier for hours, minutes or seconds is 0 it can be left out if the colon remains. Thus a value of :5: is interpreted as 5 minutes. The form used in the `Requested Resources` dialogue above is an extension, which is only valid within *qmon*.

4.3 How CODINE Allocates Resources

As shown in the last section, it is important for the user to know, how CODINE processes resource requests and how resources are allocated by CODINE. The following provides a schematic view of CODINE's resource allocation algorithm:

Read in and parse all default request files (see section „Default Requests“ on page 212). Process the script file for embedded options (see section „Active CODINE Comments:“ on page 207). All script embedding options are read, when the job is submitted regardless of their position in the script file. Now read and parse all requests from the command line.

As soon as all qsub requests are collected, **Hard** and **soft** requests are processed separately (the **hard** first). The requests are evaluated Corresponding to the following order of precedence:

- from left to right of the script/default request file
- from top to bottom of the script/default request file
- from left to right of the command line

In other words, the command line can be used to override the embedded flags.

The resources requested **hard** are allocated. If a request is not valid, the submit is rejected. If one or more requests cannot be met at submit-time (e.g. a requested queue is busy) the job is spooled and will be re-scheduled at a later time. If all **hard** requests can be met, they are allocated and the job can be run.

The resources requested **soft** are checked. The job can run even if some or all of these requests cannot be met. If multiple queues (already meeting the hard requests) provide parts of the soft resources list (overlapping or different parts) CODINE will select the queues offering the most soft requests.

The job will be started and will cover the allocated resources.

It is useful to gather some experience on how argument list options and embedded options or **hard** and **soft** requests influence each other by experimenting with small test scriptfiles executing UNIX commands like *hostname* or *date*.

4.4 Array Jobs

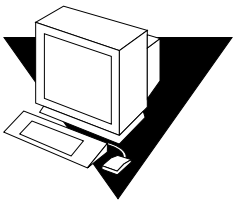
Parametrized and repeated execution of the same set of operations (contained in a job script) is an ideal application for the CODINE **array job** facility. Typical examples for such applications are found in the Digital Content Creation industries for tasks like rendering. Computation of an animation is split into frames, in this example, and the same rendering computation can be performed for each frame independently.

The array job facility offers a convenient way to submit, monitor and control such applications. CODINE, on the other hand, provides an efficient implementation of array jobs, handling the computations as an array of independent tasks joined into a single job. The tasks of an array job are referenced through an array index

number. The indices for all tasks span an index range for the entire array job which is defined during submission of the array job by a single qsub command.

An array job can be monitored and controlled (e.g. suspended, resumed or cancelled) as a total or by individual task or subset of tasks, in which case the corresponding index numbers are suffixed to the job ID to reference the tasks. As tasks execute (very much like regular jobs), they can use the environment variable `$COD_TASK_ID` to retrieve their own task index number and to access input data sets designated for this task identifier.

The following is an example of how to submit an array job from the command-line:



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

The *-t* option defines the task index range. In this case, *2-10:2* specifies that *2* is the lowest and *10* is the highest index number while only every second index (the *:2* part of the specification) is used. Thus the array job consists of 5 tasks with the task indices 2, 4, 6, 8, and 10. Each task requests a hard CPU time limit of *45* minutes (the *-l* option) and will execute the job script *render.sh* once being dispatched and started by CODINE. The tasks can use `$COD_TASK_ID` to find out whether they are task 2, 4, 6, 8, or 10 and they can use their index number to find their input data record in the data file *data.in*.

The submission of array jobs from the GUI *qmon* works identically to how it was described in previous chapters. The only difference is, that the Job Tasks input window shown in figure 64 on page 200 needs to contain the task range specification with the identical syntax as for the *qsub -t* option. Please refer to the qsub manual page in the CODINE Reference Manual for detailed information on the array index syntax.

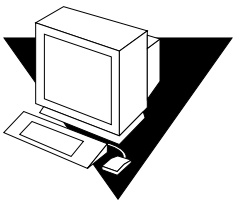
The sections 8 “Monitoring and Controlling CODINE Jobs” and 8.5 “Controlling CODINE Jobs from the Command-line” as well as the CODINE Reference Manual sections about *qstat*, *qhold*, *qrls*, *qmod*, and *qdel* contain the pertinent information about monitoring and controlling CODINE jobs in general and array jobs in particular.

☞ **Array jobs offer full access to all CODINE facilities known for regular jobs. In particular they can be parallel jobs at the same time or can have interdependencies with other jobs.**

4.5 Parallel Jobs

CODINE provides means to execute parallel jobs using arbitrary message passing environments such as PVM or MPI (see the **PVM User’s Guide** and the **MPI User’s Guide** for details) or shared memory parallel programs on multiple slots in single queues or distributed across multiple queues and (for distributed memory parallel jobs) across machines. An arbitrary number of different **parallel environment** (**PE**) interfaces may be configured concurrently at the same time.

The currently configured PE interfaces can be displayed with the commands:



```
% qconf -spl
% qconf -sp pe_name
```

The first command prints a list of the names of the currently available PE interfaces. The second command displays the configuration of a particular PE interface. Please refer to the *codine_pe* manual page for details on the PE configuration.

Alternatively, the PE configurations can be queried with the *qmon* `Parallel Environment Configuration` dialogue (see section „Configuring PEs with qmon“ on page 158 in the CODINE Installation and Administration Guide). The dialogue is opened upon pushing the `PE Config` icon button in the *qmon* main menu.

The example from section “Submitting Jobs with qmon (Advanced Example)” on page 200 already defines a parallel job requesting the PE interface `mpi` (for *message passing interface*) to be used with at least 4 but up to (and preferably) 16 processes. The icon button to the right of the parallel environment specification window can be used to pop-up a dialogue box to select the desired parallel environment from a list of available PEs (see figure 70). The requested range for the number of parallel tasks initiated by the job can be added after the PE name in the PE specification window of the advanced submission screen.

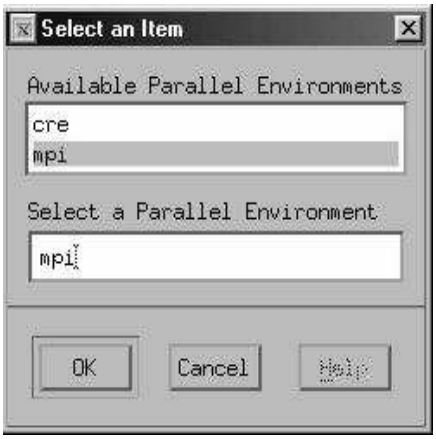


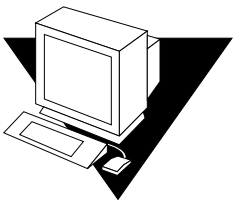
Figure 70: PE selection

The command-line submit command corresponding to the parallel job specification described above is given in section “Submitting Jobs from the Command-line” on page 211 and shows how the *qsub -pe* option has to be used to formulate an equivalent request. The *qsub* manual page in the CODINE Reference Manual provides more detail on the *-pe* syntax.

It is important to select a suitable PE interface for a parallel job. PE interfaces may utilize no or different message passing systems, they may allocate processes on single or multiple hosts, access to the PE

may be denied to certain users, only a specific set of queues may be used by a PE interface and only a certain number of queue slots may be occupied by a PE interface at any point of time. You should therefore ask the CODINE administration for the available PE interface(s) best suited for your type(s) of parallel jobs.

You can specify resource requirements as explained in section “Resource Requirement Definition” on page 213 together with your PE request. This will further reduce the set of eligible queues for the PE interface to those queues also fitting the resource requirement definition you specified. If, for example, the command



```
% qsub -pe mpi 1,2,4,8 -l nastran,arch=osf nastran.par
```

is submitted, the queues suitable for this job are those which are associated to the PE interface `mpi` by the PE configuration and also satisfy the resource requirement specification specified by the *qsub -l* option.

☞ **The CODINE PE interface facility is highly configurable. In particular, the CODINE administration can configure the PE start-up and stop procedures (see the *codine_pe* manual page) to support site specific needs. The *qsub -v* and *-V* options to export environment variables may be used to pass information from the user who submits the job to the PE start-up and stop procedures. Please ask the CODINE administration if you are required to export certain environment variables.**

4.6 Submitting Jobs to Other Queueing Systems

Some sites do not wish to install CODINE on all machines for which batch access is provided, but instead use other queueing systems already available on these hosts. Typical examples are machines which do not belong to the same organization, and thus cannot be maintained by the CODINE administration, or machines

utilizing a very special queuing system, interfacing specifically designed accounting facilities and the like (very common for so called *Supercomputers*).

In these cases, CODINE offers a general interface to such queueing systems. Access to the hosting queueing system (*QS*) is provided by the concept of **transfer queues**. A transfer queue is defined by the value `TRANSFER` in the `type` field of the queue configuration (see section „Queues and Queue Properties“ on page 181).

Jobs to be forwarded to another QS can be submitted like any other CODINE job. Resource requirements are requested for the job via *qmon* or the *qsub* command just like for *normal* CODINE jobs. It is even possible that such a job is processed either within the CODINE system or passed to a QS, depending on the available and best suited resources.

Sometimes it is necessary to supply QS special switches with the job. To perform this, there are two methods available in the CODINE QS interface:

- Add the options to the script file by usage of special comments similar to the “#$” comments in CODINE (of course the QS must support such special comments).
- The special *qsub* option *-qs_args* may be used to pass such options. Everything behind the *-qs_args* option is considered as option to the QS until the *-qs_end* option is encountered. A corresponding input field for such arguments is provided in the qmon submission dialogue as well (see section „Submitting Jobs with qmon (Advanced Example)“ on page 200).

4.7 How CODINE Jobs Are Scheduled

4.7.1 Job Scheduling

Job Priorities

Concerning the order of scheduling precedence of different jobs a first-in-first-out (fifo) rule is applied by default. I.e., all **pending** (not yet scheduled) jobs are inserted in a list, with the first submitted job being the head of the list, followed by the second submitted job, and so on. The job submitted first will be attempted to be scheduled first. If at least one suitable queue is available, the job will be scheduled. CODINE will try to schedule the second job afterwards no matter whether the first has been dispatched or not.

This order of precedence among the pending jobs may be overruled by the cluster administration via a **priority value** being assigned to the jobs. The actual priority value can be displayed by using the *qstat* command (the priority value is contained in the last column of the pending jobs display entitled `P`; refer to section "Monitoring with qstat" on page 253 for details). The default priority value assigned to the jobs at submit time is 0. The priority values are positive and negative integers and the pending jobs list is sorted Correspondingly in the order of descending priority values. I.e., by assigning a relatively high priority value to a job, the job is moved to the top of the pending jobs list. Jobs with negative priority values are inserted even after jobs just submitted. If there are several jobs with the same priority value, the fifo rule is applied within that priority value category.

Equal-Share-Scheduling

The fifo rule sometimes leads to problems, especially if user's tend to submit a series of jobs almost at the same time (e.g. via shell-script issuing one submit after the other). All jobs being submitted afterwards and being designated to the same group of queues will have to wait a very long time. **Equal-share-scheduling** avoids this problem by sorting jobs of users already owning a running job to the end of the precedence list. The sorting is performed only among jobs within the same priority value category. Equal-share-scheduling is activated if the CODINE scheduler configuration entry **user_sort** (refer to the *sched_conf* manual page for details) is set to `TRUE`.

4.7.2 Queue Selection

If submitted jobs cannot be run, because requested resources like a queue of a certain group are not available at submit-time, it would be disadvantageous to immediately dispatch such jobs to a certain queue Corresponding to the load average situation. Imagine, a suitable queue is busy with a job, that is terribly slowed down by an infrequently responding I/O device. The machine, hosting this queue, might offer the lowest load average in the CODINE cluster, however, the currently executing job might also continue to run for a very long time.

Therefore, CODINE does not dispatch jobs requesting **generic** queues if they cannot be started immediately. Such jobs will be marked as spooled at the *cod_qmaster*, which will try to re-schedule them from time to time. Thus, such jobs are dispatched to the next suitable queue, that becomes available.

As opposed to this, jobs which are requested by name to a certain queue, will go directly to this queue regardless whether they can be started or they have to be spooled. Therefore, viewing CODINE queues as computer science **batch queues** is only valid for jobs requested by name. Jobs submitted with **generic** requests use the spooling mechanism of *cod_qmaster* for queueing, thus utilizing a more abstract and flexible queuing concept.

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

5 Submit Interactive Jobs

Submitting interactive jobs instead of batch jobs is useful in situations where your job requires your direct input to influence the results of the job. This is typically the case for X-windows applications, which are interactive by definition, or for tasks in which your interpretation of immediate results is required to steer the further computation.

Three methods exist in CODINE to create interactive jobs:

- *qlogin* -a telnet like session is started on a host selected by CODINE.
- *qrsh* - the equivalent of the standard Unix *rsh* facility. Either a command is executed remotely on a host selected by CODINE or a rlogin session is started on a remote host if no command was specified for execution.
- *qsh/qmon* - an *xterm* is brought up from the machine executing the job with the display set corresponding to your specification or the setting of the `DISPLAY` environment variable. If the

DISPLAY variable is not set and if no display destination was defined specifically, CODINE directs the *xterm* to the 0.0 screen of the X server on the host from which the interactive job was submitted.

☞ **To function correctly, all the facilities need proper configuration of CODINE cluster parameters. The correct *xterm* execution paths have to be defined for qsh and interactive queues have to be available for this type of jobs. Please contact your system administrator whether your cluster is prepared for interactive job execution.**

The default handling of interactive jobs differs from the handling of batch jobs in that interactive jobs are not queued if they cannot be executed by the time of submission. This is to indicate immediately, that not enough appropriate resources are available to dispatch an interactive job right after it was submitted. The user is notified in such cases that the CODINE cluster is too busy currently.

This default behavior can be changed with the *-now no* option to *qsh*, *qlogin* and *qrsh*. If this option is given, interactive jobs are queued like batch jobs. Using *-now yes*, batch jobs submitted with *qsub* also can be handled like interactive jobs and are either dispatched for execution immediately or are rejected.

☞ **Interactive jobs can only be executed in queues of the type INTERACTIVE (please refer to "Configuring Queues" on page 79 in the CODINE Installation and Administration Guide for details).**

The subsequent sections outline the usage of the *qlogin* and *qsh* facilities. The *qrsh* command is explained in a broader context in chapter "Transparent Remote Execution" on page 228.

5.1 Submit Interactive Jobs with qmon

The only type of interactive jobs which can be submitted from *qmon* are those bringing up an *xterm* on a host selected by CODINE.

By clicking to the icon on top of the button column at the right side of the `Job Submission` dialogue until the `Interactive` icon gets displayed, the job submission dialogue is prepared for submitting interactive jobs (see figure 71 on page 226 and figure

72 on page 227). The meaning and the usage of the selection options in the dialogue is the same as explained for batch jobs in section “Submitting CODINE jobs” on page 193. The basic difference is that several input fields are set insensitive because they do not apply for interactive jobs.

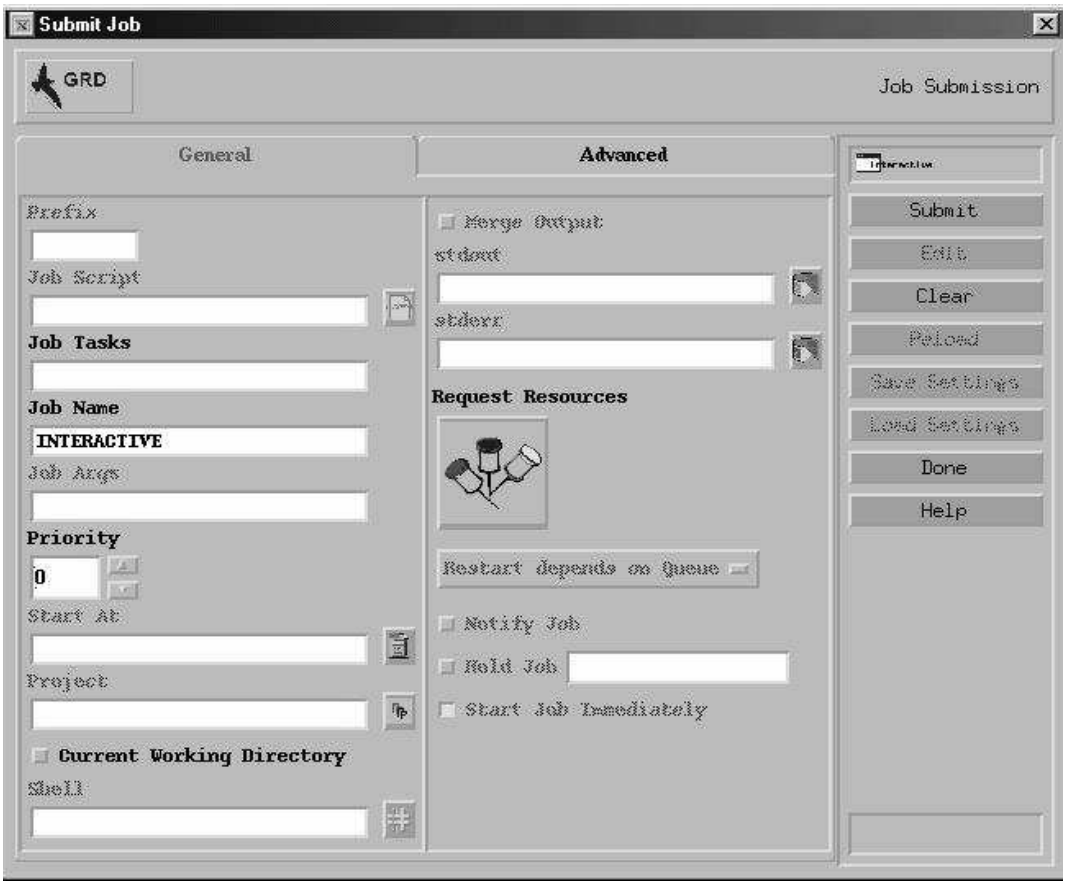


Figure 71: Interactive `Job Submission` dialogue `General`

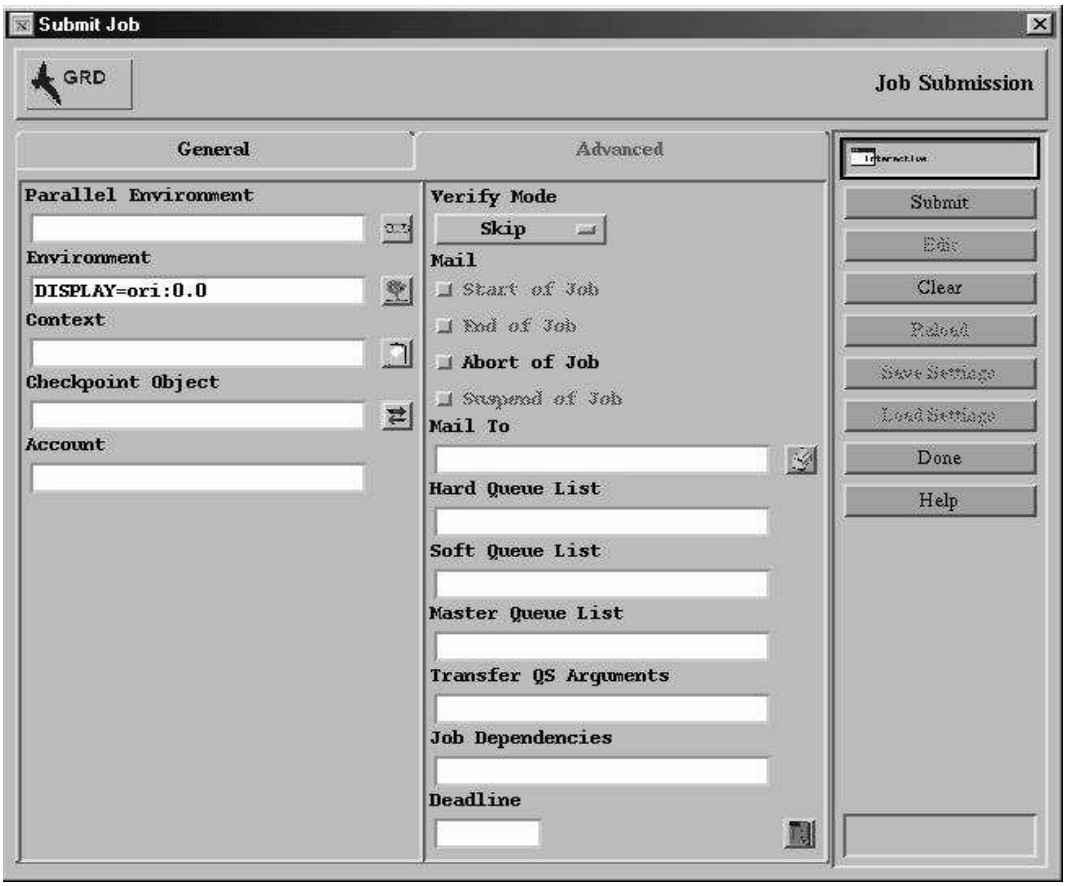
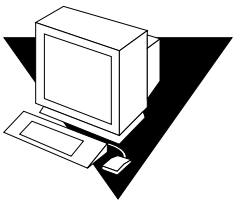


Figure 72: Interactive `Job Submission` dialogue `Advanced`

5.2 Submitting Interactive Jobs with qsh

Qsh is very similar to *qsub* and supports several of the *qsub* options as well as the additional switch *-display* to direct the display of the *xterm* to be invoked (please refer to the *qsh* manual page in the CODINE Reference Manual for details).

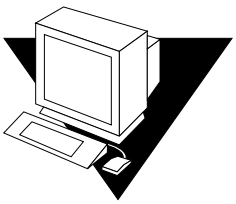
The following command will start a *xterm* on any available Sun Solaris 64bit operating system host.



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

5.3 Submitting Interactive Jobs with qlogin

The *qlogin* command can be used from any terminal or terminal emulation to initiate an interactive session under the control of CODINE. The following command will locate a low loaded host with Star-CD license available and with at least one queue providing a minimum of 6 hours hard CPU time limit.



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

☞ **Depending on the remote login facility configured to be used by CODINE you may be forced to enter your user name and/or password at a login prompt.**

6 Transparent Remote Execution

CODINE provides a set of closely related facilities supporting transparent remote execution of certain computational tasks. The core tool for this functionality is the *qrsh* command described in section “Remote Execution with qrsh” on page 229. Building on top of *qrsh*, two high level facilities - *qtcsh* and *qmake* - allow the transparent distribution of implicit computational tasks via CODINE, thereby enhancing the standard Unix facilities *make* and *csh*. *Qtcsh* is explained in section “Transparent Job Distribution with qtcsh” on page 230 and *qmake* is described in section “Parallel Makefile Processing with qmake” on page 233.

6.1 Remote Execution with qrsh

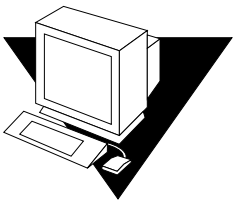
Qrsh is built around the standard *rsh* facility (see the information provided in `<codine_root>/3rd_party` for details on the involvement of *rsh*) and can be used for various purposes:

- ❏ to provide remote execution of interactive applications via CODINE comparable to the standard Unix facility *rsh* (also called *remsh* for HP-UX).
- ❏ to offer interactive login session capabilities via CODINE similar to the standard Unix facility *rlogin* (note that *qlogin* is still required as a CODINE representation of the Unix *telnet* facility).
- ❏ to allow for the submission of batch jobs which, upon execution, support terminal I/O (standard/error output and standard input) and terminal control.
- ❏ to offer a means for submitting a standalone program not embedded in a shell-script.
- ❏ to provide a batch job submission client which remains active while the job is pending or executing and which only finishes if the job has completed or has been cancelled.
- ❏ to allow for the CODINE-controlled remote execution of job tasks (such as the concurrent tasks of a parallel job) within the framework of the dispersed resources allocated by parallel jobs (see section „Tight Integration of PEs and CODINE“ on page 165 of the CODINE Installation and Administration Guide).

By virtue of all these capabilities, *qrsh* is the major enabling infrastructure for the implementation of the *qtcsh* and the *qmake* facilities as well as for the so called tight integration of CODINE with parallel environments such as MPI or PVM.

6.1.1 Qrsh Usage

The general form of the *qrsh* command is



```
% qrsh  [options] program|shell-script [arguments] \
       [> stdout_file] [>&2 stderr_file] [< stdin_file]
```

Qrsh understands almost all options of *qsub* and provides only a few additional ones. These are:

- **-now yes|no**
  controls whether the job is scheduled immediately and rejected if no appropriate resources are available, as usually desired for an interactive job – hence it is the default, or whether the job is queued like a batch job, if it cannot be started at submission time.
- **-inherit**
  qrsh does not go through the CODINE scheduling process to start a job-task, but it assumes that it is embedded inside the context of a parallel job which already has allocated suitable resources on the designated remote execution host. This form of *qrsh* commonly is used within *qmake* and within a tight parallel environment integration. The default is not to inherit external job resources.
- **-verbose**
  presents output on the scheduling process. Mainly intended for debugging purposes and therefore switched off per default.

6.2 Transparent Job Distribution with qtcsh

Qtcsh is a fully compatible replacement for the widely known and used Unix C-Shell (*csh*) derivative *tcsh* (*qmake* is built around *tcsh* - see the information provided in `<codine_root>/3rd_party` for details on the involvement of *tcsh*). It provides a command-shell with the extension of transparently distributing execution of designated applications to suitable and lightly loaded hosts via CODINE. Which applications are to be executed remotely and which requirements apply for the selection of an execution host is defined in configuration files called `.qtask`.

Transparent to the user, such applications are submitted for execution to CODINE via the *qrsh* facility. Since *qrsh* provides standard output, error output and standard input handling as well as terminal control connection to the remotely executing application, there are only three noticeable differences between executing such an application remotely as opposed to executing it on the same host as the shell:

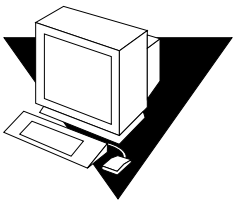
- The remote host may be much better suited (more powerful, lower loaded, required hard/software resources installed) than the local host, which may not allow execution of the application at all. This is a desired difference, of course.
- There will be a small delay incurred by the remote startup of the jobs and by their handling through CODINE.
- Administrators can restrict the usage of resources through interactive jobs (*qrsh*) and thus through *qtcsh*. If not enough suitable resources are available for an application to be started via the *qrsh* facility or if all suitable systems are overloaded, the implicit *qrsh* submission will fail and a corresponding error message will be returned ("not enough resources ... try later").

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

6.2.1 Qtcsh Usage

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

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



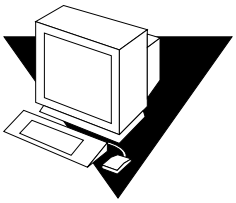
```
% [!]appl_name qrsh_options
```

The optional leading exclamation mark "!" defines the precedence between conflicting definitions in a cluster global `.qtask` file and the personal `.qtask` file of the *qtcsh* user. If the exclamation mark is missing in the cluster global file, an eventually conflicting definition in the user file will overrule. If the exclamation mark is in the cluster global file, the corresponding definition cannot be overwritten.

The rest of the line specifies the name of the application which, when typed on a command line in a *qtcsh*, will be submitted to CODINE for remote execution, and the options to the *qrsh* facility, which will be used and which define resource requirements for the application.

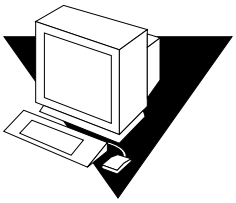
- **The application name must appear in the command line exactly like defined in the `.qtask` file. If it is prefixed with an absolute or relative directory specification it is assumed that a local binary is addressed and no remote execution is intended.**
- ***Csh* aliases, however, are expanded before a comparison with the application names is performed. The applications intended for remote execution can also appear anywhere in a *qtcsh* command line, in particular before or after standard I/O redirections.**

Hence, the following examples are valid and meaningful syntax:



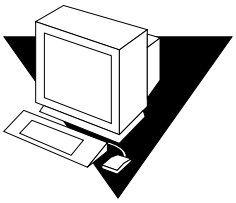
```
# .qtask file
netscape -v DISPLAY=myhost:0
grep -l h=filesurfer
```

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



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

will implicitly result in



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

Qtcsh can operate in different modes influenced by switches where each of them can be on or off:

- Local or remote execution of commands (remote is default).
- Immediate or batch remote execution (immediate is default).
- Verbose or non-verbose output (non-verbose is default).

The setting of these modes can be changed using option arguments of *qtcsh* at start time or with the shell builtin command *qrshmode* at runtime. See the *qtcsh* manual page in the CODINE Reference Manual for more information.

6.3 Parallel Makefile Processing with qmake

Qmake is a replacement for the standard Unix *make* facility. It extends *make* by its ability to distribute independent *make* steps across a cluster of suitable machines. *Qmake* is built around the popular GNU-make facility *gmake*. See the information provided in `<codine_root>/3rd_party` for details on the involvement of *gmake*.

To ensure that a complex distributed *make* process can run to completion, *qmake* first allocates the required resources in an analogous form like a parallel job. *Qmake* then manages this set of resources without further interaction with the CODINE scheduling. It distributes *make* steps as resources are or become available via the *qrsh* facility with the *-inherit* option enabled.

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

- Provided that the individual *make* steps have a certain duration and that there are enough independent *make* steps to be processed, the parallelization of the *make* process will be sped up significantly. This is a desired difference, of course.
- With each *make* step to be started up remotely there will be an implied small overhead caused by *qrsh* and the remote execution as such.
- To take advantage of the *make* step distribution of *qmake*, the user has to specify as a minimum the degree of parallelization, i.e. the number of concurrently executable *make* steps. In addition, the user can specify the resource characteristics required by the *make* steps, such as available software licenses, machine architecture, memory or CPU-time requirements.

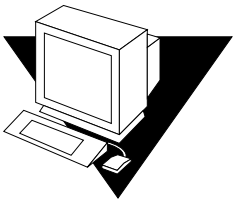
The most common use in general of *make* certainly is the compilation of complex software packages. This may not be the major application for *qmake*, however. Program files are often quite small (as a matter of good programming practice) and hence compilation of a single program file, which is a single *make* step, often only takes a few seconds. Furthermore, compilation usually implies a lot of file access (nested include files) which may not be accelerated if done for multiple *make* steps in parallel, because the file server can become the bottleneck effectively serializing all the file access. So a satisfactory speed-up of the compilation process sometimes cannot be expected.

Other potential applications of *qmake* are more appropriate. An example is the steering of the interdependencies and the workflow of complex analysis tasks through make-files. This is common in some areas, such as EDA, and each *make* step in such

environments typically is a simulation or data analysis operation with non-negligible resource and computation time requirements. A considerable speed-up can be achieved in such cases.

6.3.1 Qmake Usage

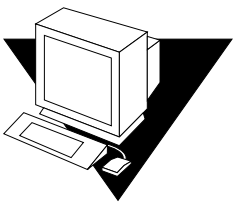
The command-line syntax of *qmake* looks very similar to the one of *qrsh*:



```
% qmake  [-pe pe_name pe_range] [further codine options] \
        -- [gnu-make-options][target]
```

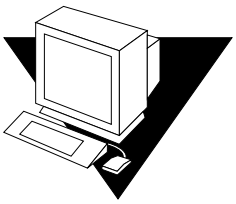
☞ **The *-inherit* option is also supported by *qmake* as described further down below.**

Specific attention has to be paid on the usage of the *-pe* option and its relation to the *gmake -j* option. Both options can be used to express the amount of parallelism to be achieved. The difference is that *gmake* provides no possibility with *-j* to specify something like a parallel environment to use. Hence, *qmake* makes the assumption, that a default environment for parallel makes is configured which is called `make`. Furthermore, *gmake´s -j* allows no specification of a range, but only for a single number. *Qmake* will interpret the number given with *-j* as a range of `1-<given_number>`. As opposed to this, *-pe* permits the detailed specification of all these parameters. Consequently, the following command-line examples are identical



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

while the following command-lines cannot be expressed via the *-j* option:



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

Apart from the syntax, *qmake* supports two modes of invocation: interactively from the command-line (without *-inherit*) or within a batch job (with *-inherit*). These two modes initiate a different sequence of actions:

- interactive – when *qmake* is invoked on the command-line, the *make* process as such is implicitly submitted to CODINE via *qrsh* taking the resource requirements specified in the *qmake* command-line into account. CODINE then selects a "master machine" for the execution of the parallel job associated with the parallel *make* job and starts the *make* procedure there. This is necessary, because the *make* process can be architecture dependent and the required architecure is specified in the *qmake* command-line. The *qmake* process on the master machine then delegates execution of individual *make* steps to the other hosts which have been allocated by CODINE for the job and which are passed to *qmake* via the parallel environment hosts file.
- batch – in this case, *qmake* appears inside a batch script with the *-inherit* option (if the *-inherit* option was not present, a new job would be spawned as described for the first case above). This results in *qmake* making use of the resources already allocated to the job into which *qmake* is embedded. It will use *qrsh -inherit* directly to start *make* steps. When calling *qmake* in batch mode, the specification of resource requirements or *-pe* and *-j* options is ignored.

☞ **Also single CPU jobs have to request a parallel environment (*qmake -pe make 1 --*). If no parallel execution is required, call *qmake* with *gmake* command-line syntax (without CODINE options and "*--*"), it will behave like *gmake*.**

Please refer to the *qmake* manual page in the CODINE Reference Manual for further detail on *qmake*.

7 Checkpointing Jobs

7.1 User Level Checkpointing

Lots of application programs, especially those, which normally consume considerable CPU time, have implemented checkpointing and restart mechanisms to increase fault tolerance. Status information and important parts of the processed data are repeatedly written to one or more files at certain stages of the algorithm. These files (called restart files) can be processed if the application is aborted and restarted at a later time and a consistent state can be reached, comparable to the situation just before the checkpoint. As the user mostly has to deal with the restart files, e.g. in order to move them to a proper location, this kind of checkpointing is called **user level** checkpointing.

For application programs which do not have an integrated (user level) checkpointing an alternative can be to use a so called *checkpointing library* which can be provided by the public domain (see the *Condor* project of the University of Wisconsin for example) or by some hardware vendors. Re-linking an application with such a library installs a checkpointing mechanism in the application without requiring source code changes.

7.2 Kernel Level Checkpointing

Some operating systems provide checkpointing support inside the operating system kernel. No preparations in the application programs and no re-linking of the application is necessary in this case. Kernel level checkpointing is usually applicable for single processes as well as for complete process hierarchies. I.e., a hierarchy of interdependent processes can be checkpointed and restarted at any time. Usually both, a user command and a C-library interface are available to initiate a checkpoint.

CODINE supports operating system checkpointing if available. Please refer to the CODINE Release Notes for information on the currently supported kernel level checkpointing facilities.

7.3 Migration of Checkpointing Jobs

Checkpointing jobs are interruptible at any time, since their restart capability ensures that only few work already done must be repeated. This ability is used to build CODINE's migration and dynamic load balancing mechanism. If requested, checkpointing CODINE jobs are aborted on demand and migrated to other machines in the CODINE pool thus averaging the load in the cluster in a dynamic fashion. Checkpointing jobs are aborted and migrated for the following reasons:

- The executing machine exceeds a load value configured to force a migration (`migr_load_thresholds` - see the *queue_conf* manual page in the CODINE Reference Manual).
- The executing queue or the job is suspended, either explicitly by *qmod* or *qmon* or automatically if a suspend threshold for the queue (see section „Configuring Load and Suspend Thresholds“ on page 85 of the CODINE Installation and Administration Guide) has been exceeded and if the checkpoint occasion specification for the job (see section „Submit/Monitor/Delete a Checkpointing Job“ on page 239) includes the suspension case.

You can identify a job which is about to migrate by the `state m` for migrating in the *qstat* output. A migrating job moves back to *cod_qmaster* and is subsequently dispatched to another suitable queue if any is available.

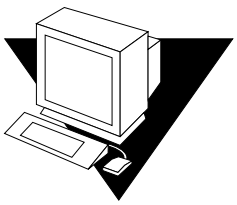
7.4 Composing a Checkpointing Job Script

Shell scripts for kernel level checkpointing show no difference from regular shell scripts.

Shell scripts for user level checkpointing jobs differ from regular CODINE batch scripts only in their ability to properly handle the case if they get restarted. The environment variable `RESTARTED` is set for checkpointing jobs which are restarted. It can be used to skip over sections of the job script which should be executed during the initial invocation only.

Thus, a transparently checkpointing job script may look similar to the one given below:

7.4.1 Example Script File



```
#!/bin/sh
# Force /bin/sh in CODINE
#$ -S /bin/sh

# Test if restarted/migrated
if [ $RESTARTED = 0 ]; then
        # 0 = not restarted
        # Parts to be executed only during the first
        # start go in here
        set_up_grid
fi

# Start the checkpointing executable
fem

#End of scriptfile
```

It is important to note that the job script is restarted from the beginning if a user level checkpointing job is migrated. The user is responsible for directing the program flow of the shell-script to the location where the job was interrupted and thus skipping those lines in the script which are critical to be executed more than once.

☞ **Kernel level checkpointing jobs are interruptible at any point of time and also the embracing shell script is restarted exactly from the point where the last checkpoint occurred. Therefore, the `RESTARTED` environment variable are of no relevance for kernel level checkpointing jobs.**

7.5 Submit/Monitor/Delete a Checkpointing Job

Submitting a checkpointing job works the same way as for regular batch scripts except for the *qsub -ckpt* and *-c* switches, which request a checkpointing mechanism and define the occasions at which checkpoints have to be generated for the job. The *-ckpt* option takes one argument which is the name of the checkpointing environment (see section „Checkpointing Support“ on page 152 in the CODINE Installation and Administration Guide) to be used.

The *-c* option is not mandatory and also takes one argument. It can be used to overwrite the definitions of the `when` parameter in the checkpointing environment configuration (see the *checkpoint* manual page in the CODINE Reference Manual for details). The argument to the -c option can be one of the following one letter selection (or any combination thereof) or a time value alternatively:

- **n**
  no checkpoint is performed. This has highest precedence
- **s**
  A checkpoint is only generated if the *cod_execd* on the jobs host is shut down.
- **m**
  Generate checkpoint at minimum CPU interval defined in the corresponding queue configuration (see the `min_cpu_interval` parameter in the *queue_conf* manual page).
- **x**
  A checkpoint is generated if the job gets suspended.
- ***interval***
  Generate checkpoint in the given interval but not more frequently than defined by `min_cpu_interval` (see above). The time value has to be specified as hh:mm:ss (two digit hours, minutes and seconds separated by colon signs).

The monitoring of checkpointing jobs just differs from regular jobs by the fact, that these jobs may migrate from time to time (signified by `state m` for migrating in the output of *qstat*, see above) and, therefore, are not bound to a single queue. However, the unique job identification number stays the same as well as the job name.

Deleting checkpointing jobs works just the same way as described in section “Controlling CODINE Jobs from the Command-line” on page 256.

7.6 Submit a Checkpointing Job with qmon

Submission of checkpointing jobs via qmon is identical to the submission of regular batch jobs with the addition of specifying an appropriate checkpointing environment. As explained in “Submitting Jobs with qmon (Advanced Example)” on page 200 the Job Submission dialogue provides an input window for the checkpointing environment associated with a job. Aside to the input

window there is an icon button, which opens the selection dialogue displayed in figure 73 on page 241. You can select a suitable checkpoint environment from the list of available ones with it. Please ask your system administrator for information on the properties of the checkpointing environments installed at your site or refer to section “Checkpointing Support” on page 152.

Figure 73: Checkpoint Object Selection

7.7 File System Requirements

When a checkpointing library based user level or kernel level checkpoint is written, a complete image of the virtual memory the process or job to be checkpointed covers needs to be dumped. Sufficient disk space must be available for this purpose. If the checkpointing environment configuration parameter `ckpt_dir` is set the checkpoint information is dumped to a job private location under `ckpt_dir`. If `ckpt_dir` is set to NONE, the directory in which the checkpointing job was started is used. Please refer to the manual page *checkpoint* in the CODINE Reference Manual for detailed information about the checkpointing environment configuration.

☞ **You should start a checkpointing job with the *qsub -cwd* script if `ckpt_dir` is set to NONE.**

An additional requirement concerning the way how the file systems are organized is caused by the fact, that the checkpointing files and the restart files must be visible on all machines in order to successfully migrate and restart jobs. Thus NFS or a similar file system is required. Ask your cluster administration, if this requirement is met for your site.

If your site does not run NFS or if it is not desirable to use it for some reason, you should be able to transfer the restart files explicitly at the beginning of your shell script (e.g. via *rcp* or *ftp*) in the case of user level checkpointing jobs.

8 Monitoring and Controlling CODINE Jobs

In principle, there are three ways to monitor submitted jobs: with the CODINE graphical user's interface *qmon*, from the command-line with the *qstat* command or by electronic mail.

8.1 Monitoring and Controlling Jobs with qmon

The CODINE graphical user's interface *qmon* provides a dialogue specifically designed for controlling jobs. The `Job Control` dialogue is opened by pushing the `Job Control` icon button in the *qmon* main menu.

The general purpose of this dialogue is to provide the means to monitor all running, pending and a configurable number of finished jobs known to the system or parts thereof. The dialogue can also be used to manipulate jobs, i.e. to change their priority, to suspend, resume and to cancel them. Three list environments are displayed, one for the running jobs, another for the pending jobs waiting to be dispatched to an appropriate resource and the third for recently finished jobs. You can select between the three list environments via clicking to the corresponding tab labels at the top of the screen.

In its default form (see figure 74 on page 246) it displays the columns `JobId`, `Priority`, `JobName` and `Queue` for each running and pending job. The set of information displayed can be configured with a customization dialogue (see figure 74 on page 246), which is opened upon pushing the `Customize` button in the `Job Control` dialogue. With the customization dialogue it is possible to select further entries of the CODINE job object to be displayed and to filter the jobs of interest. The example on page 246

selects the additional fields `MailTo` and `Submit Time`. The `Job Control` dialogue displayed in figure 74 on page 246 depicts the enhanced look after the customization has been applied in case of the `Finished Jobs` list. The example of the filtering facility in figure 77 on page 249 selects only those jobs owned by `ferstl` which run or are suitable for architecture `solaris64`. The resulting `Job Control` dialogue showing `Pending Jobs` is displayed in figure 78 on page 250.

☞ **The `Save` button the customize dialogue displayed on page page 246, for example, stores the customizations into the file `.qmon_preferences` in the user's home directory and thus redefines the default appearance of the job control dialogue.**

The `Job Control` dialogue in figure 78 on page 250 is also an example for how array jobs are displayed in qmon.

Jobs can be selected (for later operation) with the following mouse/key combinations:

- Clicking to a job with the left mouse button while the Control key is pressed starts a selection of multiple jobs.
- Clicking to another job with the left mouse button while the Shift key is pressed selects all jobs in between and including the job at the selection start and the current job.
- Clicking to a job with the left mouse button while the Control and the Shift key are pressed toggles the selection state of a single job.

The selected jobs can be suspended, resumed (unsuspended), deleted, held back (and released), re-prioritized and modified (`Qalter`) through the Corresponding buttons at the right side of the screen.

The actions suspend, unsuspend, delete, hold, modify priority and modify job may only be applied to a job by the job owner or by CODINE managers and operators (see "Managers, Operators and Owners" on page 190). Only running jobs can be suspended/resumed and only pending jobs can be held back and modified (in priority as well as in other attributes).

Suspending a job means the equivalent to sending the signal SIGSTOP to the process group of the job with the UNIX *kill* command. I.e., the job is halted and does no longer consume CPU time. Unsuspending the job sends the signal SIGCONT thereby resuming the job (see the *kill* manual page of your system for more information on signalling processes).

☞ **Suspension, unsuspension and deletion can be forced, i.e. registered with *cod_qmaster* without notification of the *cod_execd* controlling the job(s), in case the corresponding *cod_execd* is unreachable, e.g. due to network problems. Use the `Force` flag for this purpose.**

If using the `Hold` button on a selected pending job, the `Set Hold` sub-dialogue is opened (see figure 74 on page 246). It allows to set and to reset user, system and operator holds. User holds can be set/reset by the job owner as well as CODINE operators and managers. Operator holds can be set/reset by managers and operator and manager holds can be set/reset by managers only. As long as any hold is assigned to a job it is not eligible for execution. An alternate way to set/reset holds are the *qalter*, *qhold* and *qrls* commands (see the corresponding manual pages in CODINE Reference Manual).

If the `Priority` button is pressed another sub-dialogue is opened (figure 74 on page 246), which allows to enter the new priority of the selected pending jobs. The priority determines the order of the jobs in the pending jobs list and the order in which the pending jobs are displayed by the `Job Control` dialogue. Users can only set the priority in the range between 0 and -1024. CODINE operators and managers can also increase the priority level up to the maximum of 1023 (see section „Job Priorities“ on page 137 in the CODINE Installation and Administration Guide for details about job priorities).

The `Qalter` button, when pressed for a pending job, opens the `Job Submission` screen described in “Submitting CODINE jobs” on page 193 with all the entries of the dialogue set corresponding to the attributes of the job as defined during submission. Those entries, which cannot be changed are set insensitive. The others may be edited and the changes are registered with CODINE by pushing the `Qalter` button (a replacement for the `Submit` button) in the `Job Submission` dialogue.

The `Verify` flag in the `Job Submission` screen has a special meaning when used in the "qalter" mode. You can check pending jobs for their consistency and investigate why they have not been scheduled yet. You just have to select the desired consistency checking mode for the `Verify` flag and push the `Qalter` button. The system will display warnings on inconsistencies depending on the selected checking mode. Please refer to "Submitting Jobs with qmon (Advanced Example)" on page 200 and the *-w* option in the *qalter* manual page for further information.

Another method for checking why jobs are still pending is to select a job and click on the "`Why ?`" button of the `Job Control` dialogue. This will open the `Object Browser` dialogue and display a list of reasons which prevented the CODINE scheduler from dispatching the job in its most recent pass. An example browser screen displaying such a message is shown in figure 81 on page 252.

- **The "`Why ?`" button only delivers meaningful output if the scheduler configuration parameter `schedd_job_info` is set to true (see *sched_conf* in the CODINE Reference Manual).**

- **The displayed scheduler information relates to the last scheduling interval. It may not be accurate anymore by the time you investigate for reasons why your job has not been scheduled.**

The `Clear Error` button can be used to remove an error state from a selected pending job, which had been started in an earlier attempt, but failed due to a job dependent problem (e.g., insufficient permissions to write to the specified job output file).

- **Error states are displayed using a red font in the pending jobs list and should only be removed after correcting the error condition, e.g., via *qalter*.**

- **Such error conditions are automatically reported via electronic mail, if the job requests to send e-mail in cases it is aborted (e.g. via the *qsub -m a* option).**

To keep the information being displayed up-to-date, *qmon* uses a polling scheme to retrieve the status of the jobs from *cod_qmaster*. An update can be forced by pressing the `Refresh` button.

Finally, the `Submit` button provides a link to the *qmon* `Job Submission` dialogue (see figure 64 on page 200 for example).

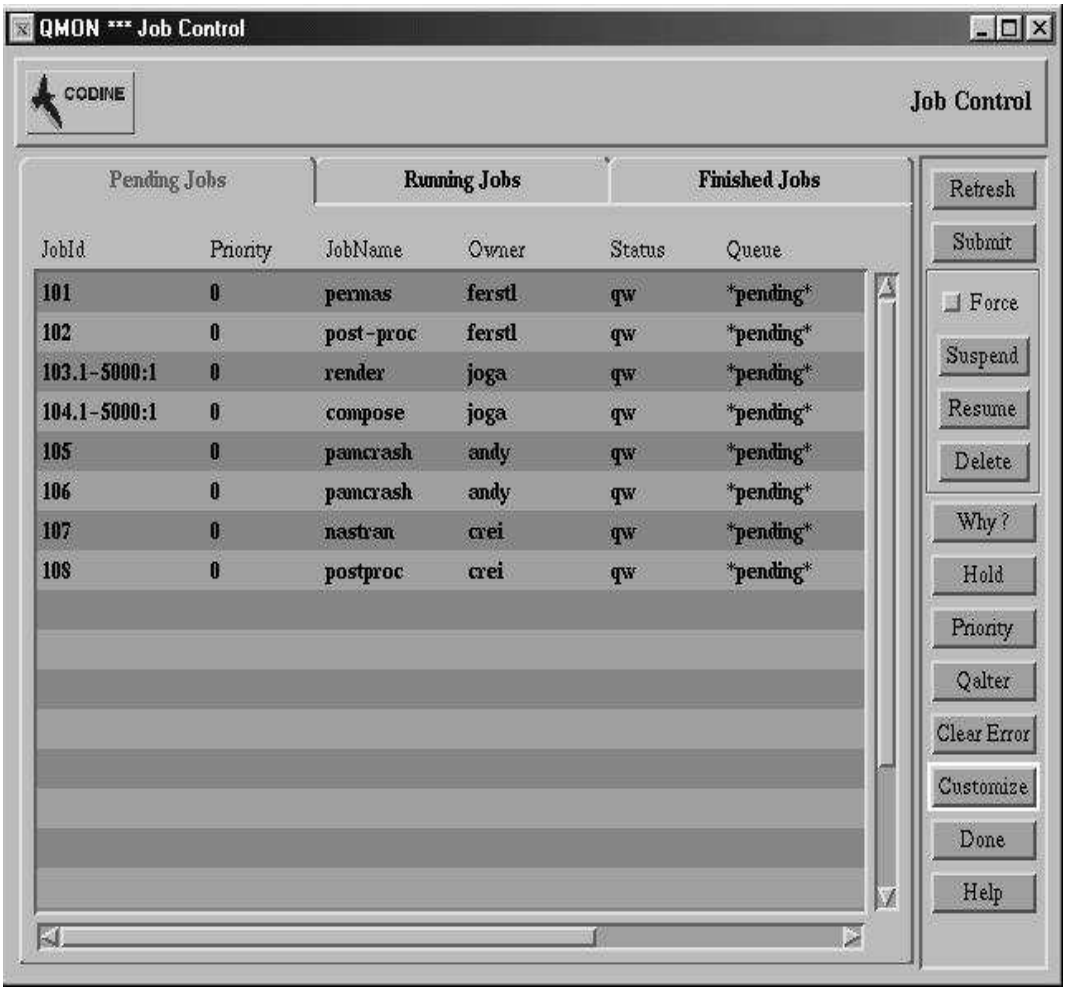


Figure 74: `Job Control` dialogue - standard form

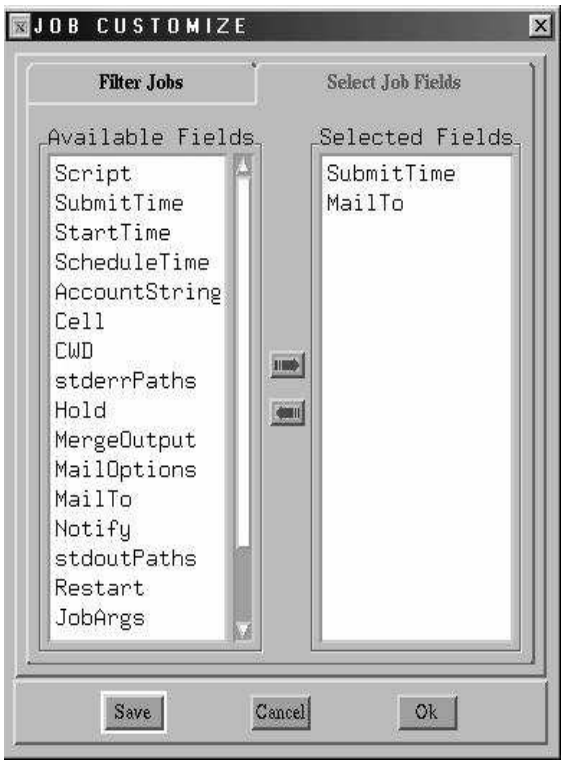


Figure 75: `Job Control` customization

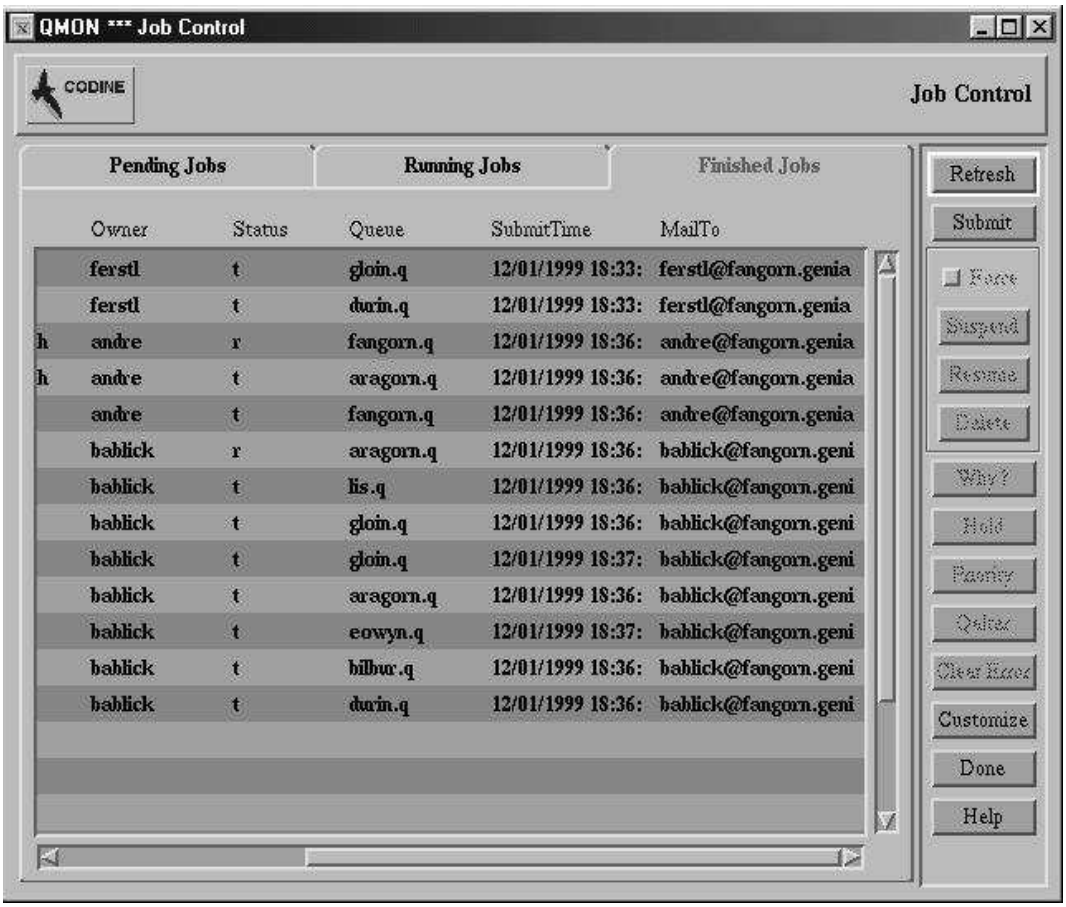


Figure 76: `Job Control` dialogue `Finished Jobs` - enhanced

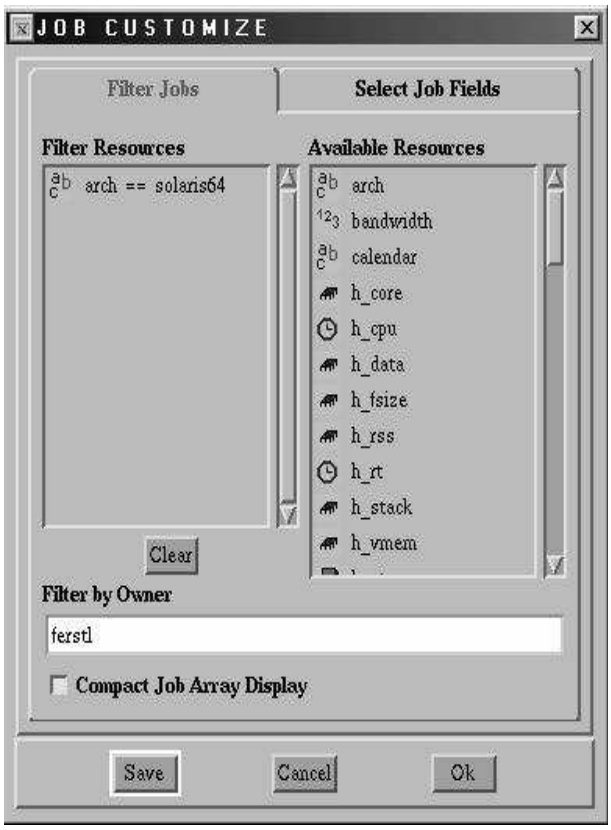


Figure 77: `Job Control` filtering

Figure 78: `Job Control` dialogue - after filtering

Figure 79: `Job Control` holds

Figure 80: `Job Control` priority definition

Figure 81: `Browser` displaying scheduling information

8.2 Additional Information with the *qmon* Object Browser

The *qmon* `Object Browser` can be used to quickly retrieve additional information on CODINE jobs without a need to customize the `Job Control` dialogue as explained in section “Monitoring and Controlling Jobs with qmon” on page 242.

The `Object Browser` is opened upon pushing the `Browser` icon button in the *qmon* main menu. The browser displays information about CODINE jobs if the *Job* button in the browser is selected and if the mouse pointer is moved over a job‘s line in the `Job Control` dialogue (see figure 74 on page 246 for example).

The browser screen in figure 82 on page 253 gives an example of the information displayed in such a situation.

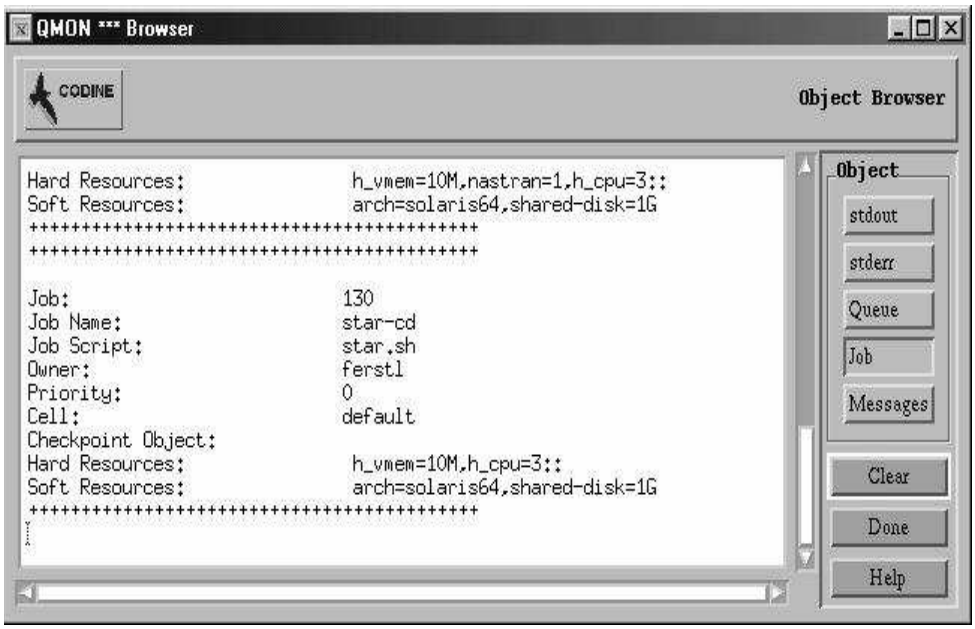
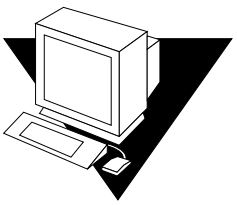


Figure 82: `Object Browser` - job

8.3 Monitoring with *qstat*

Submitted jobs can also be monitored with the CODINE *qstat* command. There are two basic forms of the *qstat* command available:



```
% qstat
% qstat -f
```

The first form provides an overview on the submitted jobs only (see table 6 on page 255). The second form includes information on the currently configured queues in addition (see table 7 on page 255).

In the first form, a header line indicates the meaning of the columns. The purpose of most of the columns should be self-explanatory. The `state` column, however, contains single character codes with the following meaning: `r` for running, `s` for suspended, `q` for queued and `w` for waiting (see the *qstat* manual page in the CODINE Reference Manual for a detailed explanation of the *qstat* output format).

The second form is divided into two sections, the first displaying the status of all available queues, the second (entitled with the `- PENDING JOBS - ...` separator) shows the status of the *cod_qmaster* job spool area. The first line of the queue section defines the meaning of the columns with respect to the enlisted queues. The queues are separated by horizontal rules. If jobs run in a queue they are printed below the associated queue in the same format as in the *qstat* command in its first form. The pending jobs in the second output section are also printed as in qstat's first form.

The following columns of the queue description require some explanation:

- **qtype**
  The queue type - one of B(atch), I(nteractive), P(arallel) and C(heckpointing) or combinations thereof or alternatively T(ransfer).
- **used/free**
  The count of used/free job slots in the queue.
- **states**
  The state of the queue - one of u(nknown), a(laram), s(uspended), d(isabled), E(rror) or combinations thereof.

Again, the *qstat* manual page contains a more detailed description of the *qstat* output format.

Various additional options to the *qstat* command enhance the functionality in both versions. The *-r* option can be used to display the resource requirements of submitted jobs. Furthermore the output may be restricted to a certain user, to a specific queue and the *-l* option may be used to specify resource requirements as described in section "Resource Requirement Definition" on page 213 for the *qsub* command. If resource requirements are used, only those

queues (and the jobs running in these queues) are displayed which match the resource requirement specification in the *qstat* command-line.

Table 6: *qstat* example output

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

Table 7: *qstat -f* example output

queuename			qtype	used/free	load_avg	arch	states	
dq			BIP	0/1	99.99	sun4	au	
durin.q			BIP	2/2	0.36	sun4		
231	0	hydra		craig	r	07/13/96	20:27:15	MASTER
232	0	compile		penny	r	07/13/96	20:30:40	MASTER
dwain.q			BIP	3/3	0.36	sun4		
230	0	blackhole		don	r	07/13/96	20:26:10	MASTER
233	0	mac		elaine	r	07/13/96	20:30:40	MASTER
234	0	golf		shannon	r	07/13/96	20:31:44	MASTER
fq			BIP	0/3	0.36	sun4		
##								
- PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS -								
##								
236	5	word		elaine	qw	07/13/96	20:32:07	
235	0	andrun		penny	qw	07/13/96	20:31:43	

8.4 Monitoring by Electronic Mail

The *qsub -m* switch requests electronic mail to be sent to the user submitting a job or to the email address(es) specified by the *-M* flag if certain events occur (see the *qsub* manual page for a description of the flags). An argument to the *-m* option specifies the events. The following selections are available:

- **b**
  Mail is sent at the beginning of the job.
- **e**
  Mail is sent at the end of the job.
- **a**
  Mail is sent when the job is aborted (e.g. by a *qdel* command).
- **s**
  Mail is sent when the job is suspended.
- **n**
  No mail is sent (the default).

Multiple of these options may be selected with a single *-m* option in a comma separated list.

The same mail events can be configured by help of the qmon Job Submission dialogue, see section „Submitting Jobs with qmon (Advanced Example)“ on page 200.

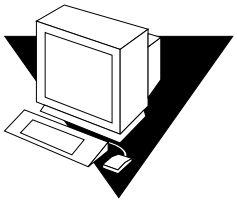
8.5 Controlling CODINE Jobs from the Command-line

The section “Monitoring and Controlling Jobs with qmon” on page 242 explains how CODINE jobs can be deleted, suspended and resumed with the CODINE graphical user‘s interface *qmon*.

From the command-line, the *qdel* command can be used to cancel CODINE jobs, regardless whether they are running or spooled. The *qmod* command provides the means to suspend and unsuspend (resume) jobs already running.

For both commands, you will need to know the job identification number, which is displayed in response to a successful *qsub* command. If you forget the number it can be retrieved via *qstat* (see section “Monitoring with qstat” on page 253).

Included below are several examples for both commands:



```
% qdel job_id
% qdel -f job_id1,job_id2
% qmod -s job_id
% qmod -us -f job_id1,job_id2
%qmon -s job_id.task_id_range
```

In order to delete, suspend or unsuspend a job you must be either the owner of the job, a CODINE manager or operator (see “Managers, Operators and Owners” on page 190).

For both commands the *-f* force option can be used to register a status change for the job(s) at *cod_qmaster* without contacting *cod_execd* in case *cod_execd* is unreachable, e.g. due to network problems. The *-f* option is intended for usage by the administrator. In case of *qdel*, however, users can be enabled to force deletion of their own jobs if the flag `ENABLE_FORCED_QDEL` in the cluster configuration `qmaster_params` entry is set (see the *codine_conf* manual page in the CODINE Reference Manual for more information).

9 Job Dependencies

The most convenient way to build a complex task often is to split the task into sub-tasks. In these cases sub-tasks depend on the successful completion of other sub-tasks before they can get started. An example is that a predecessor task produces an output file which has to be read and processed by a successor task.

CODINE supports interdependent tasks with its job dependency facility. Jobs can be configured to depend on the successful completion of one or multiple other jobs. The facility is enforced by the *qsub -hold_jid* option. A list of jobs can be specified upon which the submitted job depends. The list of jobs can also contain subsets of array jobs. The submitted job will not be eligible for execution unless all jobs in the dependency list have completed successfully.

10 Controlling Queues

As already stated in section “Queues and Queue Properties” on page 181, the owners of queues have permission to suspend/unsuspend or disable/enable queues. This is desirable, if these users need certain machines from time to time for important work and if they are affected strongly by CODINE jobs running in the background.

There are two ways to suspend or enable queues. The first, using the *qmon* `Queue Control` dialogue and the second utilizing the *qmod* command.

10.1 Controlling Queues with qmon

Clicking on the `Queue Control` icon button in the *qmon* main menu brings up the `Queue Control` dialogue. An example screen is displayed in “Queue Control dialogue” on page 259.

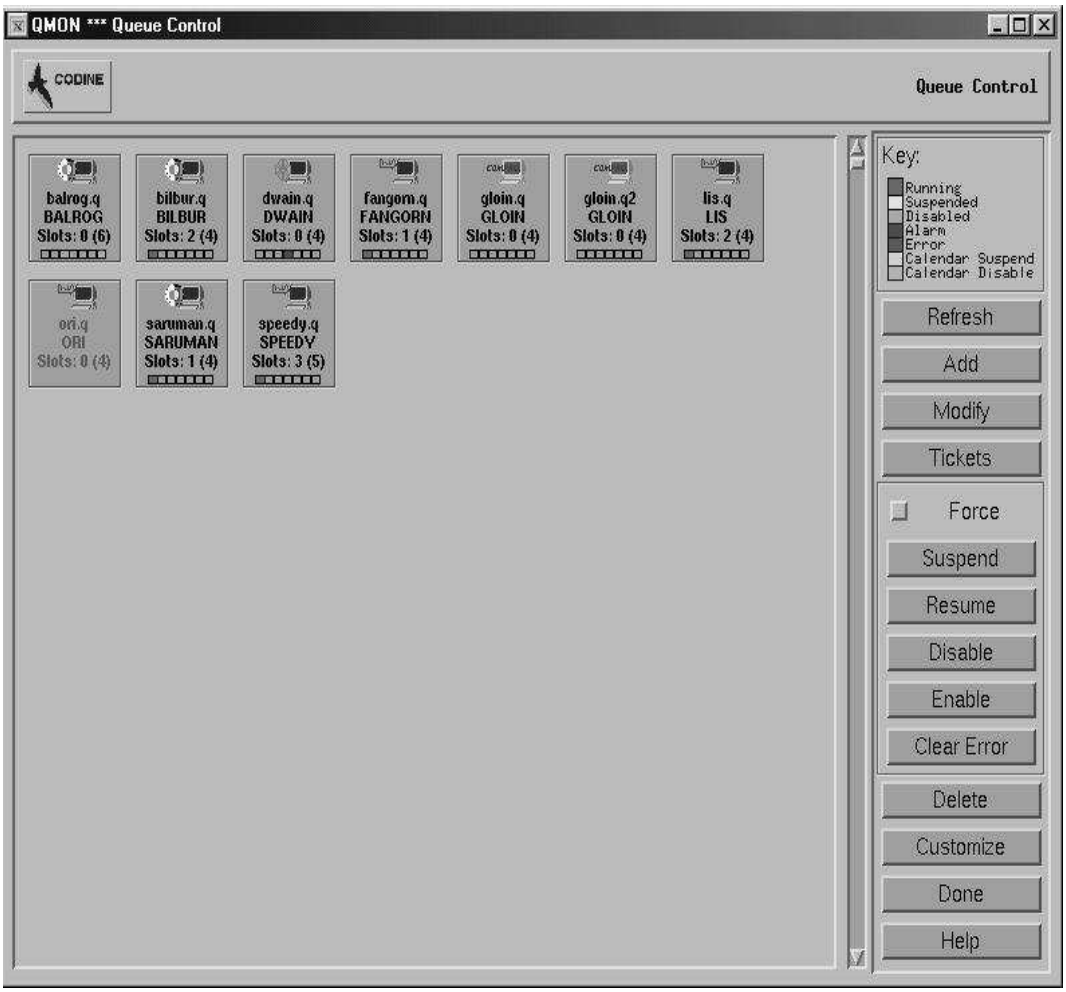


Figure 83: `Queue Control` dialogue

The purpose of the `Queue Control` dialogue is to provide a quick overview on the resources being available and on the activity in the cluster. It also provides the means to suspend/unsuspend and to disable/enable queues as well as to configure queues. Each icon being displayed represents a queue. If the main display area is empty, no queues are configured. Each queue icon is labelled with the queue name, the name of the host on which the queue resides and the number of job slots being occupied. If a *cod_execd* is

running on the queue host and has already registered with *cod_qmaster* a picture on the queue icon indicates the queue host's operating system architecture and a color bar at the bottom of the icon informs about the status of the queue. A legend on the right side of the `Queue Control` dialogue displays the meaning of the colors.

For those queues, the user can retrieve the current attribute, load and resource consumption information for the queue and implicitly of the machine which hosts a queue by clicking to the queue icon with the left mouse button while the `Shift` key on the keyboard is pressed. This will pop-up an information screen similar to the one displayed in figure 84 on page 262 (see there for a detailed description).

Queues are selected by clicking with the left mouse on the button or into a rectangular area surrounding the queue icon buttons. The `Delete`, `Suspend`/`Unsuspend` or `Disable`/`Enable` buttons can be used to execute the corresponding operation on the selected queues. The suspend/unsuspend and disable/enable operation require notification of the corresponding *cod_execd*. If this is not possible (e.g. because the host is down) a *cod_qmaster* internal status change can be forced if the `Force` toggle button is switched on.

If a queue is suspended, the queue is closed for further jobs and the jobs already executing in the queue are suspended as explained in section "Monitoring and Controlling Jobs with qmon" on page 242. The queue and its jobs are resumed as soon as the queue is unsuspended.

☞ **If a job in a suspended queue has been suspended explicitly in addition, it will not be resumed if the queue is unsuspended. It needs to be unsuspended explicitly again.**

Queues which are disabled are closed, however, the jobs executing in those queues are allowed to continue. To disable a queue is commonly used to „drain" a queue. After the queue is enabled, it is eligible for job execution again. No action on still executing jobs is performed.

The suspend/unsuspend and disable/enable operations require queue owner or CODINE manager or operator permission (see section „Managers, Operators and Owners" on page 190).

The information displayed in the `Queue Control` dialogue is update periodically. An update can be forced by pressing the `Refresh` button. The `Done` button closes the dialogue.

The `Customize` button allows you to select the queues to be displayed via a filter operation. The sample screen in figure 85 on page 263 shows the selection of only those queues which run on hosts belonging to architecture `osf4` (i.e Compaq Unix version 4). The `Save` button in the customize dialogue allows you to store your settings in the file `.qmon_preferences` in your home directory for standard reactivation on later invocations of *qmon*.

For the purpose of configuring queues a sub-dialogue is opened when pressing the `Add` or `Modify` button on the right side of the `Queue Control` screen (see section „Configuring Queues with qmon“ on page 79 in the CODINE Installation and Administration Guide for details).

In the following, a detailed description of the queue attribute screen displayed below is given:

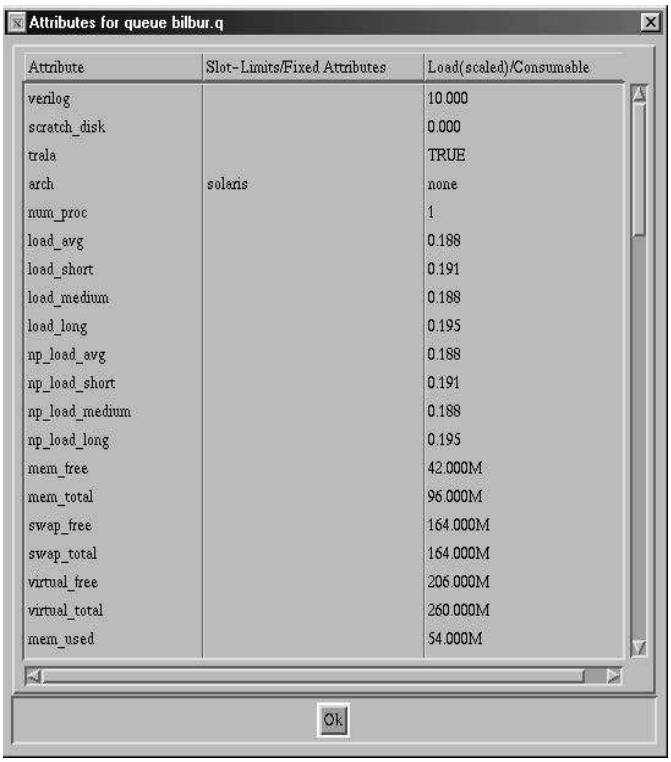


Figure 84: Queue attribute display

All attributes attached to the queue (including those being inherited from the host or cluster) are listed in the `Attribute` column. The `Slot-Limits/Fixed Attributes` column shows values for those attributes being defined as per queue slot limits or as fixed complex attributes. The `Load(scaled)/Consumable` column informs about the reported (and if configured scaled) load parameters (see section „Load Parameters“ on page 121 in the CODINE Installation and Administration Guide) and about available resource capacities based on the CODINE consumable resources facility (see section „Consumable Resources“ on page 105).

- **Load reports and consumable capacities may overwrite each other, if a load attribute is configured as a consumable resource. The minimum value of both, which is used in the job dispatching algorithm, is displayed.**

- **The displayed load and consumable values currently do not take into account load adjustment corrections as described in section "Execution Hosts" on page 65 of the CODINE Installation and Administration Guide.**

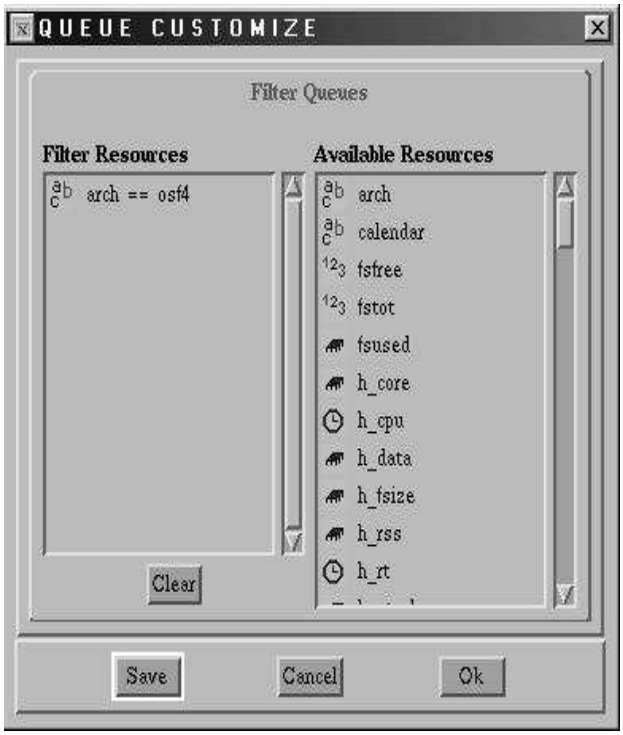
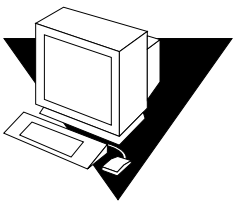


Figure 85: `Queue Control` customization

10.2 Controlling Queues with qmod

Section "Controlling CODINE Jobs from the Command-line" on page 256 explained how the CODINE command *qmod* can be used to suspend/unsuspend CODINE jobs. However, the *qmod* command additionally provides the user with the means to suspend/unsuspend or disable/enable queues.

The following commands are examples how *qmod* is to be used for this purpose:



```
% qmod -s q_name
% qmod -us -f q_name1,q_name2
% qmod -d q_name
% qmod -e q_name1,q_name2,q_name3
```

The first two commands suspend or unsuspend queues, while the third and fourth command disable and enable queues. The second command uses the *qmod -f* option in addition to force registration of the status change in *cod_qmaster* in case the corresponding *cod_execd* is not reachable, e.g. due to network problems.

- **Suspending/unsuspending as well as disabling/enabling queue requires queue owner, CODINE manager or operator permission (see section „Managers, Operators and Owners“ on page 190).**
- **You can use *qmod* commands with *crontab* or *at* jobs.**

11 Customizing qmon

The look and feel of *qmon* is largely defined by a specifically designed resource file. Reasonable defaults are compiled-in and a sample resource file is available under `<codine_root>/qmon/Qmon`.

The cluster administration may install site specific defaults in standard locations such as `/usr/lib/X11/app-defaults/Qmon`, by including *qmon* specific resource definitions into the standard `.Xdefaults` or `.Xresources` files or by putting a site specific `Qmon` file to a location referenced by standard search paths such as `XAPPLRESDIR`. Please ask your administrator if any of the above is relevant in your case,

In addition, the user can configure personal preferences by either copying and modifying the `Qmon` file into the home directory (or to another location pointed to by the private `XAPPLRESDIR` search path) or by including the necessary resource definitions into the user‘s private `.Xdefaults` or `.Xresources` files. A private `Qmon` resource file may also by installed via the *xrdb* command during operation or at start-up of the X11 environment, e.g. in a `.xinitrc` resource file.

Please refer to the comment lines in the sample `Qmon` file for detailed information on the possible customizations.

Another means of customizing *qmon* has been explained for the job and queue control customization dialogues shown in figure 74 on page 246 and in figure 85 on page 263. In both dialogues, the `Save` button can be used to store the filtering and display definitions configured with the customization dialogues to the file `.qmon_preferences` in the user’s home directory. Upon being restarted, *qmon* will read this file and reactivate the previously defined behavior.