OpenJCS - Unix Batch Job System Proposal

Version 0.1.8

By

Carl W. Kemp III

As of 26-Aug-08

Contents

Overview

The purpose of this document is to define a state-of-the-art batch scheduling system for use on Unix systems called OpenJCS (Job Control System). Through the intelligent use of load balancing, dependency control, scheduling, runbooks, queueing and resource allocation subsystems, the objective is to create a thoroughly integrated system of unmatched scheduling flexibility that still gives good visibility into system usage, while providing scalability and ease of use wherever possible.

Scheduling Features

OpenJCS has the features you have come to expect in enterprise class schedulers, such as:

- Full dependency control between jobs and across job groups.
- Calendar scheduling, including repetitive interval, and custom calendars.
- Job queueing.
- Resource control and allocation.
- Ad hoc job scheduling and control.
- Job monitoring and completion verification.
- Automatic schedule recovery and restart in event of scheduling system shutdown.
- Custom notifications for failed jobs.

But OpenJCS also provides some features you might not expect in a commercial scheduler, much less an Open Source scheduler:

- The ability to aggregate hosts in ways that make sense to you.
- The ability to see why a job is on hold, not just the fact that it is on hold.
- The ability to share special OpenJCS variables between jobs, or even between systems.
- The ability to look into a job as it executes.
- An API for hooking your programs into OpenJCS.
- The ability to integrate new dependencies and commands into OpenJCS.
- Virtual Host definition so you can cluster hosts that share a given workload.
- The ability to prompt for runtime parameters for jobs.
- Security that lets you specify who can run what, when it can run, and under whose ID it can run.
- Job Runbooks that allow greater control over how jobs run, and how to verify if they ran correctly.
- Job Runtime Analysis.
- Transparent dependency control across and between disparate systems, and the ability to synchronize jobs across and between these systems.
- Job Interrupts, Resumption, and Restart.
- Specification of special pre-processors for jobs.
- Transparent transfer of job files to target hosts.

Conceptual Basics

Batch Jobs and Events

OpenJCS is at its heart an event scheduler. So, what is an event in OpenJCS? An event can be any of the following:

- A single command.
- A file of commands to be executed.
- A batch job.
- Another dependency or schedule.
- A file containing any combination of the above.

A batch job is a set of commands that are executed as a unit in OpenJCS' own batch control system. The commands can be read from a file, or submitted a line at a time from the user's standard input. The commands usually execute under the same user name as the user who submitted the job, but if the user is given sufficient permission, the job can be executed as another user as well. Once the job is successfully submitted, the user will be given a Job Number that can be used to track the status of the job when it begins to execute. From that point forward, the job is completely separate from the user's process tree, and executes under the control and supervision of OpenJCS.

Dependencies

In OpenJCS, a dependency is anything that puts a condition on when an event can or cannot execute. For example, you might not want a job on server A to execute until another job on machine B has successfully finished. The batch control system contains a large variety of dependencies that the user may use in scheduling. Some of the dependencies available are: time and date scheduling (including drop dead date and repetitive interval scheduling), job dependency scheduling, scheduling by arbitrary logical expression, manual operation confirmation, file dependencies, resource dependencies, and job concurrency dependencies. These dependencies can be mixed and matched as desired.

Time and Date Scheduling – The ability to schedule events according to time and date dependencies is the most basic requirement for any scheduler. At the very least, a scheduler should be able to schedule by time of day, day of week, day of month, month of year, week of month, and any combination of these. OpenJCS can schedule according to all of these, plus other date combinations, such as: week of year, week of quarter, days before end of month, days, weeks or months until or since another date.

But OpenJCS can do more than just these. It also lets you define your own date patterns for things like business day calendars and fiscal calendars. This way, you can tailor your scheduling according to your calendars. OpenJCS even lets you define your own calendars and temporal units so that you can make your calendars as flexible as you want.

Logical Dependencies – OpenJCS allows you to schedule an event for when a condition becomes true, for whenever a condition becomes true, repeat the event while the condition is true, or repeat the event until a condition becomes true. So what makes up a logical condition? A logical condition results from the logical comparison between two items chosen from the following:

- A constant or literal.
- An environment variable.
- A special JCS variable (see the JCS Variable section or the variable command for further information.
- A piece of information about a current job as returned by jobinfo. (See library routine jobinfo)
- A piece of information about a file as returned by finfo. (See library routine finfo)
- A piece of information about a pending reply as returned by replyinfo. (See library routine replyinfo)
- A piece of information about a jobqueue as returned by jobqinfo. (See library routine jobqinfo)
- A piece of information about a resource as returned by resinfo. (See library routine resinfo)
- A piece of information from an SNMP MIB.
- The output of a programatically executable command.

Logical conditions can also be compounded by the use of AND and or, and can be negated with NOT. Perhaps a few examples will make this clearer.

Example 1: job –launch –file myfile –when finfo datafile,eof > 0 AND jobinfo joba,state = “FINISHED”

This will launch the job file myfile when the file datafile exists and is nonempty, and job joba has finished.

Example 2: schedule –whenever \`ps –eafl|grep sshd|wc –l\` < 1 –cmd /etc/init.d/sshd start

This will restart sshd whenever it is not found on the system.

Manual Confirmation – Sometimes, there are events that just can't be accounted for automatically. For example, you may have to wait for a tape to arrive before a job can be launched. Or you may have to wait for a manual analysis of a report before continuing on a schedule. In cases such as these, it would be nice to be able to set the event up, and put it on hold until you are ready to release it for processing. OpenJCS allows for just such a mechanism. It allows you to specify that an event must receive manual confirmation from system management before it can be processed. It also allows you to tag the hold with a label that can be used to remind you why the event is on hold.

File Dependencies – Files are the essence of data processing. It would be nice to be able to schedule not only according to the existence (or non-existence) of a file, but also according to the characteristics of the file (empty, non-empty, open, closed, size, type, permissions, modification date, and so on). OpenJCS allows you to make use of all of these dependency types in your scheduling.

Job Dependencies – When you have a job processing system, the relationships between jobs become extremely important. A job may depend on one or more other jobs completing before it can be launched. Fortunately, OpenJCS provides multiple ways to define job dependencies. It even lets you set up dependencies based on the current state of another job, even if that job is on another system.

Job Concurrency – Sometimes, you may only want a job to execute if one or more other jobs are currently running. OpenJCS allows you to specify job concurrencies that must (or must not) exist before a job can be launched, and these concurrencies can exist within a system, or across several systems. Job concurrencies are maintained via Job Runbooks.

Resource Dependencies – No matter how robust your systems are, they have their limits. Only so much of a resource is available at any time. There is only so much CPU available, only so much memory, so much bandwidth, and so on. The ability to distribute your workload according to resource availability is absolutely critical. Overloaded systems cause response time delays, and underloaded systems waste money. OpenJCS provides a way to define resources, and make your workload dependent on the availability of required resources. OpenJCS has a very flexible resource definition system that you can use to mirror the resources available in your environment. See the Resource section or the resource command for more information about resources in OpenJCS.

Other Dependencies – We have demonstrated lots of dependencies that dictate things that have to happen before a job can begin to execute, but what about if we want to start synchronizing jobs that are already executing? Well, OpenJCS provides mechanisms for doing just that. First off, OpenJCS provides special variables that can be shared between processes, jobs, users, groups and even systems in the domain. You can specify at what level the sharing takes place, and the owner of the variable can specify exactly what kind of access to the variable is allowed, and by whom. This creates unmatched opportunities for information sharing and job synchronization.

The second method of job synchronization available is in the job control shell (jcsh) itself. It provides a **when** command that releases a block for processing when a logical condition becomes true. This means you can effectively create dependencies that are checked while the job is executing, as opposed to being checked before the job can be released for processing.

Batch Environment Definitions

Job – A job is the central unit of work and entity management in this scheduler. A job is a batch process that can be formed by a single command, a series of commands or a file of commands. Once a job is successfully introduced, it is completely separate form the user's process tree, and executes under OpenJCS' batch environment. A job will be executed as a process that by default runs under the ID of the user that launched the job. The ID can be changed via the job's job card or by commands that alter the job before it begins execution. A job is launched via the **job -launch** command.

Serial Queue – A series of jobs that execute sequentially. A serial queue can be created either via the **job -launch** command or by attaching an **–after** dependency to a job.

Jobqueue – A jobqueue is a way to restrict the parallel execution groups of jobs, and to enforce consistent standards on the jobs in the queue. The maximum number of jobs that can execute at one time in a jobqueue can be changed as needed. Some of the restrictions and standards that can be placed on a queue are: minimum job priority a job must have to be admitted to the queue, maximum number of concurrent jobs in the queue, maximum load on the hosts serving the queue, job numbers that will be assigned to the queue, resources demanded by each job in the queue, queue hierarchy, conditions placed on each job in the queue before it is permitted to execute and process execution priority profiles. Jobqueues can also be nested as desired, such that the jobs in a given queue are also counted toward the maximum number of simultaneous jobs in the parent of that queue. Jobqueues are managed via the **jobq** command.

Job Number Family - A family of jobs that will receive similarly formatted job numbers, and will have similar dependencies placed on them. This permits jobs that are related to be viewed together easily. Job Number families are managed via the **jobnum** command.

Job State – In a job's lifecycle, it can pass through several states. The states defined in this system for a job are:

Job State (Primary)	**Meaning**	**Substates (Secondary)**
EXEC	Job Is Active and Executing	**EVAL** – Job is being pre-processed. **INTRO** – Job is being introduced into job input spooler. **EXEC** – Job is executing. **VERIFY** – Finished, verification proceeding.
SUSP	Job Is Active But on Hold	**SUSP** – Job suspended by system management.
SCHED	Job Has Not Yet Started Executing	**HOLD** – Job is on hold per System Management. **SCHED**- Job has unfulfilled dependencies. **RESOURCE** – Job has unmet resource needs. **WAIT** – Jobs priority is lower than the fence for the

		queue, or the queue is already full. **RUNBOOK** – Job is being evaluated according to its runbook. **LOCKED** – Job has been locked by its runbook.
EOJ	Job Ended	**OK** – Job finished successfully. **WARN** – Job finished with warnings. **ERR** – Job finished with errors. **FAIL** – Job failed before finishing. **ABORT** – Job aborted by System Management.

Job Runbooks – A job runbook is a set of constraints, permissions and instructions to be performed at the introduction and conclusion of a job. The runbook is actually a file that contains the list of instructions and permissions, and it is associated with one or more jobs via the **job -runbook** command. In the file, some of the constraints that can be placed on the job are: who can launch the job, what its priority is, when it can run, what jobs it can run with (and which ones it can't), how the job is permitted to log on, what machines are permitted to run the job, what job queue and job family it will use, dependencies that will be attached automatically to the job, and what shell it will use to execute. See the Runbook section for more information about Job Runbooks.

Preprocessors – Preprocessors are processes that read job files as input, use various types of substitutions to create the finished job file as output. The preprocessor could search the job for special reserved words, for example, and substitute in content specified by those reserved words.

Resources

Resources provide a way to restrict concurrency of jobs and simulate various aspects of system load management. For example, a resource can be created to limit the number of jobs accessing a given database at any given point in time, or a resource could be created to keep jobs from being launched when system load is too high, or when high user traffic is expected. A resource is created via the **resource -add** command. Once a resource is created, its use is not automatic. It has to be explicitly requested by the commands that support resource dependencies (**job –launch**, **jobq** and **schedule**, for example). An exception to this is when a resource requirement is set up in a job's runbook (see Job Runbooks).

There are two different allocation methods for a resource. The first way it can be allocated is if the resource has a fixed maximum count attached to it. In this case, when a request is made for that resource, if there are enough unallocated units of that resource available, the request is granted, and the number of units requested are deducted from the number available. If there are not enough units of the resource available, the request is queued until there are enough units available.

The other allocation method for a resource occurs when the resource does not have a maximum count attached to it. Instead, it has an allocation rule attached to it. In this case, whenever the resource is requested, the attached rule is executed, and the results returned by the rule dictate whether the request is granted, denied or queued and retried later. This resource type is very useful for resources that are more aligned with performance metrics, such as system load, CPU usage, available memory, or network traffic levels.

JCS Variables

JCS Variables are similar to environment variables, but JCS variables can be shared between processes, jobs, users, groups, systems or even regions as desired. JCS variables can also be made permanent if desired. This creates opportunities for unparalleled information sharing. For example, a job can set a given JCS variable to a value, and another job that has appropriate access to the variable can read that value, without the jobs having to have any knowledge of each other. Or a job can wait for a given JCS variable to be set to a particular value, without having to know which job or process will be responsible for setting the variable to the desired value. JCS variables can be used to synchronize processes, share information, and set execution values, among other things. JCS variables make cooperative processing easier, even when that cooperation needs to extend between dissimilar systems. JCS variables also provide access security, so that the owner of a variable can specify who has access to the variable, and what access is available (ability to set a variable value, read a variable value or administer permissions for the variable). JCS Variables are managed via the **variable** command.

Regions

A region collection of related hosts. Jobs within a region can share JCS variables and resources. Regions can also be used as assignment targets for virtual hosts. A region can be a collection of individual hosts, subnets, supernets, portions of a DNS domain, or any combination of the above. A region can even be a collection of other regions. One particularly useful use for regions is gathering together systems that perform a similar function so that they can share variables and resources. Regions are managed via the **region** command.

Virtual Hosts

A virtual host is a load balancing device used to distribute jobs and commands among several target machines. This provides additional scalability in the scheduling mix. The user can choose from several load balancing algorithms: round robin, load based, cpu usage based, network traffic based, fewest concurrent jobs, job response time, resource based and based on an arbitrary rule provided by the user. For further information about how virtual hosts are defined, see the **virtual** command.

Once a virtual host is defined, jobqueues, resources and JCS variables can be defined and shared among the target members of that virtual host.

Ask/Reply Facility

The ask/reply facility permits a process to ask system management for information. When a process requests information via the **ask** command, the process is stopped until the request is answered via the **reply** command. Also, the **recall** command will show all outstanding **reply** commands. Finally, the automatic reply facility provides a way to specify answers that should be provided automatically to selected **ask** commands. Automatic replies are specified via the **autoreply** command.

Rulesets

Rulesets are scripts or collections of commands used for various purposes by the scheduling system. Rulesets can, for example, open and close queues, determine how resources should be allocated, determine which host should launch a job or select the queue for a job. Rulesets are managed via the **rule** command.

Object Scope

A unique aspect of OpenJCS is object scope. The scope of an object is the range for which the object is defined. For example, an object can be limited to a user, a job, a host, a region or can have no limits at all (i.e. it will be defined all across the scheduling domain). An object scope has four parts: locality, user, job/process and class. You can specify any combination of of these parts in the scope of an object, or none of them. If no scope is specified for an object, and no default scope is specified for the object, it is assumed to be a global object (by default, only system managers can specify global objects). The locality portion of a scope definition specifies what host or hosts can see the object. Some examples of locality scope are: host, region, virtual host, and domain. The user portion of a scope definition specifies what user or users can see the object. Examples of user scope are: user, group or role. Job/process scope specifies what jobs or processes can see an object. Examples of job/process scope are: job, process, process tree, jobqueue or job number family. Class scope is rather arbitrary in nature, and is not specifically tied to any particular aspect of the scheduling system, which makes it potentially useful for defining relationships that are not immediately obvious, or for partitioning the other scopes into more finely grained units.

A particularly useful application of scope is in the definition of variables and how they are shared. Because of scope definition, you can have two distinct variables with the same name, as long as their scopes do not overlap. For example, you could have a variable named JOBS_REMAINING on host a with its scope set to host a, and a different variable named JOBS_REMAINING on host b with its scope set to host b. The two hosts would only see their own JOBS_REMAINING variable. Or for another example, suppose you wanted to create a variable called ACCT_UPDATING visible only to your Accounting machines. For this, you would first create a region that encompasses your Accounting machines, then create a variable ACCT_UPDATING whose scope is that particular region. Scope is a very useful means of defining who can see what in the scheduling environment.

Default scope for an object is defined via the scope command.

The objects that can be given scope are: variables, resources, jobs, jobqueues, job number families, auto-replies and virtual hosts.

Examples of scope usage in creating objects:

variable shutdown%global=0

Creates a global (visible to the whole domain) variable called shutdown, and assigns it a value of 0. By default, this command will work only if the user issuing it is a system manager.

scope resource:hosta* -for host:hosta -to host:hosta
resource –add –limit 3 hosta_db

Creates a resource named hosta_db local to host hosta (assuming the resource command was issued from hosta), and allows up to 3 of this resource to be allocated at any time. Note that the scope command says that any resource created on hosta that begins with hosta will be limited to that host only.

Architectural Fundamentals

OpenJCS relies on master scheduler daemons (openjcsd), a client daemon on each host participating in the scheduling domain (jcsjobd) which also doubles as the job controller/dispatcher for that host, and one or more service listeners for the services provided by OpenJCS. The services and daemons used by OpenJCS are as follows:

JCS Master (**openjcsd**) – The master controller for an OpenJCS domain. This daemon handles domain membership, region administration, service map distribution and permission lists. Every host will have an openjcsd running on it, and communication to the other OpenJCS daemons pass through openjcsd to get to the target daemons. There can only be one active master openjcsd for the domain, but others in the domain will be defined as standby masters, and will also serve as pass-through systems to the current master when not active.

JCS Execution (**jcsjobd**) – This daemon is responsible for every process that is dispatched from OpenJCS to execute on the host.

JCS Heartbeat (**jcshbd**) – This process sends a UDP heartbeat request to openjcsd and each of the service masters periodically, and if its heartbeat fails to the current master, but not to one of the candidate failover masters, it calls for a master election. This process is a child process to jcscomd.

JCS Scheduling (**jcssched**) – Provides the repository for schedule checking, and dispatches events to target clients.

JCS Variables (**jcsvard**) – Maintains all JCS variables across the domain.

JCS Resources (**jcsresd**) – Maintains all JCS resources, and controls allocation of same.

JCS Replies (**jcsaskd**) – Maintains list of pending replies, as well as all automated replies.

JCS Virtual Hosts (**jcsvirtd**) – Maintains Virtual Host lists, rules and tokens, and allocates Virtual Host workloads.

JCS Calendars (**jcscald**) – Maintains all information regarding OpenJCS calendars.

JCS File Transfer (**jcscpd**) – Performs file transfers between hosts in OpenJCS.

The listeners communicate via SOAP calls. The call is first routed to the jcsjobd for the current host, which then uses its service map to decide where to route the request.

The location and quantity of listeners defined for a domain is contained in the openjcs.conf file. The default is one listener per service on each host in the domain, with daemons for virtual hosts and regions being kept on the current master.

System Level Components

In order for OpenJCS to perform its job properly, certain system level components must be in place. Some of these components are:

- **DBMS** – As of this writing, the only DBMS that OpenJCS will use to maintain its job execution history and component values is PostgreSQL. Other databases may be supported in the future if there is sufficient demand to warrant the development effort involved.
- **Apache** Web Server – This is required if the Web GUI for OpenJCS is used (highly recommended).
- **C++** – Used to implement the performance-critical aspects of OpenJCS (memory caches, for example).
- **Python and Zope** – Used to implement the less performance-critical aspects of the OpenJCS system.
- **Java** Virtual Machine – Another requirement if the Web GUI is used. It is also required on the client machine if the client GUI is used.
- **PHP** – Used in the Reporting GUI.
- **ssh** – Used to secure the communications between clients and services.

System Object Definitions

Scope
Job
Jobqueue
JobNumber
Reply
AutoReply
Variable
Resource
Region
VirtualHost
VirtualHostMember
Rule

System Master Election and Failover

An OpenJCS domain is controlled by the system master, which is elected by all hosts in the domain. Whenever a host cannot reach a master or service provider, it calls for an election on that master or service. The election is conducted on a "black ball" basis. That is, first the hosts "black ball," or vote out, prospective masters they cannot get a response from within the timeout interval (default is 30 seconds). If one prospective master has fewer "black ball" votes than the others, it becomes the master. If not, all the hosts tied for fewest "black ball" votes run in a response time election. The response times in milliseconds to the original "black ball" election for the candidates are tallied, if one has a shorter maximum response time than the others, it becomes the master. If no clear winner is decided at this point, the prospective masters tied for the lowest maximum response time are ordered by their appearance in the elector list for the service, and the first host in the list that is found among the remaining eligible prospects is selected as the master.

Notes

Order of dependency resolution:

1) Except
2) Until
3) Date Dependencies (at, between, deadline, every, from, to, restart, in, on)
4) Job Dependencies (after, job, with, without)
5) File Dependencies (file)
6) Logical Dependencies (parm, when, whenever, if, unless, until)
7) Ask
8) Resource Dependencies (resource)

Priority for Job modifiers:

1) permissions from job runbook
2) modify command
3) job command
4) launch command
5) jobcard
6) jobq
7) virtual server
8) jobnumber

Commands

answer [{-host *hostname*|-virtual *vmname*}] *pid replystring*

answer will be a symlink to reply

ask [–timeout *nnn*] [-default *answer*] [-maxlen *len*] [-minlen *len*] [-edit *regexpr*] [-[no]echo] [-hint *hint_text*] [{-host *hostname*|-virtual *vmname*}] *question_text*
ask [{-host *hostname*|-virtual *vmname*}] –cancel *pidnum* [-error *errnum*] [-message *msgtext*]

Create a pending reply. The command prints the response received. If echo is enabled (default), the question_text is echoed to stderr. Timeout is the maximum amount of time in minutes that the ask command will wait for an answer. If the request times out, then the default answer will be returned. If no default is given, a null answer will be returned. Minlen and maxlen specify the minimum and maximum lengths a response can be, and edit specifies a regex that will be used to check the format of the answer. If the answer fails the check, the ask command is re-posted. –cancel cancels a pending reply and forces it to error out. errnum specifies the error status that will be returned by the ask command, and msgtext specifies the error message that will be returned to stderr. Default errnum is 1 and default text is "Request cancelled by system management."

autoreply {*name*|-all}[%*scope*] -show
autoreply {*name*|-all}[%*scope*] -delete
autoreply *name*[%*scope*] [-file *filename* […]] [-keep] -response [*keytext*=]replystring[;…]

Sets up, deletes or displays an automatic reply as well as optionally setting the scope for which the automatic reply is defined. –file refers to the file that was used to stream a job via the launch command. Keytext= specifies that the automatic reply will be used only for a pending reply whose text matches the regex specified by keytext. Multiple automatic replies for a process can be specified by separating the responses with semicolons (no unquoted spaces in either the keytext or replystring). –keep specifies that the automatic reply will not be deleted after it is used (default is delete the automatic reply after it is exhausted). For the definition of scope, see the scope command

calendar {-create|-modify} *calendarname calendarfile*
calendar {-delete|-default|-show} *calendarname*

Calendar maintenance.

datecalc [-format *format*] [-calendar *calendarname*] [increment […]]
Where

increment::=[[+|-]*nnn* [*interval*]|*nlsdate*]
and nlsdate is a date in the given calendar's default format

Displays a date calculation. Default interval is days. If no calendar is specified, current system calendar is assumed, and default date format is the current NLS format (must be quoted if the format includes blanks) if available, and mm/dd/yyyy@hh:mi otherwise.

duration –from *dateexpr* [-to *dateexpr*] [-units *unitname*] [-int|-real]

Determine the duration in temporal units specified by unitname between two date expressions. Default for –to is now, and default unit is whole days.

finfo *filename*,*parmnum*

Returns info about a file.

jcsallow {-permit|-forbid} *permid* [*identifier* […]] {–command "*cmdimage*"[…]|-role *rolename* […]
jcsallow –show [-perm *permid*] [*identifier* […]]
jcsallow –rename *permid newpermid*
jcsallow –delete *permid*

Where
identifier::= {{-host|-virtual}=*hostname*|-region=regionname| -jobname=*jobname*|-user=*userid*|-login=[*jobname*,]*user.group*| -group=*groupname*| -role=*rolename*| -jobnum=*jobnum*| -file=*filename*|-directory=dirname}

Sets execution permissions for various commands. All commands are permitted to system management and cannot be forbidden to system management. By default, **job -launch** commands are permitted to everyone, but only for jobs that login to the same user, group and host as the user launching the job, and **job -show** is permitted to everyone. Also by default, **ask** commands are permitted for everyone, and **reply** is permitted when the requestor's user, group and host match the user attempting the reply. Permissions are applied by permid. Permid can cotain alphanumeric characters plus ",", ".", "/", "+", "-", "_", "=", "@", "#", and ":".

By default, the following commands are permitted to everyone:

datecalc
finfo
variable (for variables in the user's scope)

job –show
job –launch (if the job logs on as the user's userid and group on the user's login host)
jobinfo
jobnum –show
jobnuminfo
jobnums
jobq –show
jobqinfo
preprocess
recall
replyinfo
resinfo
resuser
schedule
showsched

And by default, the following commands are reserved for system management only:

autoreply
calendar
jcsallow
job (except job-launch and job-show)
jobnum (except jobnum –show)
jobperms
jobq (except jobq –show)
region
resource
rule
stdlist
virtual

Roles are defined in the jcs.config file. The roles manager, supervisor, operator and monitor are pre-defined. All except manager can have their capabilities modified in the jcs.config file.

jcsconfig {-load|-save} [-file *filename*]
jcsconfig {-show}

Loads, saves or displays the current running configuration. This command can be issued by System Management only.

jcsmaster {-elect|-show|*hostname*} [-service *serviceid*]

Service master control. The –show option shows the list of current service masters. The option –elect forces an election among the eligible candidates for master for the service. If a hostname is provided, then that host is assigned as the master for the service.

job *selector* […] *mods*
job *selector* […] –read
job *selector* […] –runbook *filename* [-show|-delete]
job [*selector* […]] -show [*showformat*]
job [*selector* […]] -stats [*stats_selectors*] [*stats_format*]}
job -launch [-f *filename*][...] [*schedule_modifier*[...]][*job_modifier*[…]]

Where:

selector::= -select {{host|virtual}=*hostname*|-region=regionname| jobname=*jobname*|user=*userid*|login=[*jobname*,]*user.group*| group=*groupname*| jobnum=*jobnum*|state={EXEC|OK|SUSP|HOLD[:*holdtype*[,…]] |WARN|ERR|FAIL|EVAL|RUNBOOK}|file=*filename*|directory=dirname}

mods::= {-cancel|-continue|-restart [*label*|*linenum*|prev|next|first]|-flush|-abort|-inpri *nnn*|-suspend|-resume|-hold *desc*|-release *releasespec*| {*schedule_modifier*|*job_modifier*[…]}}

releasespec::=[now|hold=*holdtype*]

holdtype::= {*hold_desc*|*schedule_modifier*|*job_modifier*}

showformat::= –format {[0-9]|short|long|deps|*fieldname*[:*width*[,…]} [-sort *fieldname*[,…]] [-[no]head]

stats_selectors::= -stats [-start *startdate*] [-stop *stopdate*]

stats_format::=–format {[0-9]|short|long|*fieldname*[:*width*[,…]} [-sort *fieldname*[,…]] [-rollup *fieldname*[,…]]

Manages an individual job or group of jobs. Selection criteria can be mixed and repeated as desired. Selection criteria of the same type are logically OR-ed together, and the results logically AND-ed together with selection criteria of differing types (Example: -select user=root –select state=EXEC –select user=ckemp –select state=SUSP means find all jobs running as root or ckemp whose current state is EXEC or SUSP). If jobnum matches a job number family, the action is taken for the entire family, otherwise jobnum is assumed to be a single job number or JSID. –cancel cancels a pending job. To cancel an executing or suspended job, use –abort. –continue restarts a job that stopped because of an error. Execution of this job continues from the next job step. –restart restarts a stopped job at some other point in the jobstream. –flush deletes a stopped job. –inpri changes the job's input priority. If the job's input priority is less than or equal to the job fence for the job's queue, the job will not be introduced. –suspend places an executing job in a sleep state, and –resume resumes a suspended job. –hold causes the job to be prevented from executing, and *desc* will show up under dependencies on a –show for the

job. –release releases a hold on a job, and –release now removes all dependencies. –show shows currently executing jobs. If a custom –format is used, the fields must match the fieldnames in the jobinfo intrinsic below (use field names, not numbers).–read prints the contents of the job, and indicates the current execution step of the job. –stats gives wall time and CPU usage. Job -launches a job. If no filename is specified, the command takes the job to be launched from STDIN. If multiple filenames are specified, they are launched as a serial queue. All dependencies are attached to the first job in the queue. Job modifiers are applied to all jobs in the queue. Default priority for a launched job is 20.

jobinfo {*jobnum*|*jobname*} *parmnum*

Returns info about a scheduled job. See the library routine jobinfo for the list of parameters and information returned by this command.

jobnum [*jobfamily*[%*scope*] [-spec *jobspec*] [-select *selectrule*] [-start *nnn* -stop *nnn*] [-restart|-delete] [*schedule_modifier*|*job_modifier*[…]]]
jobnum [*jobfamily*[%*scope*]] -show

Creates a smart job number family. spec must not contain white space, and each element of the spec must be alphanumeric or one of: "@", "#", "-", "_", "+", ":", ",", ".", "/", a date format element (see date command) or "{}" to indicate the JSID. If the specification does not include "{}", the JSID will be appended to the specification. Example of a job spec: #J{}-ACCT. jobnum by itself shows all current job number families. Default spec is jobfamily-{}. Select specifies that a launched job will be placed in this job number family if its launch does not already specify one and if the job fits selectrule. The values supplied for start and stop must be positive integers such that start<stop. Also, if a start/stop is provided for a queue, it must not overlap the start/stop for any other job number family. Once a job number range is exhausted, it will restart from the beginning. A –restart will force job numbering to restart from the beginning. If a start/stop combination is not supplied, JSIDs will be assigned from all JSIDs not assigned to a job number family. The unassigned job numbers can be restarted by doing a -restart on reserved job number family DEFAULT. The spec for DEFAULT is #J{}. Job number families are always local to a host machine. For the definition of scope, refer to the scope command.

jobnuminfo *jobfamily*,*info*

Returns information about a job family. See the library routine **jobnuminfo** for the parameters passed to and information returned by this command.

jobnums *spec*

Returns job numbers of all jobs fitting spec.

jobperms [*selector* […]] –set [-temp|-perm] *permid ident perm* […]
jobperms [*selector* […]] {-add|-remove} *permid* [*ident*] *perm* […]
jobperms {-show [*permid*]| |-delete *permid*}

Where:

selector::= -select {{host|virtual}=*hostname*|-region=*regionname*| jobname=*jobname*|user=*userid*|login=[*jobname*,]*user.group*| group=*groupname*| jobnum=*jobnum*| file=*filename*|directory=*dirname*}

ident::=[*role*,]{*user*|~|*}[.{*group*|~|*}][@{*host*|~|*}[:{host|virtual|region}]

perm::={-permit|-forbid|-force|-delete} {schedule_modifier|job_modifier|runas *userid*[.*group*]|password| nopass|admin}

Sets job permissions. Selectors (if provided) limit the target range across which the permission is enforced. For example, -select region=accounting –select directory=/opt/acct/jobs will affect jobs launched from the directory /opt/acct/jobs on all machines in the accounting region. Selectors that reference a login, user or group refer to the login of the job being administered. If no selectors are provided, then the permission takes effect across the entire domain. An identifier refers to the person attempting to launch or administer the job. "~" in an identifier means that part of the user's login must match the corresponding part of the job being launched for the permission rule to be applied (Example: ~.~@~ indicates that the permission will be applied only if the user's userid, group and login host match that of the job the user is trying to administer). Default is a job can only be administered by system management and by the user and group that launched it. System management cannot be forbidden administrative control over a job. Permissions are applied to a job in order of their permid. Only system management and executing runbooks can use the jobperms command.

jobq -create [*queuespec*[…]] *queuename*
jobq -alter [-add|-remove|-set] *queuespec*[…] *queuename*
jobq {-delete|-open|-close} *queuename*
jobq -show [*formatspec*] [*queuename*[…]]

Where

queuespec::= [-select *selectrule*] [-maxload {*nnn*|*jcsvar*}] [-limit {*nnn*|*jcsvar*}[per {source|target} {user|group|jobname|class|login|host|virtual|region}]] [–fence {*nnn*|*jcsvar*}] [–pri {*min*|*jcsvar*},{*max*|*jcsvar*}] [-cutoff *cutoffrule*] [–parent *queuename*] [-slice {*msecs*|*jcsvar*},{*msecs*|*jcsvar*}] [-nice {*nice*|*jcsvar*}] [-incr *prichg*[%]] [*schedule_modifier*[...]] [*job_modifier*[…]]

queuename::=name[%*scope*]

formatspec::= [-v|-[0-9]|-format *fieldname*[:*nnn*][,…]]

Manage a jobqueue. Main queue is MASTER. It defines the maximum total size of all user queues combined. Default queue is DEFAULT. SYSTEM queue is reserved for system administration, and exists outside MASTER. Closing a queue does not affect any running jobs, but simply forbids admission of any more jobs to the queue, regardless of priority. If a cutoff rule is specified, that rule is executed and the queue is shut if the status returned from the rule is greater than zero. Select specifies that a launched job will be placed in this queue if its launch does not already specify a queue and if the job fits selectrule. Maxload specifies a system load level at which the queue will be shut down automatically. Limit specifies the maximum number of active (EXECUTING and SUSPENDED) jobs that may occupy the queue at any point in time. Fence specifies the minimum introduction priority a waiting job must have before being considered for admission to the queue (default is 10). Pri specifies the minimum and maximum process priority for the queue. Parent specifies the queue which is immediately above this queue. Note that all active jobs in a queue count toward the parent queue limit. Slice specifies the minimum and maximum number of CPU milliseconds per second that all jobs combined in the queue may use. Note, however, that in no case will a process' priority be placed outside the process priority limits for the queue. Nice specifies a nice value that will automatically be applied to every job in the queue. Incr specifies the increment by which a process' priority will be changed (up or down as needed) when it goes outside the usage limits specified for the queue. jobq by itself displays all jobqueues. Note that a jobqueue name must be unique across all machines that serve it. That is, if a jobqueue is set up for a virtual host, none of the target hosts can have an existing queue with that name, and if a jobqueue is set up on a local system, none of the virtual hosts for which it is a target may have a jobqueue of the same name. For the format of a scope specification, see the scope command.

jobqinfo *queuename*,*parmnum*

Returns info about a jobqueue. See the library routine **jobqinfo** for the parameters passed to and information returned by this command.

pause schedule_modifier[…]

Waits until the schedule modifiers specified are all true. If the command exits without all the dependencies being filled (for example, hitting a specified –deadline, or specifying an –if condition that is not true), the command exits with a nonzero status. If all dependencies are met, then the command exits with a zero status. This command allows the user to create a more fine-grained control of portions of a job as it executes.

preprocess [-as *userid.groupid*] [-[no]nest] [-processor *processor*] [*file* …]

Runs input through a preprocessor.

recall [-host {*hostname*|all}|-virtual {*vmname*[%*scope*]|all}|-region {*regionname*|all}]

List pending replies posed via **ask**. Recall items are listed in the following format:

rpid/*hh*:*mi*/*host*:{H|V|R}/*userid.group*/*jobnum*/*question_text*?[(hint)]

If no filters are specified, then only pending replies for the current host are shown. For format of the scope portion of the command, see the scope command.

recap [-q|-n] [*scope*]

Displays the current schedule and/or performs a system status check. If –q is specified, only the status check will be performed. If –n is specified, the schedule is displayed without performing a status check. Default is perform both status check and schedule display. If a *scope* is displayed, only the events fitting that scope are displayed. To see the format of the scope specification, see the scope command.

region *regionname* [[-add|-set|-remove][*file*[…]]|-delete]

Creates a region of related systems. The lines of the region file have the format:

{{include|exclude} {system|virtual|region|network|dnsname}=spec[,…]|#comment}

If no region files are specified, then the elements for the region are taken from stdin.

reply [{-host *hostname*|-virtual vmname[%*scope*]}] *rpid reply*

Answer pending reply posed via **ask**. For the format of the scope portion of the command, refer to the scope command.

replyinfo *replyspec*,*parmnum*

Returns info about a pending reply. See the library routine **replyinfo** to see the parameters passed to and information returned by this command.

resinfo *resource*,*parmnum*

Returns info about a resource. See the library routine **resinfo** to see the parameters passed to and information returned by this command.

resource -pending [-ppid *ppid*] [-pid *pid*] [{-host *hostname*|-virtual *vmname*[%*scope*]}]
{-deny |-return|-modify {-timeout *nnn*|-pri *priority*|-resource [+|-]*resource*spec,…}}
resource –get [-nowait|[-timeout *nnn*|-pri *priority*][…]] *resource*[:*count*|all] [...]
resource –return {-all|-last|*resourcespec* […]}
resource {-add|-delete|-modify|-freeze|-unfreeze} [*properties*] *resource*
resource –access *resource* [-owner [+|-][r][x][w]] [-member [+|-][r][x][w]] [-other [+|-][r][x][w]] [-scope *scope* [+|-][r][x][w]]
resource –show [-pending] [{-host *hostname*|-virtual *vmname*}] [*scope*] –sort *sortspec* [*resource*[…]]

Where

properties::= [–d *description*] {–limit {*max*|*jcsvar*}|–rule *command*} [–min {*priority*|*jcsvar*}] [–max {*priority*|*jcsvar*}] [-contains *resourcespec* [,…] [-parent *resource*] [{-host *hostname*|-virtual *vmname*}] [-alloc *sortspec*[,…]]

resourcespec::=*resource*:{*num*|all}

resource::=*resourcename*[%*scope*]

sortspec::={AGE|PRIORITY|COUNT|REQUESTS|AVAIL|*sortrule*}[:ASC|:DESC]

Resource control. Resource -pending works on pending resource requests. Resource –get requests a resource. The -nowait form of the command gets a resource immediately. If the request is successful, the command returns a 0. If not, it returns an error code. A resource –get without –nowait will wait for the resource(s) specified before it returns a status. If it is not successful within the time allowed, or if the request is denied, an error code is returned. Resource -return –all returns all resources requested by this process. Resource -return –last returns all resources requested by the last need. A process exiting returns all resources it requested. By default, a process can only get resources for groupings of which that process is a member. The –access form of the command defines what access is granted to the resource. An access of "r" means the scope specified can do a show on the resource. An access of "x" means –gets and –returns are permitted. An access of "w" provides the capability of administering the resource (add, delete, modify, grant, refuse).

The –add/-delete/-modify form of the command manages the resources themselves. If max<=zero or not specified, resource has no specific limit. In this case, a rule MUST be provided. Exit 0 from rule means request granted, anything else denied. For example, using system load as a resource would allow a user to specify that a job can be run only when system load is below 3. If no count is specified, or if the count is set to <=0, then rule is executed to see whether to allocate the resource or not. If count >0, then rule is executed only when more than one process is waiting for a resource, and rule determines which process is allocated first. The names of all resources available on any given host must be unique, and a resource name can only contain alphanumeric characters plus "." and "_". If a parent resource is specified, every time this resource is incremented or decremented, the corresponding parent resource is treated likewise. For the format of the scope portion of the command, refer to the scope command.

resuser *resource pid parmnum*

Returns info about a user of a resource. See the library routine **resuser** for the parameters passed to and information returned by this command.

rule *rulename*[%*scope*][(*parm*[=*defaultvalue*][,…]) –file filename
rule *rulename*[%*scope*][(*parm*[=*defaultvalue*][,…])[=]
do
.
.
.
[return *value*]
.
.
.
done

rule -show [{*rulename*|*pattern*}[%*scope*]]
rule -delete *rulename*[%*scope*]
rule –access *rulename*[%*scope*] [-owner [+|-][r][w][x]] [-member [+|-][r][w][x]] [-other [+|-][r][w][x] [-scope *scope* [+|-][r][w][x]]

Where

Defines a rule for use in resource and job control commands. For the format of the scope portion of the command, refer to the scope command.

schedule [-name *jobname*] [-user *userid*] [-group *groupname*] [-jobnum *jobnum*]
[*schedule_modifier*][...]
[*job_modifier*][...] –cmd *cmdimage*
schedule [-name *jobname*] [-user *userid*] [-group *groupname*] [-jobnum *jobnum*]
[*schedule_modifier*][...]
[*job_modifier*][...] -delete
schedule [-name *jobname*] [-user *userid*] [-group *groupname*] [-jobnum *jobnum*]
[*schedule_modifier*][...]
[*job_modifier*][...] -edit
schedule [-name *jobname*] [-user *userid*] [-group *groupname*] [-jobnum *jobnum*]
[*schedule_modifier*][...]
[*job_modifier*][...] -modify
schedule [-name *jobname*] [-user *userid*] [-group *groupname*] [-jobnum *jobnum*]
[*schedule_modifier*][...]
[*job_modifier*][...]
do
.
.
.
done

Schedules an event.

scope [*object_type*:namepattern] [–for *scope*] {[-to] *scope*|-delete|-show}

Where

object_type::={all|variable|resource|virtual|job|jobnumber|jobqueue|autoreply}

scope::={*localityscope*|*userscope*|*jobscope*|*subclass*}

localityscope::={local|region:*regionname*|host:*hostname*|virtual:*vmname*|global}
[/*userscope*|/*jobscope*|/*subclass*]

userscope::={user[:*userid*]|group[:*groupid*]|role:*rolename*}[/*jobscope*|/*subclass*]

jobscope::={temp|ptree[:*pid*]|job[:*jobname*]|jobnum[:*jobnum*]|
jobq[:*jobqueuename*]}[/subclass]

subclass::=class:*classname*

Sets the default scope for a scheduling object.

showsched [{-host *hostname*|-virtual *vmname*}] [-from *fromdate*] [-to *todate*] [-calendar *calendarname*]

Prints a schedule.

stdlist [*hostname*:]*jobname*

Prints a job's stdlist (stderr + stdout).

variable [–indirect] *jcsvar*[%*scope*][=*value*]
variable -show [{*jcsvar*|*pattern*}[%*scope*]]
variable -delete *jcsvar*[%*scope*]
variable –lock [-conditional|-unconditional] *jcsvar*[%*scope*]
variable -unlock {-all|*jcsvar*[%*scope*]}
variable –access *jcsvar*[%*scope*] [-owner [+|-][r][w][x]] [-member [+|-][r][w][x]] [-other [+|-][r][w][x] [-scope *scope* [+|-][r][w][x]]

Where

jcsvar::=*variablename*["["*index*[,…]"]"]
Example: CPUtime["myhost",jobnumber]

Sets, shows or deletes a JCS variable. Default is local variable for the current process tree. The advantage of JCS variables is that they can be shared between processes, jobs or even multiple systems. JCS variable names can contain alphanumeric characters, as well as "_", "-" and "." A JCS variable name must start with either an alpha character or "_". The first form (variablename=value) sets a value for a JCS variable, and creates the variable if it does not already exist. –show displays ALL JCS variables matching the criteria specified. The variables are displayed in a variable=value format, one per line. –delete deletes a JCS variable. If no assignment is made in a variable command, and neither –show, -delete, -lock, -unlock or –access are specified, the command returns a single value for the variable specified. For the format of the scope portion of the command, please refer to the scope command.

virtual –show [*vmname*[%*scope*]]
virtual –delete *vmname*[%*scope*]
virtual –modify *vmname*[%*scope*] {-param
{method|minfree|state|failover|overload|limit}= ”*valuestring*”|-member
name=”*memvals*”}[…]]
virtual –edit [*vmname*[%*scope*]]
virtual *vmname*[%*scope*]
begin
[method [weighted] [*balance_method*]
[minfree {*nnn*|*jcsvar*}]
[state {enabled|disabled|halted|failed}]
[failover *vmname*]
[overload *vmname*]
[limit [per {user|group|jobname|file}] {*nnn*|*jcsvar*|*rule*}]
[maxqueue {*nnn*|*jcsvar*|*rule*}]
[options {*job_modifier*|*schedule_modifier*}[…]
[runbook *runbookfile*]

{*member_spec* […]|*region_spec*}

end

Where

member_spec::= member [virtual] *name* [weight [rule] *weight*] [priority [rule] *priority*] [limit {*nnn*|*jcsvar*|*limit*}] [state {enabled|disabled|halted|failed}] [runbook *runbookfile*] [resource *resourcespec*] [availability *rule*] [options “{*schedule_modifier*|*job_modifier*}[…]}”

region_spec::= region *regionname* [weight [rule] *weight*] [priority [rule] *priority*] [runbook *runbookfile*] [resource *resourcespec*] [availability *rule*] [options “{*schedule_modifier*|*job_modifier*}[…]}”

balance_method::= { roundrobin|load|cpu|network|jobs|jobresp|resource *resourcespec*|*rule*}

resourcespec::=*resource*:{*num*|*jcsvar*|all}[…]

resource::=*resourcename*[%*scope*]

Sets up a load balancing virtual host. Only system management can set up a virtual host. Weighted means that the load balancing method will take into consideration the weight assigned to the target hosts in its calculation of the next host to be used. For example, on a virtual host using a load balancing method of weighted load, if host_a has a current load of 3 and a weight of 10, and host_b has a load of 5 and a weight of 50, then the next load of work will be assigned to host_b because host_b’s load would have to be 5 times that of

host_a before host_a would be assigned work. Weighted load balancing cannot be used with the resource load balancing method. You can specify resource at the virtual server level or at the member level, but not both. Membership in a virtual host can be specified by member host or region, but not both. If a virtual host is set up by region, only one region can be specified for the virtual host. You can, however, create a virtual host made up of other virtual hosts, each containing a different region. If priorities are used, each member of the highest priority must be within minfree jobs of its limit before any member in the next lower priority group can be used. If an availability rule is specified, that rule is executed to see if the member host is available for job assignment. For the format of the scope portion of the command, refer to the scope command. **Note**: the scope for a virtual host **cannot** be limited to a single host.

Virtual Host States. A virtual host state of enabled means that at least one target host available to process jobs. A state of disabled means that existing jobs will be processed, but no new jobs are being accepted. Blocked means that no target machines are available to process jobs. Halted means that the virtual host has been aborted, and the state of jobs on the target machines cannot be ascertained. Aborted means all jobs on target machines for this virtual host have been aborted, and the virtual host Halted. Failed means the virtual host has been failed over to its failover virtual host.

Member Host States. A member host state of enabled means that the host is available to process jobs. A state of disabled means that existing jobs will be processed, but no new jobs are being accepted. Blocked means that the host has reached its job capacity. Halted means that the host state and the state of its jobs cannot ascertained. Aborted means all jobs on target machines for this host have been aborted, and the host disabled from accepting new jobs.

vsh [*user@*]*vhostname*[%*scope*] [-l *user*] [-e *mech*] [*cmdimage*]

Executes the command on the virtual host indicated. Supported values for *mech* are rsh and ssh. For the format of the scope portion of the command, refer to the scope command.

openjcs.conf File

The openjcs.conf file contains the startup configuration for OpenJCS. When the file is modified, send the openjcsd daemon a –HUP signal to force it to re-read the file.

Here is a basic layout of the jcs.config file:

Sections:

1. Aliases
    1.1. Section Header: [ALIASES]
2. Properties
    2.1. Section Header: [PROPERTIES]
    2.2. Entries
        2.2.1. property_name=property_value
        2.2.2. Valid Property Names
            2.2.2.1.jcsuser
                2.2.2.1.1. Default User Login for OpenJCS Daemons
                2.2.2.1.2. Default is root
            2.2.2.2.jcsgroup
                2.2.2.2.1. Default Group for OpenJCS Daemons
                2.2.2.2.2. Default is wheel
            2.2.2.3.init
                2.2.2.3.1. Command to execute when OpenJCS Starts
                2.2.2.3.2. Default is :
            2.2.2.4.fin
                2.2.2.4.1. Command to Execute When OpenJCS Shuts Down
                2.2.2.4.2. Default Is :
            2.2.2.5.wrapper
                2.2.2.5.1. Wrapper Process for Communication
                2.2.2.5.2. Default is ssh (must be in PATH)
            2.2.2.6.auth
                2.2.2.6.1. Authentication Method
                2.2.2.6.2. Default is passwd
            2.2.2.7.session_timeout
                2.2.2.7.1. Session Timeout Duration in Minutes
                2.2.2.7.2. Default Is 30
                2.2.2.7.3. 0 Disables Session Timeouts
            2.2.2.8.cache
                2.2.2.8.1. Defines What Is Cached on Each System
                2.2.2.8.2. Default is var, resource, job
                2.2.2.8.3. Available
                    2.2.2.8.3.1.var
                        2.2.2.8.3.1.1. JCS Variable Tables
                    2.2.2.8.3.2.resource
                        2.2.2.8.3.2.1. Resource Listings and Queues

2.2.2.8.3.3.job
2.2.2.8.3.3.1. Executing Job Tables
2.2.2.8.3.4.jobnum
2.2.2.8.3.4.1. Job Number Families
2.2.2.8.3.5.jobq
2.2.2.8.3.5.1. Job Queues
2.2.2.8.3.6.virtual
2.2.2.8.3.6.1. Virtual Host Definitions
2.2.2.8.3.7.ask
2.2.2.8.3.7.1. Pending Replies and Auto-Replies
2.2.2.8.3.8.calendar
2.2.2.8.3.8.1. Calendar Definitions
2.2.2.9.root_gui
2.2.2.9.1. Denotes Whether root Logins Via GUI Are Permitted
2.2.2.9.2. Default Is OFF (Not Permitted)
2.2.2.10. root_web
2.2.2.10.1. Denotes Whether root Logins Via Web Interface Are Permitted
2.2.2.10.2. Default Is OFF (Not Permitted)
2.2.2.11. remote_auth
2.2.2.11.1. Denotes Whether Re-Authentication Is Required When Working On Remote Host
2.2.2.11.2. Values
2.2.2.11.2.1. ALWAYS
2.2.2.11.2.2. NEVER
2.2.2.11.2.3. ONCE
2.2.2.11.3. Default Is ONCE (Must Re-Authenticate First Time Only)

3. Logging
   3.1. Logging Facility Definition
      3.1.1. Section Header: [LOGGING]
      3.1.2. Notes
         3.1.2.1.Need to be able to define whether it goes to syslog, local logging, both or neither
         3.1.2.2.Need to be able to define logging levels for various message types (examples: pending reply, job failure, job introduction)
         3.1.2.3.Possibly an external logging processor?
4. Services
   4.1. Service Definitions
      4.1.1. Section Header: [SERVICES]
      4.1.2. Entries
         4.1.2.1.service_type:{job|schedule}:keyword[…]
         4.1.2.2.Examples
            4.1.2.2.1. reply:schedule:ask
            4.1.2.2.2. jobq:job:jobq:jobqrule
   4.2. Service Daemons
      4.2.1. Section Header: [DAEMONS]
      4.2.2. Entries

4.2.2.1.daemon_name:service[…]
4.2.3. Examples:
4.2.3.1.jcsaskd:reply
4.2.3.2.jcsjobd:jobq:jobnumber
4.3. Service Targets
4.3.1. Section Header: [TARGETS]
4.3.2. Entries
4.3.2.1.service_type:servicename:{host|region|virtual|global}:loc_name:target_host[,…]:port
4.3.2.2.Service Types
4.3.2.2.1. jobq
4.3.2.2.2. jobnumber
4.3.2.2.3. variable
4.3.2.2.4. resource
4.3.2.2.5. jcsallow
4.3.2.2.6. reply
4.3.2.2.7. calendar
4.3.2.2.8. region
4.3.2.2.9. schedule
4.3.2.2.10. rule
4.3.2.2.11. virtual
4.3.2.2.12. master
4.3.2.3.Examples
4.3.2.3.1. jobq:DEFAULT:global::hosta:8666
4.3.2.3.1.1.The jobq DEFAULT will be managed for all hosts by hosta, port 8666
4.3.2.3.2. resource::virtual:acct:pactdb01:27365
4.3.2.3.2.1.All resource allocations for the virtual host acct will be handled by host pactdb01, listening on port 27365
4.3.2.3.3. jobq:::host:~:9999
4.3.2.3.3.1.Each host will manage its own jobqueues, and the listener for the service will be on port 9999
4.3.2.3.4. ::global::master01,master02,master03:8686
4.3.2.3.4.1.All services will be managed by host master01, with failover on master02 and master03, all listening on port 8686

5. Scheduling Domains
5.1. Section Header: [DOMAINS]
5.2. Parameters
5.2.1. HIERARCHICAL=[on|off]
5.2.2. DELIMITER=char (Default is .)
5.2.3. PARENT=parentserver[:port]

5.2.4. Domain Name
5.2.4.1.DOMAIN=domainname
5.2.5. Master
5.2.5.1.MASTER=masterserver[:port]

Job Runbooks

Introduction

Runbooks are ways of specifying how and when a job can run, and what sort of processing needs to be done before the job is launched and after it concludes. A runbook is simply an ASCII file that is related to a job via the job –runbook command. The file itself contains aliases, selectors indicating the conditions under which various sections of the runbook are to be performed, and the executable sections of the runbook.

Parts of Runbook File

- User Aliases
- Job Aliases
- Concurrency Aliases
- Scheduling Aliases
- Calendar Aliases
- Directory Aliases
- File Aliases
- Source Machine Aliases
- Target Machine Aliases
- Login Aliases
- Logical Aliases
- Selectors
- Launch Rules

User Alias:

USER_ALIAS *name* =
{*userid*|*|~}[.*group*|*|~][@{*host*|*|~}[:{HOST|VIRTUAL|REGION}]] [[+|-]…]
Examples:
USER_ALIAS admin=root-root@lab01-root@lab02-root@lab03
USER_ALIAS acct=*.acct-nobody.acct-
www.acct+*.fin+*.sys+*.*@acct:REGION

Job Alias:

JOB_ALIAS *name*= [{*jobname*|*|~,}]{*userid*|*|~}[.*group*|*|~]
[@{*host*|*|~}[:{HOST|VIRTUAL|REGION}]] [[+|-]…]
Examples:
JOB_ALIAS admin=root-root@lab01-root@lab02-root@lab03
JOB_ALIAS acct=*.acct-nobody.acct-
www.acct+*.fin+*.sys+*.*@acct:REGION+acctjob,*.*

Concurrency Alias:

CONCURRENCY_ALIAS name=[NOT][MIN|MAX] *nnn* [FOR [NOT] (*job_expr*|*job_alias*)] [NOT](*condition*)) [AND|OR …]

Where

condition::={*subcondition*|(*subcondition*) {AND|OR} …}

subcondition::=*term* {{=|<|>|<>|<=|>=|matches|contains|begins|ends} *term*| between *term*,*term*|in *term*[|[…]]}

term::={*jobinfo_expr*|JOBNAME|*jobq_expr*|*jobnum_expr*|*login_expr*| *user_expr*|*var_expr*|*nnn*|"*quoted string*"|`*cmdimage*`}

jobinfo_expr::=JOBINFO [*nnn*|*mnemonic*|*jcsvar*]

jobq_expr::= JOBQINFO [*nnn*|*mnemonic*|*jcsvar*]

jobnum_expr::= JOBNUMINFO [*nnn*|*mnemonic*|*jcsvar*]

login_expr:: = LOGIN [jobname,]user[.group]

user_expr::= USER user[.group][@*host* [:{HOST|REGION|VIRTUAL}]]

var_expr::= *variablename*[%*scope*]

For the format of scope expressions, refer to the scope command.

Scheduling Alias:

SCHEDULING_ALIAS name=[not] schedule_modifier [and|or …]

Calendar Alias:

CALENDAR_ALIAS name=*condition*

condition::={*subcondition*|(*subcondition*) {AND|OR} …}

subcondition::=*term* {{=|<|>|<>|<=|>=|matches|contains|begins|ends} *term*} |between *term*,*term*|in *term*[|[…]]}

term::={*date_spec*|"*quoted string*"|`*cmdimage*`|interval *unit*[.*calendar*] *date_calc*,*date_calc*}

date_spec::= date_calc[:format]

date_calc::= {*date*|*date_increment*}[{+|-}…]

date_increment::= *count*[.*duration*[.*calendar*]]
default duration is days, and default calendar is master.

date::=date_string[.*calendar*]
The format of date_string is dependent upon the NLS calendar format specification for the calendar invoked (default is mm/dd/yyyy[@hh:mi[:ss]]])

Directory Alias:

DIRECTORY_ALIAS *name=directory* […]

File Alias:

FILE_ALIAS *name=file* […]

Source Alias:

SOURCE_ALIAS name={*host_reference*|*ip_reference*}[…]

Where

host_reference::=[HOST:|VIRTUAL:|REGION:]*targetname*

ip_reference::={*ip_addr*[/{*bits*|*netmask*}|-*ip_addr*]
Examples:

192.168.0.0/24
192.168.1.128/255.255.255.128
192.168.1.50-192.168.1.100
192.168.1.101

Target Alias:

TARGET_ALIAS name={*host_reference*|*ip_reference*|*hw_addr*}[…]

Where

host_reference::=[HOST:|VIRTUAL:|REGION:]*targetname*

hw_addr::=MAC:*mac_address*

ip_reference::={*ip_addr*[/{*bits*|*netmask*}|-*ip_addr*]
Examples:
192.168.0.0/24
192.168.1.128/255.255.255.128
192.168.1.50-192.168.1.100
192.168.1.101

Login Alias:

LOGIN_ALIAS *name*=[{*jobname*|*},]{*user*|*}[.{*grp*|*}][{+|-}…]
Examples:
LOGIN_ALIAS acctjobs=acct.*+fin.*+acctjob,*.*
LOGIN_ALIAS systems=root.*+*.sys+*.bin+*.dba+oracle.*

Logical Alias:

LOGICAL_ALIAS *name*=[NOT] (*condition*) [{AND|OR} …]

Selectors:

Introduction:
INTRODUCTION *intro_action*[:*selection*]
Conclusion:
CONCLUSION *concl_action*[:*selection*]

Where

intro_action::={LOCK| LAUNCH [-edit "*sed_cmd*"[…]]| DENY "*errmsg*"[,*errnum*]| HOLD "*holdmsg*"| *sectionname*}

concl_action::={{ABORT|FAIL|ERR|WARN|OK} ["*errmsg*"[,*errnum*]] |*sectionname*}

launch_params::=[-trace] [job_modifier] […] [schedule_modifier] […]

selection::= [NOT] (subselector) [{AND|OR} …]

subselector::={USER {*user_alias*|*user_spec*}| JOB {job_alias|job_spec}|CONCURRENCY {concurrency_alias|concurrency_spec}| SCHEDULE {schedule_alias|schedule_spec}| CALENDAR {calendar_alias|calendar_spec}| DIRECTORY {directory_alias|directory_spec}| FILE {file_alias|file_spec}| SOURCE {source_alias|source_spec}| TARGET {target_alias|target_spec}| LOGIN {login_alias|login_spec}|LOGICAL {logical_alias|logical_spec}}

Introduction selectors are evaluated before the job is introduced. Conclusion selectors are evaluated after the job finishes. If a selection is true (or if no selection is provided), then the corresponding action is performed.

The special actions for the introduction selector are: LOCK, LAUNCH, DROP, EDIT, HOLD and DENY. LOCK indicates that the job is to be locked in the RUNBOOK state until this selection becomes false. DROP specifies launch parameters that should be removed from the **job –launch** command line before the job is submitted. EDIT specifies a sed command or commands that should be used to operate on the **job –launch** command line before the command is launched. LAUNCH specifies that the job is to be launched, with any specified parameters added to the **job –launch** command. HOLD indicates that a manual hold is placed on the job that must be released by system management before the job can execute. DENY specifies that the job is to be rejected for launch, and will print the error message specified to stderr.

The special actions for the conclusion selector are: ABORT, FAIL, ERR, WARN, OK. These correspond to the state that will be assigned to the job for its completion.

Launch Rules:

SECTION sectionname [-shell shellname]
BEGIN
.
.
.
END

The Launch Rules are simply shell scripts that are executed after any selector that references it as a target is set to true. A launch rule will be executed a maximum of one time for any given job.

JCS - jcsh

Notes:

Include base preprocessor into jcsh.
[expr] defines expression to be evaluated. Can be changed through environment variables EXPRBEGIN, EXPREND and EXPRESCAPE.
Maybe add custom modules for expression evaluators (similar to Apache modules).
.jcrc file for startup options.

Commands

:autocont {on|off}

Sets implied continue for every command in the job

:alias cmd,cmdname [$*nnn*|*text*[...]]

:cierror {*nnn*|ERROR|WARN|FAIL|OK}

Sets a job's execution state.

:continue

Continue job even if following job command fails.

:eoj [-nowait] [*nnn*|ERROR|WARN|FAIL|OK|*cmd*]

Specifies End of Jobstream. –nowait kills any executing background jobs.

:goto *label*

does an unconditional transfer to a label.

:jobcard *jobname*[,*user*[,*group*]] [job_modifier[…]

Defines the job card for a job. If provided, must be the first line in a job file.

:label *labelname*

Provides a label location from which a job can be restarted or to which control can be transferred.

:try

.
.
.

[:recover [condition]]

.
.
.

:endtry

Sets up a try/recover block. First, the try block is executed. At the end of the try block (or when it errors out), if the cierror >= ERROR, the first true recover block is executed.

:**when** *condition*
do

.
.
.

done

Base Preprocessor Expressions

chr nnn – Returns ASCII equivalent character.

date [+|-]nnn[.interval[.calendar][{+|-}...][;format] – date calculation displayed in a given format. Default format controlled by environment variable NLSDATEFMT.

do *cmd* – executes *cmd* while preprocessing input

for *parmname*%*scope* **in list**
do
.
.
.
done - for loop inclusion

if *cmd*
.
.
.
else
.
.
.
fi - sets up conditional include blocks

include filename[,recurse] – includes contents of filename into stream.

my {user|usernum|group|groupnum|dir|shell|pty|source} – returns info about the user who launched the job.

parmname[%*scope*] – returns value of a JCS variable.

while *cmd*
do
.
.
.
done - sets up include loops

Schedule Modifiers

schedule_modifier for the above commands is of the form:

[-after *jobname*[,...]]
[-ask "*question_text*"][...]]
[-at *timespec*[.*calendar*]]
[-between *timespec*[.*calendar*], *timespec*[.*calendar*]]
[-deadline *interval*]
[-every *interval*
[-from *interval*]
[-to *interval*]
[-restart *interval*]]
[-except *condition*]
[-file [!][*host*:]*filename*[.*state*]][...]
[-hold "*hold text*"]
[-if *condition*]
[-in *interval*]
[-job *jobname*[.*jobstate*]]
[-on *calendarelement*[.*calendar*][=*value*][,…]]
[-parm *pname*[%*scope*]{=|>|<|<>|>=|<=}{"*value*"|*nnn*|*pname*[%*scope*}[
...]
[-resource *resourcespec*[,...]]
[-sameday {on|off}]
[-unless *condition*]
[-until *condition*]
[-when *condition*]
[-whenever *condition*]
[-with *jobname*[:*nnn*][,…]]
[-without *jobname*[:*nnn*][,…]]

Where

state::={[{user|group|world}.]{ readable|writable|executable}|
exist|setuid|setgid|created {+|-}nnn|modified {+|-}nnn|accessed {+-}nnn}

resourcespec::=*resource*[:{*num*|*jcsvar*|all}]

resource::=*resourcename*[%*scope*]

and interval is of the form:
{*date*|[+/-]*count*[.*duration*[.*calendar*]]}[{+|-}...]
default duration is days, and default calendar is master.

date::=date_spec[.*calendar*]

The format of date_spec is dependent upon the NLS calendar format specification for the calendar invoked (default is mm/dd/yyyy[@hh:mi[:ss]]])

For the format of scope in the above, refer to the scope command.

-after — Specifies one or more jobs that must finish before this one can be introduced. b)

-ask — Specifies a pending reply that must be answered before the job can be released. b)

-at — Specifies a release time for a job. b)

-between — Specifies a calendar window during which a job may be introduced. b)

-deadline — Specifies a deadline for introducing a job. b)

-every — Repeats a job at regular interval. If –from is specified, the –from interval specifies the initial release interval for the job. If –to is specified, the –to interval specifies the final release interval for the job. If –restart is specified, the cycle is re-initialized after the –restart interval passes.

-except — Overrides release of a job if a condition is true.

-file — Specifies that a job is to be released when a given file reaches the specified state. b)

-hold — Places a manual hold on a job that must be cleared by system management before the job will be permitted to execute. The text specified will be shown as the reason the job is being held up. b)

-if — Will release the job for execution only if the condition is true when the job is ready to be released. If the condition specified is not true, the job is pre-empted and flushed. b)

-in — Releases a job after a given interval has passed. b)

-job — Specifies that a job is to be released when another job reaches a given state. If the job never reaches this state, the job to be released will be flushed. b)

-on — Specifies a calendar pattern that must be met in order to release the job. b)

-parm — Specifies that a job is to be released based on the value of one or more JCS parameters. b)

-resource	Specify resources that the job must have available before it can be introduced. The resources are not tied to physical resources, but can be thought of as chips allocated to a job from an available pool. b)
-sameday	Determines whether to release a job now or defer until the next occurrence of a time/date pattern (default is defer). a), b)
-unless	Will release the job for execution only if the condition is false when the job is ready to be released. If the condition specified is true, the job is pre-empted and flushed.
-until	Ends a cycle for a job when a condition is true. b)
-when	Specifies a logical condition that must be met before a job can be released. b)
-whenever	Releases a job every time a condition is met. b)
-with	Specifies jobs that must be running when the job is introduced. If a count of nnn is specified, at least nnn copies of the job must be running. b)
-without	Specifies jobs that must not be running when the job is introduced. If a count of nnn is specified, no more than nnn-1 copies of the job can be running. b)

Job Modifiers

job_modifier for the above commands is of the form:

[-cpu *cpulimit*]
[-del]
[-dir *dirname*]
[-err *cmd*]
[-fail *cmd*]
[-finish *cmd*]
[-autoflush|-noflush]
[-host *host*[(*condition*)][,…]]
[-hostrule *cmd*]
[-init *cmd*]
[-inpri *nnn*]
[-jobnum *jobfamily*]
[-jobq *queuename*]
[-jobqrule *cmd*]
[-login [*jobname*,]*user*[.*group*]]
[-maxwait *mins*]
[-minload *host*[:*weight*],*host*[:*weight*][,…]]
[-need [*host*:]*filename*[,{keep|scrap}}
[-nice *nicevalue*]
[-preprocessor {*cmd*|/OFF/}]
[-{pri|batchq} {*nnn*}]
{*batchq*}
[-setjcs *varname*=*value*
[-shell *progname*]
[-stderr *cmd*]
[-stdout *cmd*]
[-time *timelimit*]
[-type {critical}]
{semi}
{normal}
{custom *rule*}
[-uses [hostname:]file[,[r][w][x]]
[-validate {stdlist|stderr} *cmd*]
[-virtual *vmname* [-token *name* [-expires *expiration*]]]
[-warn *cmd*]

-cpu specifies cpu time limit in seconds for job. Sets job state to ERROR and aborts when time expires.

-del Delete stderr/stdout after successful completion. b)

-dir	Sets the starting directory for the job. Default is user's login directory. b)
-err	Execute command if job ends in ERROR state.
-fail	Execute command if job ends in FAIL state.
-finish	Execute when job ends (prior to checking final state).
-host	Submits the job on the first available host whose associated condition (if specified) is true. If no conditions are specified on any hosts, it is submitted on the host with the lowest load over the last 5 minutes. a), b)
-hostrule	Submits the job to the host returned as output from *cmd*. a), b)
-init	executes the command upon successful initiation of the job. b)
-inpri	Specifies the job's priority in the job queue (higher number is higher priority, must be above fence for queue to be introduced). b)
-jobnum	Specifies the smart job number sequence that will be used to generate this job's job number. a), b)
-jobq	The name of the job queue that will hold this job. (Auto suspend if switching an active job and destination queue is full).
-jobqrule	The name of the job queue that will hold this job. The name of the job queue will be returned by the output of *cmd*.
-login	The name of the job, user and group that the job will run under. If the user is neither specifically allowed nor forbidden from running the job under that login, the user must provide the password for the login.
-maxwait	The maximum number of minutes a job can wait before being canceled.
-minload	submits the job on the available host whose load for the last 5 minutes was lowest. If a weight factor is added, the load on that host is divided by that weight factor. a), b)
-nice	Sets a nice value for the job.
-preprocessor	A command used to process the contents of the job file before it is actually launched. Using /OFF/ as the preprocessor turns off all preprocessing a), b)
-pri	Sets the job's execution priority. Can be numeric, or the name of a scheduling queue.

-shell Sets the startup shell for the job. Default is default shell for the job's login user. b)

-stderr Pipes stderr to command. Must be terminated by an escaped semicolon b)

-stdout Pipes stdout to command. Must be terminated by an escaped semicolon. b)

-time specifies wall time limit in seconds for job. Sets job state to ERROR and aborts when time expires.

-type tells whether job is critical, semi-critical or non-critical. If specified as type custom, then rule will be executed upon abort or EOJ, and the return value of rule will be used as the job's type. b)

-uses Specifies that the job must have local access to the file specified. If the file is required for writing, it is transferred back after the job finishes.

-validate Specifies a command to be used to validate the job for successful completion.

-virtual Specifies the virtual host that will be used to load balance the request. If a token is specified, that token is used to specify a job persistence target for future jobs that must exit on the same target machine as this job. For example, suppose job A is assigned to virtual host VIRTUAL1 made up of hosts X, Y and Z. Job A has a token of MYTOKEN attached to it. If job A is assigned to host Y for execution, and at some point later job B is assigned to virtual host VIRTUAL1, if it also has the token MYTOKEN attached to it, it will also be assigned to host Y for execution, as will any job with the token MYTOKEN attached to it, until the token MYTOKEN expires. Tokens only expire when a job with a token specifies an expiration for its token, and the expiration becomes true. Each succeeding job that has an expiration for a token will overwrite the current expiration for the token with its own expiration.

-warn Execute command if job ends in WARN state.

Notes:

a) Cannot be changed once job is launched
b) Cannot be changed once job is introduced
c) May be repeated.

jobinfo({*jsid*} , {*nnn* })
{*jobname*}, {*varname* }
{*jobnumber*}, {*mnemonic*}
{*filename*}

Num.	Mnemonic	Return Type	Description
0	EXISTS	Boolean	Returns TRUE if the job exists on the system, returns false otherwise.
1	CIPID	Integer	PID for job master process
100	JSID	Integer	Job Number
101	JOBNUM	String	Job Number
102	LOGIN	String	Returns the logon name in the form: [jobname,]user.group
103	USERID	Integer	User ID number for job.
104	USER	String	User Name
105	GROUPID	Integer	Group ID number for job
106	GROUP	String	Returns the Job/Session's current logon group.
110	JOBNAME	String	Returns Job Name
115	SHELL	String	Job's shell on startup.
116	DIR	String	Starting directory
117	JOBFILE	String	Name of the file used as source for this job
118	DINTRO	String	Returns the introduction date for the Job in the form: "DD-MTH-YYYY HH:MI"
119	INTRO	String	Returns the introduction date for the Job in the form: "YEARMMDDHHMI"
120	IINTRO	Integer	Returns the introduction date for the Job as offset from 1/1/1970
121	STREAMEDBY	String	User who launched this job
122	STDIN	String	Returns filename of $STDIN for the Job
123	STDLIST	String	Returns filename of $STDLIST for Job
124	STDERR	String	Returns filename of stderr for Job
125	JOBSTEP	String	Returns the current job step for the job
126	JOBS	Integer	Returns the number of active jobs.
127	INPRI	Integer	Job input priority
129	DLAUNCHED	String	Returns the launch date for the Job in the form: "DD-MTH-YYYY HH:MI"
130	LAUNCHED	String	Returns the launch date for the Job in the form: "YEARMMDDHHMI"
131	ILAUNCHED	Integer	Returns the introduction date for the Job as offset from 1/1/1970
134	DEFERRED	Boolean	Returns TRUE if job is deferred, false otherwise
136	HOLD	Integer	Returns TRUE if the original job is on hold, false otherwise
138	WALLTIME	Integer	The Wall time limit (in minutes) for the Job
139	CPULIMIT	Integer	The CPU limit established for the Job
140	STATE	String	The general state category for the job (EXEC, SUSP, SCHED or EOJ)
141	SUBSTATE	String	The current Job state (EXEC, SUSP, WAIT, INTRO,

			HOLD, OK, CANCEL, ABORT, RESOURCE, RUNBOOK, LOCKED, ERR, WARN, FAIL, VERIFY, EVAL or SCHED)
145	DELETESTDLIST	Boolean	Returns TRUE if STDLIST=DELETE, false otherwise
148	ERRCMD	String	Command to execute if job ends in ERR Status.
149	FAILCMD	String	Command to execute if job ends in FAIL Status.
150	FINISHCMD	String	Command to execute if job validates successfully.
151	INITCMD	String	Command to execute when job is introduced, but before it begins executing.
152	VALIDATECMD	String	Command to execute if job ends in OK Status.
153	WARNCMD	String	Command to execute if job ends in WARN status.
154	AUTOFLUSH	Boolean	Returns TRUE if job is set to automatically flush from queue upon abend, FALSE otherwise.
155	MAXWAIT	Integer	Maximum number of minutes job can remain in WAIT state before being flushed.
156	NICE	Integer	Nice value for job.
157	PREPROCESSOR	String	Command used to pre-process job template file in order to generate final job file.
158	SETJCS	String	List of JCS variables set prior to launching job.
159	CRITICAL	String	Returns “CRITICAL” for critical job, “SEMI” for semi-critical job, “NORMAL” for normal importance, “NONCRITICAL” for non-critical job, and the rule to be executed for a variable criticality job.
200	JOBQ	Integer	Jobqueue to which job belongs.
201	JOBLIMIT	Integer	System job limit
202	JOBQLIMIT	Integer	Job limit for this job’s jobqueue.
203	JOBFENCE	Integer	Current system jobfence
204	JOBQFENCE	Integer	Jobfence for this job’s jobqueue.
399	RESOURCES	String	List of resources for this job.
500	JOBFAMILY	String	Returns the name of the job number family to which this job belongs.
899	DEPENDENCIES	String	Returns list of pending dependencies for job.
901	NUMABORT	Integer	Number of jobs fitting pattern specified that are in ABORT state
902	NUMACTIVE	Integer	Number of jobs fitting pattern specified that are logged on
903	NUMCANCEL	Integer	Number of jobs fitting pattern specified that are in CANCEL state
904	NUMERROR	Integer	Number of jobs fitting pattern specified that are in ERROR state
905	NUMEXEC	Integer	Number of jobs fitting pattern specified that are in EXEC state
906	NUMFAIL	Integer	Number of jobs fitting pattern specified that are in FAIL state
907	NUMFINISH	Integer	Number of jobs fitting pattern specified that are in FINISH state
908	NUMHOLD	Integer	Number of jobs fitting pattern specified that are in HOLD state
909	NUMINTRO	Integer	Number of jobs fitting pattern specified that are in INTRO state

910	NUMRESOURCE	Integer	Number of jobs fitting pattern specified that are in RESOURCE wait state
911	NUMSCHED	Integer	Number of jobs fitting pattern specified that are in SCHED state
912	NUMSUSP	Integer	Number of jobs fitting pattern specified that are in SUSP state
913	NUMVERIFY	Integer	Number of jobs being verified for successful completion.
914	NUMWAIT	Integer	Number of jobs fitting pattern that are in WAIT state
915	NUMWARN	Integer	Number of jobs fitting pattern specified that are in WARN state
999	TOTAL	Integer	Number of jobs fitting pattern

jobnum(spec) – returns job numbers of all jobs fitting spec.

jobqinfo(queuename,parmnum) – returns info about a jobqueue.

Num.	Mnemonic	Return Type	Description
0	EXISTS	Boolean	Returns TRUE if the job queue exists on the system, returns false otherwise.
200	JOBQ	Integer	Jobqueue name.
201	JOBLIMIT	Integer	System job limit
202	JOBQLIMIT	Integer	Job limit for this jobqueue.
203	JOBFENCE	Integer	Current jobfence
204	JOBQFENCE	Integer	Jobfence for this jobqueue.
205	JOBS	Integer	Returns the number of active jobs.
206	QSTATE	String	The current Jobqueue state (OPEN, CLOSED, SUSPEND, LOAD or ERROR).
207	RULE	String	Rule for job introduction order
212	MAXLOAD	Double	Maximum system load before queue shuts down.
213	PARENT	String	Parent queue for this jobqueue.
214	MINSLICE	Integer	Minimum CPU slice (0.1%) process must get
215	MAXSLICE	Integer	Maximum CPU slice (0.1%) process may get
216	NICE	Integer	Nice value for each job in the queue
217	INCR	Integer	Priority change for jobs outside slice limits
901	NUMABORT	Integer	Number of jobs in the jobqueue that are in ABORT state
902	NUMACTIVE	Integer	Number of jobs in the jobqueue that are logged on
903	NUMCANCEL	Integer	Number of jobs in the jobqueue that are in CANCEL state
904	NUMERROR	Integer	Number of jobs in the jobqueue that are in ERROR state
905	NUMEXEC	Integer	Number of jobs in the jobqueue that are in EXEC state
906	NUMFAIL	Integer	Number of jobs in the jobqueue that are in FAIL state
907	NUMFINISH	Integer	Number of jobs in the jobqueue that are in FINISH state
908	NUMHOLD	Integer	Number of jobs in the jobqueue that are in HOLD state
909	NUMINTRO	Integer	Number of jobs in the jobqueue that are in INTRO state
910	NUMRESOURCE	Integer	Number of jobs in the jobqueue that are in RESOURCE wait state
911	NUMSCHED	Integer	Number of jobs in the jobqueue that are in SCHED state
912	NUMSUSP	Integer	Number of jobs in the jobqueue that are in SUSP state
913	NUMVERIFY	Integer	Number of jobs in the jobqueue being verified for successful completion.
914	NUMWAIT	Integer	Number of jobs in the jobqueue that are in WAIT state
915	NUMWARN	Integer	Number of jobs in the jobqueue that are in WARN state
999	TOTAL	Integer	Total Jobs assigned to the jobqueue

```
replyinfo (     {JOBNUM,    jobnum},            {nnn }          )
                {PID,       nnn },              {jcwname }
                {LOGON,     [sname,]user.grp},  {mnemonic }
                {PROCESS,   filename},
                {STREAM,    streamfile},
                {REPLYID,   replynum},
                {HOST,      hostname},
                {VIRTUAL,   vmname},
```

The information returned by replyinfo (by request number/mnemonic) is as follows:

Num.	Mnemonic	Return Type	Description
0	EXISTS	Boolean	Reply exists for user.
1	PID	String	PID of process requesting REPLY.
100	JSID	Integer	Job Number for job requesting REPLY.
101	JOBNUM	String	Job number of job requesting REPLY.
102	LOGIN	String	Job card of job requesting REPLY in form [sess,]user.grp
103	USERID	Integer	User ID number requesting reply.
104	USER	String	User name requesting reply.
105	GROUPID	Integer	Group ID number requesting reply.
106	GROUP	String	Group name requesting reply.
110	JOBNAME	String	Job card session name requesting reply.
400	REPLYID	Integer	ID of pending reply.
404	TEXT	String	Text of pending REPLY.
405	TIME	String	24 hour time REPLY was posted.
406	TIMEOUT	Integer	Time in minutes after REPLY was posted that it will timeout.
407	ITIME	Integer	Time REPLY was posted in seconds offset from 1/1/1970.
408	DURATION	Integer	Duration in minutes reply has been pending.
410	PROCESS	String	Name of process requesting REPLY.
411	MINLEN	Integer	Minimum length in characters for response.
412	MAXLEN	Integer	Maximum length in characters for response.
413	EDIT	String	Regex that will be used to check validity of response.
414	DEFAULT	String	Response that will be issued if REPLY times out.
415	HOST	String	Host name where reply will show.
416	ECHO	Boolean	Determines whether ASK command will echo to stderr.
700	VIRTUAL	String	Virtual host whose targets will show the reply.
999	TOTAL	Integer	Total number of outstanding REPLY's fitting requested pattern

resinfo(resource,parmnum) – returns info about a resource.

Num.	Mnemonic	Return Type	Description
0	EXISTS	Boolean	Resource exists
17	RULE	String	Allocation rule for Resource
300	RESOURCE	String	Resource name
301	LEVEL	String	LOCAL, GLOBAL or SHARED
302	SUBCLASS	Array	Name of job number family, jobqueue, user, etc. owning resource.
303	DESC	String	Description of the Resource
304	LIMIT	Integer	Total number of the Resource
305	MIN	Integer	Minimum priority of a process requesting a Resource
306	MAX	Integer	Maximum priority of a process requesting a Resource
307	AVAIL	Integer	Number currently free for allocation
309	SORT	String	Sort order of processes requesting Resource
310	USED	Integer	Number currently in use
311	USERS	Array	List of PID, JSID and COUNT using a resource
312	WAITING	Array	List of PID, JSID and COUNT waiting for a resource

resuser(resource,pid,parmnum) – returns info about a user of a resource.

Num.	Mnemonic	Return Type	Description
0	EXISTS	Boolean	Resource exists
1	PID	Integer	PID for job master process
2	JSID	Integer	Job Number
3	JOBNUM	String	Job Number
4	LOGIN	String	Returns the logon name in the form: [jobname,]user.group
5	USERID	Integer	User ID number for job.
6	USER	String	User Name
7	GROUPID	Integer	Group ID number for job
8	GROUP	String	Returns the Job/Session's current logon group.
9	JOBNAME	String	Returns Job Name
17	RULE	String	Allocation rule for Resource
300	RESOURCE	String	Resource name
301	LEVEL	String	LOCAL, GLOBAL or SHARED
302	SHARES	Array	Hosts sharing the resource
303	DESC	String	Description of the Resource
304	LIMIT	Integer	Total number of the Resource
305	MIN	Integer	Minimum priority of a process requesting a Resource
306	MAX	Integer	Maximum priority of a process requesting a Resource
307	AVAIL	Integer	Number currently free for allocation
309	SORT	String	Sort order of processes requesting Resource
310	USED	Integer	Number currently in use
311	USERS	Array	List of PID, JSID and COUNT using a resource
312	WAITING	Array	List of PID, JSID and COUNT waiting for a resource

getvar(parmname,scope) – returns value of a JCS variable.

setvar(parmname,scope,value[,indirect]) – sets a value for a JCS variable.

finfo(filename,parmnum) – returns info about a file.

duration(units,from[,to[,calendarname]]) – returns the number of temporal units between two dates.

iduration(units,from[,to[,calendarname]]) – returns the integral number of temporal units between two dates.