Platform™ LSF™ API Reference

Version 7 Update 5
Release date: March 2009
Last modified: March 20, 2009
Comments to: doc@platform.com
Support: support@platform.com

Copyright

We’d like to hear from you

You can help us make this document better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this document, please address your comments to doc@platform.com.

Your comments should pertain only to Platform documentation. For product support, contact support@platform.com.

Document redistribution and translation

Internal redistribution

Trademarks

Third-party license agreements

Third-party copyright notices

Contents

lslib

Application Programming Interface (API) library routines for LSF services

LSLIB routines allow application programs to contact the Load Information Manager (LIM) and Remote Execution Server (RES) daemons in order to obtain LSF services. These services include obtaining static system configuration information and dynamic load information for the hosts in distributed clusters, obtaining task placement advice from LIM, executing tasks (UNIX processes) on remote hosts with a high degree of transparency using RES, remote file operations across hosts that do not share file systems, performing remote terminal I/O and signal operations, and other related functions. You can build distributed applications on top of LSLIB to effectively exploit the resources available on the network, improving application performance and resource accessibility.

The LSLIB APIs are grouped by function on the following man pages:

ls_readconfenv() Library API for reading the LSF configuration environment. The `ls_readconfenv()` API is documented on this page.

ls_info() Library routines for obtaining load sharing cluster configuration information. The APIs documented on this page include:

- `ls_info()`
- `ls_getclustername()`
- `ls_getmastername()`
- `ls_getmodelfactor()`

ls_hostinfo() Library routines for obtaining host configuration information. The APIs documented on this page include:

- `ls_gethosttype()`
- `ls_gethostmodel()`
- `ls_gethostfactor()`
- `ls_gethostinfo()`

ls_load() Library routines for obtaining host load information. An application can use this information to make task placement decisions instead of using LIM’s. The APIs documented on this page include:

- `ls_loadinfo()`
- `ls_load()`
- `ls_loadofhosts()`
- `ls_sharedresourceinfo()`

ls_policy() Library routines implementing LIM’s task placement policy. These routines include calls to obtain task placement information, and calls to adjust host load measures. The APIs documented on this page include:

- `ls_placereq()`
- `ls_placeofhosts()`
- `ls_loadadj()`

ls_task() Library routines for displaying and manipulating the local and remote task lists. These lists specify the eligibility of various types of tasks for remote execution, and their resource requirement characteristics. The APIs documented on this page include:

- `ls_eligible()`
- `ls_listrtask()`
- `ls_listltask()`
- `ls_insertrtask()`
- `ls_insertltask()`
- `ls_deletertask()`
- `ls_deleteltask()`

REMOTE EXECUTION

Library routines related to remote execution, including initiation, connection and remote environment manipulation. There are a number of manual pages that describe remote execution calls:

ls_initrex() Library routine for initializing an LSF application for remote execution. The APIs documented on this page is:

`ls_initrex()`

ls_connect() Library routines for establishing and querying remote connections. The APIs documented on this page include:

- `ls_connect()`
- `ls_isconnected()`
- `ls_findmyconnections()`

ls_rexecv() Library routines for executing remote tasks. The APIs documented on this page include:

- `ls_rexecv()`
- `ls_rexecve()`
- `ls_rtask()`
- `ls_rtaske()`

ls_stdinmode() Library routines for querying and manipulating stdin for remote tasks. The APIs documented on this page include:

- `ls_stdinmode()`
- `ls_getstdin()`
- `ls_setstdin()`

ls_rwait() Wait for a remote or local task, then return its status. The routines documented on this page include:

- `ls_rwait()`
- `ls_rwaittid()`

ls_chdir() Set the remote current working directory.

ls_rsetenv()	Set up a remote task environment.
ls_rkill()	Send a signal to a remote task.
ls_donerex()	Clean up before closing a remote connection.
ls_fdbusy()	Check if a file descriptor is claimed by LSF.
ls_stoprex()	Stop the network I/O server and restore local tty settings.
ls_conntaskport()	Get a socket connected to a remote task port.
ls_rfs()	Library routines for operations on files on remote hosts. The APIs documented on this page include: ◆ `ls_ropen()` ◆ `ls_rread()` ◆ `ls_rwrite()` ◆ `ls_rlseek()` ◆ `ls_rclose()` ◆ `ls_rstat()` ◆ `ls_rfstat()` ◆ `ls_rgetmnthost()` ◆ `ls_rfcontrol()`
ls_perror()	Library routine for load sharing error messages. The routines documented on this page are: ◆ `ls_perror()` ◆ `ls_sysmsg()` ◆ `ls_errlog()`
ls_admin()	Library routines for administering and controlling the LSF system. The APIs documented on this page include: ◆ `ls_limcontrol()` ◆ `ls_lockhost()` ◆ `ls_unlockhost()`
ls_rescontrol()	Remote Execution Server control.
ls_readrexlog()	Read a record from the remote executed task log file.
ls_getmnthost()	Get the name of the host that exports a file system.
ls_getmyhostname()	Deprecated. Get the name used throughout LSF to represent the local host.
ls_getmyhostname2()	Replaces ls_getmyhostname(). Get the name used throughout LSF to represent the local host.

NOTES

All LSLIB routines require that the LSF header file `<lsf/lsf.h>` be included.

Many LSLIB APIs return a pointer to an array or structure. These data structures are in static storage or on the heap. The next time the routine is called, the storage is overwritten or freed.

Any program using LSLIB routines that change the state of the LSF system (that is, those routines documented in `ls_admin()` and `ls_rex()`) must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

Any program using LSLIB routines documented in the `ls_rex()` or `ls_rfs()` man pages must call `ls_initrex()` before calling any of the other routines.

On systems which have both System V and BSD programming interfaces, LSLIB typically requires the BSD programming interface. On System V-based versions of UNIX, for example SGI IRIX, it is normally necessary to link applications using LSLIB with the BSD compatibility library.

On AFS systems, the following needs to be added to the end of your linkage specifications when linking with LSLIB (assuming your AFS library path is `/usr/afsws`):

For HP-UX and Solaris,

```
-lc -L/usr/afsws/lib -L/usr/afsws/lib/afs -lsys -lrx -llwp
/usr/afsws/lib/afs/util.a
```

For other platforms,

```
-lc -L/usr/afsws/lib -L/usr/afsws/lib/afs -lsys -lrx -llwp
```

FILES

`${LSF_ENVDIR-/etc}/lsf.conf`

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.cluster_name`

SEE ALSO

`ls_readconfenv()`, `ls_info()`, `ls_hostinfo()`, `ls_load()`, `ls_policy()`, `ls_task()`, `ls_rex()`, `ls_initrex()`, `ls_connect()`, `ls_rexecv()`, `ls_stdinmode()`, `ls_rwait()`, `ls_chdir()`, `ls_rsetenv()`, `ls_rkill()`, `ls_donerex()`, `ls_fdbusy()`, `ls_stoprex()`, `ls_rfs()`, `ls_perror()`, `ls_admin()`, `ls_rescontrol()`, `lim`, `res`, `nios`

lsblib

Application Programming Interface (API) library functions for batch jobs

LSBLIB functions allow application programs to get information about the hosts, queues, users, jobs and configuration of the batch system. Application programs can also submit jobs and control hosts, queues and jobs. Finally, application programs can read batch log files and write batch error messages.

LSBLIB contains the following functions:

lsb_chkpntjob()

Checkpoint a job

lsb_closejobinfo()

Close the job information connection with mbatchd

lsb_geteventrec()

Get an event record from a log file

lsb_hostcontrol()

Enable or disable a host, restart or shutdown a slave batch daemon

lsb_hostgrpinfo()

Get membership of batch host groups

lsb_hostinfo()

Get information about job server hosts

lsb_sharedresourceinfo()

Get information about shared resource used for scheduling

lsb_hostpartinfo()

Get information about host partitions

lsb_init()

Initialize LSBLIB and get the configuration environment

lsb_mig()

Migrate a job from one host to another

lsb_movejob()

Move a job in a queue

lsb_openjobinfo()

Open a job information connection with the mbatchd

lsb_parameterinfo()

Get information about the batch cluster

lsb_peekjob()

Retrieve the name of a job’s output file

lsb_pendreason()

Explain why a job is pending

lsb_perror()

Print a batch error message on stderr

lsb_queuecontrol()

Dynamically change the status of a batch job queue

lsb_queueinfo()

Get information about batch queues

lsb_readjobinfo()

Get the next job information record from the connection with mbatchd

lsb_reconfig()

Reconfigure the batch cluster

lsb_signaljob()

Send a signal to a job

lsb_submit()

Submit a job to the batch system

lsb_suspreason()

Explain why a job was suspended

lsb_switchjob()

Switch a job to another queue

lsb_sysmsg()

Return an batch error message

lsb_usergrpinfo()

Get membership of batch user groups

lsb_userinfo()

Get information about users and user groups

NOTES

All LSBLIB APIs require that the batch header file `<lsf/lsbatch.h>` be included.

Many LSBLIB APIs return a pointer to an array or structure. These data structures are in static storage or on the heap. The next time the API is called, the storage is overwritten or freed.

Any program using LSBLIB APIs that change the state of the batch system (that is, except for APIs that just get information about the system) must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

On systems which have both System V and BSD programming interfaces, LSBLIB typically requires the BSD programming interface. On System V-based versions of UNIX, for example SGI IRIX, it is normally necessary to link applications using LSBLIB with the BSD compatibility library.

On AFS systems, the following needs to be added to the end of your linkage specifications when linking with LSBLIB (assuming your AFS library path is `/usr/afsws`):

For HP-UX and Solaris,

```
-lc -L/usr/afsws/lib -L/usr/afsws/lib/afs -lsys -lrx -llwp
/usr/afsws/lib/afs/util.a
```

For other platforms,

```
-lc -L/usr/afsws/lib -L/usr/afsws/lib/afs -lsys -lrx -llwp
```

FILES

`${LSF_ENVDIR-/etc}/lsf.conf`

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.`*cluster_name*

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.cluster_name`

`$LSB_CONFDIR/cluster/lsb.hosts`

`$LSB_CONFDIR/cluster/lsb.params`

`$LSB_CONFDIR/cluster/lsb.queues`

`$LSB_CONFDIR/cluster/lsb.users`

SEE ALSO

`lsb_chkpntjob()`, `lsb_closejobinfo()`, `lsb_hostgrpinfo()`, `lsb_usergrpinfo()`, `lsb_hostcontrol()`, `lsb_hostinfo()`, `lsb_sharedresourceinfo()`, `lsb_hostpartinfo()`, `lsb_init()`, `lsb_mig()`, `lsb_movejob()`, `lsb_openjobinfo()`, `lsb_parameterinfo()`, `lsb_peekjob()`, `lsb_pendreason()`, `lsb_perror()`, `lsb_queuecontrol()`, `lsb_queueinfo()`, `lsb_readjobinfo()`, `lsb_reconfig()`, `lsb_signaljob()`, `lsb_submit()`, `lsb_suspreason()`, `lsb_switchjob()`, `lsb_sysmsg()`, `lsb_userinfo()`, `lsb.queues`, `lim`, `res`, `mbatchd`

glb_close_all()

Closes the connections to all License Scheduler daemons.

DESCRIPTION

`glb_close_all()` is a routine that closes the connections to all License Scheduler daemons specified in `lsf.licensescheduler`. This is useful when there are multiple License Scheduler master and slave daemons running for failover purposes.

SYNOPSIS

```
#include <glb/glb.h>
void glb_close_all(void)
```

PARAMETERS

void: There are no parameters.

RETURN VALUES

void: There is no return value.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_init_all()` - Initializes and establishes a connection with all License Scheduler daemons

Files

`LSF_CONFDIR/lsf.licensescheduler`

glb_groupinfolist()

Returns an array of License Scheduler project group information.

DESCRIPTION

`glb_groupinfolist()` returns an array of `glbGroupInfoList` structures, which contains the License Scheduler project hierarchical group information.

SYNOPSIS

```
#include <glb/glb.h>
struct glbGroupInfoList *glb_groupinfolist(int *num)

struct glbGroupInfoList {
    int num;
    char *featurename;
    char *sdlist;
    int totalavail;
    struct glbGroupInfo *groups;
};

struct glbGroupInfo {
    char *name;
    char *path;
    struct groupWeight *w;
    int numChildren;
};
```

PARAMETERS

***num:** The number of desired `glbGroupInfoList` structures returned in the array.

RETURN VALUES

***glbGroupInfoList:** A list of License Scheduler project group information.

ERRORS

On failure, `glberrno` is set to indicate the error.

glb_info()

Returns an array of license features information.

DESCRIPTION

`glb_info()` returns an array of `glbInfo` structures, which contains the license features information as configured in the `Feature` section of `glb.conf`.

SYNOPSIS

```
#include <glb/glb.h>
struct glbInfo *glb_info(struct glbHandle *h2, int *num)

struct glbInfo {
    char *featureName;
    int numdomains;
    struct glbSrvDomain *domains;
};
```

PARAMETERS

***h2:** The License Scheduler daemon containing the desired license features information.

***num:** The number of desired `glbInfo` structures returned in the array.

RETURN VALUES

***glbInfo:** A list of parameters associated with the License Scheduler daemon.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_jobinfo()` - Returns an array of running jobs that are using license tokens and licenses

`glb_param()` - Returns an array of parameters of the specified License Scheduler daemon

`glb_perror()` - Prints LSF License Scheduler error messages

`glb_userinfo()` - Returns an array of users that are using license tokens and licenses

`glb_workloadinfo()` - Returns an array of workload distribution information

glb_init_all()

Initializes and establishes a connection with all License Scheduler daemons.

DESCRIPTION

`glb_init_all()` is a routine that initializes and establishes a connection with all License Scheduler daemons specified in `lsf.licensescheduler`. This is useful when there are multiple License Scheduler master and slave daemons running for failover purposes.

SYNOPSIS

```
#include <glb/glb.h>
```

```
link_t *glb_init_all() {
    return *link_t;
};

typedef struct link_t {
    int num;
    void *ptr;
    struct link_t *next
};
```

RETURN VALUES

***link_t:** A linked list of License Scheduler daemon handles that `glb_init_all()` initialises and connects to.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_close_all()` - Closes the connections to all License Scheduler daemons

Files

`LSF_CONFDIR/lsf.licensescheduler`

glb_jobinfo()

Returns an array of running jobs that are using license tokens and licenses.

DESCRIPTION

`glb_jobinfo()` returns an array of `glbJob` structures, which contains information on each job that is using a license token and a license from the license server. Each job is either an LSF batch job or a taskman-controlled application.

SYNOPSIS

```
#include <glb/glb.h>
struct glbJob *glb_jobinfo(int *njobs)

struct glbJob {
    char *jobid;
    char *client;
    char *user;
    char *host;
    char *cluster;
    int starttime;
    int status;
    int nSpec;
    struct glbJobSpec *jobSpec;
};
```

PARAMETERS

***njobs** The number of jobs that are currently in the system. This parameter is used as a return value.

RETURN VALUES

***glbJob:** An array of running jobs currently using license tokens and licenses from the license server.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_info()` - Returns an array of license features information

`glb_param()` - Returns an array of parameters of the specified License Scheduler daemon

`glb_perror()` - Prints LSF License Scheduler error messages

`glb_userinfo()` - Returns an array of users that are using license tokens and licenses

`glb_workloadinfo()` - Returns an array of workload distribution information

glb_param()

Returns an array of parameters of the specified License Scheduler daemon.

DESCRIPTION

`glb_param()` gets an array of `glbParams` structures which contains all the parameters associated with the specified License Scheduler daemon.

SYNOPSIS

```
#include <glb/glb.h>
struct glbParams *glb_param(struct glbHandle *h2)

struct glbParams {
    int globPort;
    char *host;
    char *logMask;
    char *logEvent;
    char *adminName;
    char *logDir;
    char *workDir;
    int logInterval;
    int lmStatInterval;
    char *licfile;
};
```

PARAMETERS

***h2:** The `glbHandle` structure that specifies the License Scheduler daemon.

RETURN VALUES

***glbParams:** An array of internal parameters associated with the License Scheduler daemon.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_info()` - Returns an array of license features information

`glb_jobinfo()` - Returns an array of running jobs that are using license tokens and licenses

`glb_perror()` - Prints LSF License Scheduler error messages

`glb_userinfo()` - Returns an array of users that are using license tokens and licenses

`glb_workloadinfo()` - Returns an array of workload distribution information

glb_perror()

Prints LSF License Scheduler error messages.

DESCRIPTION

`glb_perror()` is a library routine for printing LSF License Scheduler error messages corresponding to the reported error number.

glb_perror()

SYNOPSIS

```
#include <glb/glb.h>
void glb_perror (const char *msg)
```

PARAMETERS

***msg** The user-defined message. This is printed in front of the error message and is separated from the error message by a colon.

RETURN VALUES

void: There is no return value.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`glb_info()` - Returns an array of license features information

`glb_jobinfo()` - Returns an array of running jobs that are using license tokens and licenses

`glb_param()` - Returns an array of parameters of the specified License Scheduler daemon

`glb_userinfo()` - Returns an array of users that are using license tokens and licenses

`glb_workloadinfo()` - Returns an array of workload distribution information

glb_userinfo()

Returns an array of users that are using license tokens and licenses.

DESCRIPTION

`glb_userinfo()` returns an array of `glbUser` structures, which contains information on each user that is using a license token and a license from the license server. The license server identifies users by the identifier `user_name@host_name`.

SYNOPSIS

```
#include <glb/glb.h>
struct glbUser *glb_userinfo(int *nusers, int *lmstatIntvl)

struct glbUser {
    char *feature;
    char *serviceDomain;
    char *name;
    char *host;
    char *version;
    char *vendor;
    int ndisplays;
    char **display;
    int numlics;
    char *checkoutTime;
    int handle;
    int ntasks;
    char *client;
    int nonglb;
    int nTotalLic;
};
```

PARAMETERS

***nusers**	The number of users that are currently in the system. This parameter is used as a return value.
***lmstatIntvl**	The current `lmstat` interval. This parameter is used as a return value.

RETURN VALUES

***glbUser:**	An array of users currently using license tokens and licenses from the license server.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_info()` - Returns an array of license features information

`glb_jobinfo()` - Returns an array of running jobs that are using license tokens and licenses

SEE ALSO

`glb_param()` - Returns an array of parameters of the specified License Scheduler daemon

`glb_perror()` - Prints LSF License Scheduler error messages

`glb_workloadinfo()` - Returns an array of workload distribution information

glb_workloadinfo()

Returns an array of workload distribution information.

DESCRIPTION

`glb_workloadinfo()` returns an array of `glbWorkload` structures, which contains detailed LSF and non-LSF workload distribution information for a particular feature.

SYNOPSIS

```
#include <glb/glb.h>
```

```
struct glbWorkload *glb_workloadinfo(void)

struct glbWorkload {
    struct glbWorkload *next;
    char *feature;
    char *sDomain;
    struct workloadBasket lsf;
    struct workloadBasket nonlsf;
};
```

PARAMETERS

void:	There are no parameters

RETURN VALUES

***glbWorkload:**	A linked list of workload distribution inforation for a particular feature.

ERRORS

On failure, `glberrno` is set to indicate the error.

SEE ALSO

Related APIs

`glb_info()` - Returns an array of license features information

`glb_jobinfo()` - Returns an array of running jobs that are using license tokens and licenses

`glb_param()` - Returns an array of parameters of the specified License Scheduler daemon

`glb_perror()` - Prints LSF License Scheduler error messages

`glb_userinfo()` - Returns an array of users that are using license tokens and licenses

ls_chdir()

DESCRIPTION

`ls_chdir()` sets the application's working directory on the remote host to the directory specified by *clntdir*. If the application subsequently requests remote execution with the flag `REXF_CLNTDIR`, the Remote Execution Server (RES) uses the application's working directory on the remote host, instead of the application's local current working directory, as the current working directory for the remote execution. The RES keeps a working directory for each application, which is initialized as the user's home directory. The application can call this routine to change its working directory on a particular host. *clntdir* must be the full pathname of a valid directory on the host *host*.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

Sets an application's working directory on a remote host to a specified directory.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_chdir(char *host, char *clntdir)
```

PARAMETERS

***host**	The remote host containing the client directory.
***clntdir**	The full pathname of a valid directory on the host *host*.

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.
integer:-2	A warning if the RES fails to check *clntdir* due to permission denial. This is a temporary mechanism to get around the root uid mapping problem of NFS.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_clusterinfo()

Returns general information about LSF clusters.

DESCRIPTION

This routine returns general configuration information about LSF clusters.

`ls_clusterinfo()` returns an array of clusterInfo data structures, as defined in `<lsf/lsf.h>`. Each entry contains information about one cluster. The information includes cluster name, cluster status, the master host name, LSF primary administrator login name (for backward compatibility), LSF primary administrator user Id (for backward compatibility), total number of server hosts, total number of client hosts, available resource names, host types, host models, total number of LSF administrators, LSF administrator user Ids and LSF administrator login names.

The parameter `resreq` is designed to select eligible clusters that satisfy the given resource requirements from candidate clusters. This parameter is currently ignored. `clusterlist` gives a list of cluster names whose information should be returned, if they satisfy the `resreq`. If `clusterlist` is `NULL`, then all clusters known to LSF satisfying `resreq` will be returned. `listsize` gives the size of the `clusterlist`. If `numhosts` is not `NULL`, then `*numhosts` will be modified to return the number of clusters selected. The parameter `options` is currently ignored.

This routine returns a pointer to dynamically allocated data which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
struct clusterInfo *ls_clusterinfo(char *resreq,
                    int *numclusters, char **clusterlist,
                    int listsize, int options)

struct clusterInfo {
    char clusterName[MAXNAMELEN];
    int status;
    char masterName[MAXHOSTNAMELEN];
    char managerName[MAXHOSTNAMELEN];
    int managerId;
    int numServers;
    int numClients;
    int nRes;
    char **resources;
    int nTypes;
    char **hostTypes;
    int nModels;
    char **hostModels;
    int nAdmins;
    int *adminIds;
    char **admins;
    int   jsLicFlag;
    char  afterHoursWindow[MAXLINELEN];
    char  preferAuthName[MAXLSFNAMELEN];
    char  inUseAuthName[MAXLSFNAMELEN];
};
```

PARAMETERS

***resreq**	Select eligible clusters that satisfy the given resource requirements from candidate clusters. This parameter is currently ignored.
***numclusters**	If `numclusters` is not `NULL`, then `*numclusters` will be modified to return the number of clusters selected.
****clusterlist**	Gives a list of cluster names whose information should be returned, if they satisfy the `resreq`. If `clusterlist` is `NULL`, then all clusters known to LSF satisfying `resreq` will be returned.
listsize	The size of the `clusterlist`.
option	The parameter `options` is currently ignored.

RETURN VALUES

array:	Function was successful.
struct:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_connect()

Sets up an initial connection with the Remote Execution Server (RES) on a specified remote host.

DESCRIPTION

`ls_connect()` sets up an initial connection with the Remote Execution Server (RES) on a specified remote host. You can then use this connection for future remote execution or control interactions. This routine is called automatically if a connection is not set up when a remote execution request is made (see `ls_rexecv()`). The explicit invocation of this routine has performance advantages in certain cases, typically for parallel applications. The routine returns immediately when the connection is established rather than waiting for the completion of the possibly time consuming authentication and status checking process by the RES (see `res()`). The application can set up initial connections with many remote hosts simultaneously, overlapping the authentication processes on all remote hosts. On successful completion, this routine returns a socket descriptor through which the connection has been established. If the caller's effective uid is root, this socket has been bound to a privileged port during `ls_initrex()`. `ls_connect()` uses a socket created by the preceding invocation of `ls_initrex()` and invokes `connect()` to connect to the specified host. If the connection fails, -1 is returned and the socket is closed.

The successful return of `ls_connect()` does not mean that the RES has granted remote execution permission, it means that the authentication process has been initiated. If the RES does not grant remote execution permission, an error is returned in the next interaction with the RES. Calling `ls_connect()` multiple times with the same host name does not create multiple connections; the same connection is always used.

SYNOPSIS

```
#include <lsf.h>
int ls_connect(char *host)
```

PARAMETERS

***host**	The host that is set up with a Remote Execution Server.

RETURN VALUES

character:	Function was successful.
character:-1	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_rexecv()`

`ls_initrex()`

`ls_isconnected()`

`ls_findmyconnections()`

Equivalent line command

none

Files

none

ls_conntaskport()

Connects a socket to a remote task port.

DESCRIPTION

`ls_conntaskport()` connects a socket to the task port that was created by the remote RES for the remote task `tid` returned by an `ls_rtask()` or `ls_rtaske()` call. You must have started the remote task with the `REX_TASKPORT` flag (see `ls_rtask()`).

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf.h>
int ls_conntaskport(int tid)
```

PARAMETERS

tid

RETURN VALUES

integer:(non-zero?)	Function was successful
integer:-1	Function failed

A connected socket is returned on success.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_initrex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_deleteltask()

Deletes the specified task from the local task list.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsltasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_deleteltask()` deletes the specified `task` from the local task list.

`task` is a character string containing a task name.

For example,

`"cc/select[swp>20 || (mem>10 && pg<5)] order[swp:pg] rusage[swp=20]"`

See *Administering Platform LSF* for a description of the resource requirement string.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_deleteltask(char *task)
```

PARAMETERS

***task**	The task to be deleted from the local task list.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

`ls_resreq()`

`ls_listrtask()`

SEE ALSO

`ls_listltask()`

`ls_inesrtrtask()`

`ls_inesrtltask()`

`ls_deletertask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_deletertask()

Deletes a specified task from the remote task list.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsrtasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_deletertask()` deletes the specified task from the remote task list.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_deletertask(char *task)
```

PARAMETERS

***task** `task` is a character string containing the name of the task to be deleted and, for a remote task, optionally a resource requirement string, separated by '/'.

For example,

`"cc/select[swp>20 || (mem>10 && pg<5)] order[swp:pg] rusage[swp=20]"`

See *Administering Platform LSF* for a description of the resource requirement string.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

`ls_resreq()`

`ls_listrtask()`

SEE ALSO

`ls_listltask()`

`ls_inesrtrtask()`

`ls_inesrtltask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_donerex()

Kills the Network I/O server (NIOS) and restores the tty environment before a remote execution connection is closed.

DESCRIPTION

`ls_donerex()` kills the Network I/O server (NIOS) and restores the tty environment before a remote execution connection is closed. You need to call this routine only if a remote execution was started by either `ls_rtask()` or `ls_rtaske()`, and the option `REXF_USEPTY` was set. If the application exits without calling this routine, the terminal may behave abnormally on occasion. If the option `REXF_USEPTY` is not specified when either `ls_rtask()` or `ls_rtaske()` is called, there is no need to call `ls_donerex()`.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_donerex(void)
```

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_initrex()`

`ls_rtask()`

`ls_rtaske()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_eligible()

Checks to see if a task is eligible for remote execution.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsrtasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_eligible()` checks whether or not `taskname` is eligible for remote execution and, if so, obtains its resource requirements. `ls_eligible()` returns `TRUE` if `taskname` is eligible for remote execution, `FALSE` otherwise.

`resreq` is an output parameter; you supply the character array. If `taskname` is eligible for remote execution, the resource requirements associated with `taskname` in the remote task lists are copied into `resreq`. If no resource requirements are associated with `taskname`, an empty string is copied into `resreq`.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_eligible(char *taskname, char *resreq, char mode)
```

PARAMETERS

***taskname**	The task that is being checked to see if it can be remotely executed.
***resreq**	The resource requirements associated with the task `taskname`. If no resource requirements are associated with `taskname`, an empty string is copied into `resreq`.
mode	One of two constants defined in `<lsf/lsf.h>`. If `mode` is `LSF_LOCAL_MODE`, the routine searches through the remote task lists to see if `taskname` is on a list. If found, the task is considered eligible for remote execution, otherwise the task is considered ineligible. If `mode` is `LSF_REMOTE_MODE`, the routine searches through the local task lists to see if `taskname` is on a list. If found, the task is considered ineligible for remote execution, otherwise the task is considered eligible.

RETURN VALUES

character:TRUE	Returned if the task can be remotely executed.
character:FALSE	Returned if the task can not be remotely executed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_resreq()`

`ls_listrtask()`

`ls_listltask()`

`ls_insertrtask()`

`ls_inesrtltask()`

`ls_deletertask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_errlog()

Logs error messages.

DESCRIPTION

`ls_errlog()` is a LSLIB library routine for logging LSF error messages. The global variable `lserrno`, maintained by LSLIB, indicates the error number of the most recent LSLIB call that caused an error.

`ls_errlog()` is very similar to the `fprintf()` function, except that it prints out the time before it prints other information. You can specify an additional conversion character ‘`m`’ in the format to represent the error message that corresponds to `lserrno`. This function is typically used by load sharing applications running in the background (such as daemons) to log error messages to a log file.

The vector of error message strings, `ls_errmsg[ ]`, is provided for the benefit of application programs. You can use `lserrno` as an index into this table to obtain the corresponding LSF error message. The global variable `ls_nerr` indicates the size of the table.

SYNOPSIS

```
#include <lsf/lsf.h>
void ls_errlog(FILE *fp, char *fmt, ...)

char *ls_errmsg[ ];
int lserrno;
int ls_nerr;
```

PARAMETERS

***fp**	.
***fmt**	.
ls_errmsg	Vector of error message strings, provided for the benefit of application programs.
lserrno	Index into this table to obtain the corresponding LSF error message.
ls_nerr	Indicates the size of the table.

RETURN VALUES

void: There is no return value.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_perror()`

`ls_sysmsg()`

Equivalent line command

none

Files

`lsf/lsf.h`

ls_fdbusy()

Tests if a specified file descriptor is in use or reserved by LSF.

DESCRIPTION

`ls_fdbusy()` tests if a specified file descriptor is in use or reserved by LSF. The possible descriptors used by LSF are those used for contacting the LIM and for remote execution. `fd` is the file descriptor to test. This call is typically used when an application wants to close all unneeded file descriptors.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_fdbusy(int fd)
```

PARAMETERS

fd	`fd` is the file descriptor to test.

RETURN VALUES

integer:True	Returned if `fd` is in use or reserved by LSF.
integer:False	Returned if `fd` is not in use or reserved by LSF.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_initrex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_findmyconnections()

Finds established connections to hosts.

DESCRIPTION

`ls_findmyconnections()` retrieves the list of hosts with which the application has established a connection. A connection is established upon the first successful return from an `ls_connect()`, `ls_rtask()`, `ls_rtaske()`, `ls_rexecv()`, or `ls_rexecve()` call. The returned host name list is terminated by a NULL value. The function maintains a static array, each element of which points to a host name. This array is overwritten the next time `ls_findmyconnections()` is called. If a connection is not found, the first element of the array is a NULL pointer.

Any program using these routines must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf.h>
char **ls_findmyconnections(void)
```

RETURN VALUES

character:NULL	Function was successful.
character:	

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_connect()`

`ls_isconnected()`

`ls_rtask()`

`ls_rtaske()`

`ls_rexecv()`

`ls_rexecve()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_getclustername()

Returns the name of the local load sharing cluster.

DESCRIPTION

This routines provides access to LSF cluster configuration information.

`ls_getclustername()` returns the name of the local load sharing cluster defined in the configuration files.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_getclustername(void)
```

RETURN VALUES

character

character:NULL

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_info()` – Returns a pointer to an `lsInfo` structure

`ls_getmastername()`

`ls_getmodelfactor()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_gethostfactor()

Returns a pointer to a floating point number that contains the CPU factor of the specified host.

DESCRIPTION

This routine obtains static resource information about hosts. Static resources include configuration information as determined by LSF configuration files (see `lsf.shared` and `lsf.cluster`) as well as others determined automatically by LIM at start up.

`ls_gethostfactor()` returns a pointer to a floating point number that contains the CPU factor of the specified host.

This routine returns a pointer to dynamically allocated data structures which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
float *ls_gethostfactor(char *hostname)
```

PARAMETERS

***hostname**	The host to which the floating point number is returned.

RETURN VALUES

ls_gethostfactor	Returns a pointer to a floating point number that contains the CPU factor of the specified host.
char:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_gethosttype()`

`ls_gethostmodel()`

`ls_gethostinfo()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_gethostinfo()

Returns an array of `hostInfo` data structures.

DESCRIPTION

This routine obtains static resource information about hosts. Static resources include configuration information as determined by LSF configuration files (see `lsf.shared` and `lsf.cluster`) as well as others determined automatically by LIM at start up.

`ls_gethostinfo()` returns an array of `hostInfo` data structures, as defined in `<lsf/lsf.h>`. Each entry contains information about one host, including the host type, the host model, its CPU normalization factor, the number of CPUs, its maximum memory, swap and tmp space, number of disks, the resources available on the host, the run windows during which the host is available for load sharing, the busy thresholds for the host, whether the host is a LSF server, and the default priority used by the RES for remote tasks executing on that host. The `windows` field will be set to "–" if the host is always open. The `busyThreshold` field is an array of floating point numbers specifying the load index thresholds that LIM uses to consider a host as busy. The size of the array is indicated by the `numIndx` field. The order of the array elements is the same as the load indicies returned by `ls_load()`.

This routine returns a pointer to dynamically allocated data structures which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
struct hostInfo *ls_gethostinfo(char *resreq, int *numhosts,
                 char **hostlist, int listsize, int options)

struct hostInfo {
    char hostName[MAXHOSTNAMELEN];
    char *hostType;
    char *hostModel;
    float cpuFactor;
    int maxCpus;
    int maxMem;
    int maxSwap;
    int maxTmp;
    int nDisks;
    int nRes;
    char **resources;
    char *windows;
    int numIndx;
    float *busyThreshold;
    char isServer;
    char licensed;
    int rexPriority;
    int licFeaturesNeeded;
    int licClass;
    int cores;
    char hostAddr[INET6_ADDRSTRLEN];
    int pprocs;
    int cores_per_proc;
    int threads_per_core;
};
```

PARAMETERS

***resreq**	*resreq* specifies resource requirements that a host must satisfy if it is to be included in the `hostInfo` array returned. See *Administering Platform LSF* for information about resource requirement string syntax. If this parameter is a `NULL` pointer or is an empty string, then the default resource requirement will be used, which is to return all hosts.
***numhosts**	*numhosts* is the address of an integer which, if it is not `NULL`, will contain the number of `hostInfo` records returned on success.
****hostlist**	*hostlist* gives a list of hosts or clusters whose information is returned if they satisfy the requirements in *resreq*. If *hostlist* is `NULL`, all hosts known to LSF that satisfy the requirements in *resreq* are returned.
listsize	*listsize* gives the size of the `hostlist`.
options	*options* is constructed from the bitwise inclusive OR of zero or more of the flags that are defined in `<lsf/lsf.h>`. These flags are documented in `ls_load()`.

RETURN VALUES

struct	
struct:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_gethosttype()`

`ls_gethostmodel()`

`ls_gethostfactor()`

`ls_load()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_gethostmodel()

Returns the model of the specified host.

DESCRIPTION

This routine obtains static resource information about hosts. Static resources include configuration information as determined by LSF configuration files (see `lsf.shared` and `lsf.cluster`) as well as others determined automatically by LIM at start up.

`ls_gethostmodel()` returns the model of the specified host.

This routine returns a pointer to dynamically allocated data structures which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_gethostmodel(char *hostname)
```

PARAMETERS

***hostname**	The host whose model is to be determined.

RETURN VALUES

char	Function was successful.
char:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_gethosttype()`

`ls_gethostfactor()`

`ls_gethostinfo()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_gethosttype()

Returns the type of the specified host.

DESCRIPTION

This routine obtains static resource information about hosts. Static resources include configuration information as determined by LSF configuration files (see `lsf.shared` and `lsf.cluster`) as well as others determined automatically by LIM at start up.

`ls_gethosttype()` returns the type of the specified host.

This routine returns a pointer to dynamically allocated data structures which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_gethosttype(char *hostname)
```

PARAMETERS

***hostname**	The host whose type is to be determined.

RETURN VALUES

char	Function was successful.
char:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_gethostfactor()`

`ls_gethostmodel()`

`ls_gethostinfo()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_getmastername()

Returns the name of the host running the local load sharing cluster’s master LIM.

DESCRIPTION

This routines provides access to LSF cluster configuration information.

`ls_getmastername()` returns the name of the host running the local load sharing cluster’s master LIM.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_getmastername(void)
```

RETURN VALUES

character

character:NULL

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_info()` – Returns a pointer to an `lsInfo` structure

`ls_getclustername()`

`ls_getmodelfactor()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_getmnthost()

Returns the name of the file server containing a specific file.

DESCRIPTION

`ls_getmnthost()` returns the name of the file server that exports the file system containing `file`, where `file` is a relative or absolute path name. For remote files, use `ls_rgetmnthost()` instead.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_getmnthost(char *file)
```

PARAMETERS

***file**	The relative or absolute path name for the file server.

RETURN VALUES

character:file server	Function was successful.
character:NULL	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_rgetmnthost()`

Equivalent line command

none

Files

none

ls_getmodelfactor()

Returns the CPU normalization factor of the specified host model.

DESCRIPTION

This routines provides access to LSF cluster configuration information.

`ls_getmodelfactor()` returns the CPU normalization factor of the specified host model as defined in the LSF configuration files. This factor is based on the the host model's CPU speed relative to other host models in the load sharing system.

SYNOPSIS

```
#include <lsf/lsf.h>
float *ls_getmodelfactor(char *modelname)
```

PARAMETERS

***modelname**	The model for which the CPU nominalization factor is returned.

RETURN VALUES

character

character:NULL

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_info()` – Returns a pointer to an `lsInfo` structure

`ls_getclustername()`

`ls_getmastername()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_getmyhostname2()

New. Returns the name used throughout LSF to represent the local host.

DESCRIPTION

`ls_getmyhostname2()` returns the name used throughout LSF to represent the local host.

This function supports the use of both IPv4 and IPv6 protocols. Call this function rather than ls_getmyhostname().

This routine returns a pointer to static data which can be overwritten by subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_getmyhostname2(void)
```

RETURN VALUES

character:hostname	Function was successful.
character:NULL	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

ls_getmyhostname

Equivalent line command

none

Files

none

ls_getmyhostname()

Deprecated. Returns the name used throughout LSF to represent the local host.

DESCRIPTION

`ls_getmyhostname()` returns the name used throughout LSF to represent the local host.

This routine returns a pointer to static data which can be overwritten by subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_getmyhostname(void)
```

RETURN VALUES

character:hostname	Function was successful.
character:NULL	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

ls_getmyhostname2()

Equivalent line command

none

Files

none

ls_getstdin()

Allows an application program to query and specify how stdin is assigned to remote tasks.

DESCRIPTION

`ls_getstdin()` gives an application program the ability to query and specify how stdin is assigned to remote tasks. It allows you to assign stdin to to all remote tasks. You can change this setting at any time.

`ls_getstdin()` gets the list of remote task IDs that receive (or do not receive) standard input. If `on` is non-zero, the task IDs of the remote tasks that are enabled to receive standard input are stored in `tidlist. maxlen` is the size of the `tidlist` array. If `on` is zero, then the IDs of remote tasks whose standard input is disabled are returned. The ID of a task is assigned by the LSLIB when `ls_rtask()` is called.

Upon success, `ls_getstdin()` returns the number of entries stored in `tidlist`. On failure, -1 is returned, and the error code is stored in `lserrno`. In particular, if there are more than `maxlen` remote task IDs to be returned, `lserrno` is set to `LSE_RPIDLISTLEN`.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_getstdin(int on, int *tidlist, int *maxlen)
```

PARAMETERS

on If `on` is non-zero, the task IDs of the remote tasks that are enabled to receive standard input are stored in `tidlist`. If `on` is zero, then the IDs of remote tasks whose standard input is disabled are returned. The ID of a task is assigned by the LSLIB when `ls_rtask()` is called.

***tidlist**

***maxlen** The size of the `tidlist` array.

RETURN VALUES

integer:# of Entries Stored

The function was successful.

integer:-1

Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error. If there are more than `maxlen` remote task IDs to be returned, `lserrno` is set to `LSE_RPIDLISTLEN`.

SEE ALSO

Related APIs

`ls_stdinmodel()`

`ls_setstdin()`

`ls_initrex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_info()

Returns a pointer to an lsInfo structure.

DESCRIPTION

This routine provides access to LSF cluster configuration information.

On success, `ls_info()` returns a pointer to an `lsInfo` structure, which contains complete load sharing configuration information. This information includes the name of the cluster, the name of the current cluster master host, the set of defined resources, the set of defined host types and models, the CPU factors of the host models, and all load indices (`resTable[0]` through `resTable[numIndx - 1]`), including the site defined external load indices (`resTable[MAX + 1]` through `resTable[MAX + numUsrIndx]`).

The set of defined resource items is a list of all resources that may be assigned to various hosts in the cluster. The resource names can be used to build expressions for querying information about hosts, or for describing how tasks are to be scheduled. New resources may be defined as desired by the LSF administrator. See `ls_task()` for more information about how resource names can be used to describe resource requirements.

The `valueType` component of the `resItem` structure indicates whether the type of the resource is `NUMERIC`, `STRING`, or `BOOLEAN`.

The `orderType` indicates how hosts should be ordered from best to worst based on the resource. If the `orderType` is `INCR`, the hosts should be ordered from the lowest to the highest value for that resource; if `DECR`, they should be ordered from the highest to lowest. If the `orderType` is `NA`, then the resource cannot be used to order hosts.

The `flags` component is used to indicate the attributes of the resource. It is formed from the bitwise inclusive OR of zero or more of the following flags, as defined in `<lsf/lsf.h>`:

RESF_BUILTIN	Indicate whether this resource is builtin to LSF or configured by the LSF administrator (external).
RESF_DYNAMIC	Indicate whether the value of this resource can change dynamically or is static. Information about dynamic resources for a host can be retrieved through `ls_load()`. Information about static resources can be retrieved through `ls_gethostinfo()`.
RESF_GLOBAL	Indicate whether the resource name is defined for every host in the cluster. The value of the resource is specific to each host. This type of resource is also called a non-shared resource.
RESF_SHARED	Indicate whether the resource is a shared resource. A shared resource is a resource whose value is shared by more than one host, and the resource may be defined only on a subset of the hosts.

The `interval` component applies to resources with dynamic values. It indicates how frequently (in seconds) the resource value is evaluated.

The set of host types `hostTypes` in the `lsInfo` structure is a list of all defined host architectures in the cluster. All machines that can run the same binaries are generally considered to be of the same host type.

The set of host models `hostModels` in `lsInfo` structure is a list of all defined computer models in the cluster. Generally, machines of the same host type that have exactly the same performance characteristics are considered to be the same model.

SYNOPSIS

```
#include <lsf/lsf.h>
struct lsInfo *ls_info(void)

struct lsInfo {
    int nRes;
    struct resItem *resTable;
    int nTypes;
    char hostTypes[MAXTYPES][MAXLSFNAMELEN];
    int nModels;
    char hostModels[MAXMODELS][MAXLSFNAMELEN];
    char hostArchs[MAXMODELS][MAXLSFNAMELEN_70_EP1];
    int modelRefs[MAXMODELS];
    float cpuFactor[MAXMODELS];
    int numIndx;
    int numUsrIndx;
};

struct resItem {
    char name[MAXLSFNAMELEN];
    char des[MAXRESDESLEN];
    enum valueType valueType;
    enum orderType orderType;
    int flags;
    int interval;
};

enum valueType {LS_BOOLEAN, LS_NUMERIC, LS_STRING};
enum orderType {INCR, DECR, NA};
```

RETURN VALUES

struct:

struct:NULL

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_getclustername()`

`ls_getmastername()`

`ls_getmodelfactor()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_initrex()

Initializes the LSF library for remote execution.

DESCRIPTION

`ls_initrex()` initializes the LSF library for remote execution. This routine must be called before any other remote execution LSLIB library routines (see `ls_rex()`) can be used.

Two remote execution security options are supported in LSF. The first option is to set the effective user ID of an LSF application to root, as other UNIX applications that access remote resources (e.g., `rlogin`) do. Using this option, `numports` of the application's file descriptors are bound to privileged ports by `ls_initrex()`. These sockets are used only for remote connections to RES. If `numports` is 0, then the system will use the default value `LSF_DEFAULT_SOCKS` defined in `<lsf/lsf.h>`. If successful, the number of socket descriptors starting from `FIRST_RES_SOCK` (defined in `<lsf/lsf.h>`) that are actually bound to privileged ports is returned, -1 otherwise. To use this option for authentication, the application must be installed as setuid to root. The second security option is to use an authentication daemon supporting the Ident protocol (RFC 931/1413/1414). In this case, this routine returns the value of the input parameter `numports` if it succeeds, -1 otherwise.

`ls_initrex()` selects the security option according to the following rule: if the application program invoking it has the effective uid of root, then privileged ports are created; otherwise, no such port is created, and RES will contact an authentication daemon on a connection request (see `ls_connect()`).

Currently, the only option that can be specified in `options` is `KEEPUID`, which instructs `ls_initrex()` to preserve the current user ID. If the `KEEPUID` bit is not set in `options` (i.e. `options` is zero), then `ls_initrex()` will change the real, effective and saved user ID to the real user ID.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

This function must be called before calling any other remote execution function.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_initrex(int numports, int options)
```

PARAMETERS

numports

hostname

RETURN VALUES

integer:numports	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_connect()`

`ls_rex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_insertltask()

Adds the specified task to the local task list.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsltasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_insertltask()` adds the specified `task` to the local task list.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_insertltask(char *task)
```

PARAMETERS

***task** `task` is a character string containing the name of the task to be inesrted.

For example,

`"cc/select[swp>20 || (mem>10 && pg<5)] order[swp:pg] rusage[swp=20]"`

See *Administering Platform LSF* for a description of the resource requirement string.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

`ls_resreq()`

`ls_listrtask()`

`ls_listltask()`

`ls_inesrtrtask()`

`ls_deletertask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_insertrtask()

Adds a specified task to the remote task list.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsrtasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_insertrtask()` adds the specified `task` to the remote task list.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_insertrtask(char *task)
```

PARAMETERS

***task** `task` is a character string containing the name of the task to be inesrted and, for a remote task, optionally a resource requirement string, separated by '/'.

For example,

`"cc/select[swp>20 || (mem>10 && pg<5)] order[swp:pg] rusage[swp=20]"`

See *Administering Platform LSF* for a description of the resource requirement string.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

`ls_resreq()`

`ls_listrtask()`

`ls_listltask()`

`ls_inesrtltask()`

`ls_deletertask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_isconnected()

Tests if the specified host is currently connected with the application.

DESCRIPTION

`ls_isconnected()` tests if the specified host is currently connected with the application. A connection is established on the first successful return from an `ls_connect()`, `ls_rtask()`, `ls_rtaske()`, `ls_rexecv()`, or `ls_rexecve()` call.

Any program using these routines must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf.h>
int ls_connect(char *host)
```

PARAMETERS

***host**	The host that is tested for connection with an application.

RETURN VALUES

integer:non-zero	Function was successful.
integer:0	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_connect()`

`ls_findmyconnections()`

`ls_rtask()`

`ls_rtaske()`

`ls_rexecv()`

`ls_rexecve()`

Equivalent line command

none

Files

none

ls_limcontrol()

Shuts down or reboots a host's LIM.

DESCRIPTION

To remove a host from a cluster, use `ls_limcontrol()` to shut down the host's LIM. Next, to return a removed host to a cluster, use `ls_limcontrol()` to reboot its LIM. When you reboot the LIM, the configuration files are read again and the previous LIM status of the host is lost.

The use of `ls_limcontrol()` is restricted to root and the LSF administrator as defined in the file `LSF_CONFDIR/lsf.cluster.cluster_name`.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_limcontrol(char *hostname, int opCode)
```

PARAMETERS

hostname

Specifies the host

opCode

Specifies the shutdown or reboot LIM command.

LIM_CMD_SHUTDOWN

Command to shutdown the LIM.

LIM_CMD_REBOOT

Command to reboot the LIM.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs:

`ls_lockhost()` - locks a local host

`ls_unlockhost()` - unlocks a local host

SEE ALSO

Equivalent line command

```
lsadmin limstartup
lsadmin limshutdown
lsadmin limrestart
```

Files:

```
${LSF_ENVDIR-/etc}/lsf.conf
$LSF_CONFDIR/lsf.shared
$LSF_CONFDIR/lsf.cluster.cluster_name
```

ls_listltask()

Returns the contents of the user's local task list.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsltasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_listltask()` returns the contents of a user's local task list in `tasklist`. Memory for `tasklist` is allocated as needed and freed by the next call to `ls_listltask()`. If `sortflag` is non-zero, then the returned task list is sorted alphabetically. Each of the returned tasks is a character string consisting of a task name optionally followed by '/' and the associated resource requirement string. `ls_listltask()` return the number of items in the `tasklist` on success, -1 on error.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_listltask(char ***taskList, int sortflag)
```

*****taskList**	The task list to be accessed.
sortflag	

RETURN VALUES

integer:Number of Items in Task List

The function was successful.

integer:-1

Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

SEE ALSO

`ls_resreq()`

`ls_listrtask()`

`ls_insertrtask()`

`ls_inesrtltask()`

`ls_deletertask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_listrtask()

Returns the contents of the user's remote task list.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsrtasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_listrtask()` returns the contents of a user's remote task list in `tasklist`. Memory for `tasklist` is allocated as needed and freed by the next call to `ls_listrtask()`. If `sortflag` is non-zero, then the returned task list is sorted alphabetically. Each of the returned tasks is a character string consisting of a task name optionally followed by '/' and the associated resource requirement string. `ls_listrtask()` return the number of items in the `tasklist` on success, -1 on error.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_listrtask(char ***taskList, int sortflag)
```

PARAMETERS

*****taskList**	The task list to be accessed.
sortflag	

RETURN VALUES

integer:Number of Items in Task List

The function was successful.

integer:-1

Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

`ls_resreq()`

`ls_listltask()`

`ls_insertrtask()`

`ls_inesrtltask()`

`ls_deletertask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_load()

Returns all load indices.

DESCRIPTION

This routine returns the dynamic load information of qualified hosts.

`ls_load()` returns all load indices. The result of this call is an array of `hostLoad` data structures as defined in `<lsf/lsf.h>`. The `status` component of the `hostLoad` structure is an array of integers. The high order 16 bits of the first integer are used to mark the operation status of the host. Possible states defined in `<lsf/lsf.h>` are as follows:

LIM_UNAVAIL	The host Load Information Manager (LIM) is unavailable (e.g. the host is down or there is no LIM). If LIM is unavailable the other information in the `hostLoad` structure is meaningless.
LIM_BUSY	The host is busy (overloaded).
LIM_LOCKEDU	The host's LIM is locked by the root, LSF administrator or a user.
LIM_LOCKEDW	The host's LIM is locked by its run windows.
LIM_RESDOWN	The host's Remote Execution Server (RES) is not available.
LIM_UNLICENSED	The host has no software license.

The low order 16 bits of the first integer are reserved. The other `integer()` of the status array is used to indicate the load status of the host. If any of these bits is set, then the host is considered to be busy (overloaded). Each bit (starting from bit 0) in `integer()` represents one load index that caused the host to be busy. If bit i is set then the load index corresponding to `li[i]` caused the host to be busy. An integer can be used to for 32 load indices. If number of load indices on the host, both built-in and user defined, are more than 32, more than one integer will be used.

Programmers can use macros to test the status of a host. The most commonly used macros include:

```
LS_ISUNAVAIL(status)
LS_ISBUSY(status)
LS_ISBUSYON(status, index)
LS_ISLOCKEDU(status)
LS_ISLOCKEDW(status)
LS_ISLOCKED(status)
LS_ISRESDOWN(status)
LS_ISUNLICENSED(status)
LS_ISOK(status)
```

In the `hostLoad` data structure, the `li` vector contains load information on various resources on a host. The elements of the load vector are determined by the *namelist* parameter.

SYNOPSIS

```
#include <lsf/lsf.h>
```

```
struct hostLoad *ls_load(char *resreq, int *numhosts,
                   int options, char *fromhost)
struct hostLoad {
    char hostName[MAXHOSTNAMELEN];
    int *status;
    float *li;
};
```

PARAMETERS

***resreq** `resreq` is a character string describing resource requirements. Only the load vectors of the hosts satisfying the requirements will be returned. If `resreq` is `NULL`, the load vectors of all hosts will be returned.

***numhosts** `numhosts` is the address of an integer which initially contains the number of hosts requested. If `*numhosts` is 0, request information on as many hosts as satisfy `resreq`. If `numhosts` is `NULL`, requests load information on one (1) host. If `numhosts` is not `NULL`, then `*numhosts` will contain the number of `hostLoad` records returned on success.

options `options` is constructed from the bitwise inclusive OR of zero or more of the following flags, as defined in `<lsf/lsf.h>`.

EXACT

Exactly `*numhosts` hosts are desired. If `EXACT` is set, either exactly `*numhosts` hosts are returned, or the call returns an error. If `EXACT` is not set, then up to `*numhosts` hosts are returned. If `*numhosts` is zero, then the `EXACT` flag is ignored and as many hosts in the load sharing system as are eligible (that is, those that satisfy the resource requirements) are returned.

OK_ONLY

Return only those hosts that are currently in the ‘`ok`’ state. If `OK_ONLY` is set, those hosts that are busy, locked, or unavail are not returned. If `OK_ONLY` is not set, then some or all of the hosts whose status are not ‘`ok`’ may also be returned, depending on the value of `*numhosts` and whether the `EXACT` flag is set.

NORMALIZE

Normalize CPU load indices. If `NORMALIZE` is set, then the CPU run queue length load indices `r15s`, `r1m`, and `r15m` of each host returned are normalized. See *Administering Platform LSF* for the concept of normalized queue length. Default is to return the raw queue length. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

EFFECTIVE

If `EFFECTIVE` is set, then the CPU run queue length load indices of each host returned are effective load. See *Administering Platform LSF* for the concept of effective queue length. Default is to return the raw queue length. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

IGNORE_RES

Ignore the status of RES when determining the hosts that are considered to be ‘`ok`’. If `IGNORE_RES` is specified, then hosts with RES not running are also considered to be ‘`ok`’ during host selection.

DFT_FROMTYPE

Return hosts with the same type as the `fromhost` which satisfy the resource requirements. By default all host types are considered.

***fromhost** `fromhost` is the name of the host from which a task might be transferred. This parameter affects the host selection in such a way as to give preference to `fromhost` if the load on other hosts is not much better. If `fromhost` is `NULL`, the local host is assumed.

RETURN VALUES

character:

character:NULL Depends on which parameter is returned with NULL.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_loadinfo()`

`ls_loadofhosts()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_loadadj()

Sends a load adjustment request to LIM after the execution host or hosts have been selected outside the LIM by the calling application.

DESCRIPTION

`ls_loadadj()` sends a load adjustment request to LIM after the execution host or hosts have been selected outside the LIM by the calling application. Use this call only if a placement decision is made by the application without calling `ls_placereq()` (for example, a decision based on the load information from an earlier `ls_load()` call). This request keeps LIM informed of task transfers so that the potential load increase on the destination `host()` provided in `placeinfo` are immediately taken into consideration in future LIM placement decisions. listsize gives the total number of entries in `placeinfo`.

`ls_loadadj()` returns 0 on success, otherwise -1 is returned and `lserrno` is set to indicate the error.

`ls_loadadj()` can adjust all load indices with the exception of `ls`, `it`, `r15m` and external load indices.

SYNOPSIS

```
#include <lsf/lsf.h>

int ls_loadadj(char *resreq, struct placeInfo *placeinfo,
    int listsize)

struct placeInfo {
    char hostName[MAXHOSTNAMELEN];
    int numtask;
}
```

PARAMETERS

***resreq**	`resreq` is a resource requirement expression (which can be `NULL`) that describes the resource requirements for which the load must be adjusted. These typically are the resource requirements for the previously placed task (see `ls_task()`). LIM adjusts the host load indices according to the resource requirement. If `NULL` is specified, then LIM assumes that the task or tasks are both CPU and memory intensive (this is the default).
***placeinfo**	`placeinfo` is a pointer to an array of `placeInfo` structures. A `placeInfo` structure contains a `hostname`, and an integer, `numtask`, that represents a particular number of tasks. The host load indices (specified by `resreq`) of all the hosts that are specified in the array are increased by the number of tasks specified. Each task is assumed to have the same resource requirements. The requirements are those specified in `resreq`.
listsize	
hostname	Name of the host.
numtask	Number of tasks on the host.

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_placereq()`

`ls_placeofhosts()`

`ls_eligible()`

`ls_info()` – Returns a pointer to an `lsInfo` structure

`ls_load()`

`ls_task()`

Equivalent line command

none

Files

none

ls_loadinfo()

Returns the requested load indices of the hosts that satisfy specified resource requirements.

DESCRIPTION

This routine returns the dynamic load information of qualified hosts.

`ls_loadinfo()` returns the requested load indices of the hosts that satisfy the specified resource requirements. The result of this call is an array of `hostLoad` data structures as defined in `<lsf/lsf.h>`. The `status` component of the `hostLoad` structure is an array of integers. The high order 16 bits of the first integer are used to mark the operation status of the host. Possible states defined in `<lsf/lsf.h>` are as follows:

LIM_UNAVAIL	The host Load Information Manager (LIM) is unavailable (e.g. the host is down or there is no LIM). If LIM is unavailable the other information in the `hostLoad` structure is meaningless.
LIM_BUSY	The host is busy (overloaded).
LIM_LOCKEDU	The host’s LIM is locked by the root, LSF administrator or a user.
LIM_LOCKEDW	The host’s LIM is locked by its run windows.
LIM_RESDOWN	The host’s Remote Execution Server (RES) is not available.
LIM_UNLICENSED	The host has no software license.

The low order 16 bits of the first integer are reserved. The other `integer()` of the status array is used to indicate the load status of the host. If any of these bits is set, then the host is considered to be busy (overloaded). Each bit (starting from bit 0) in `integer()` represents one load index that caused the host to be busy. If bit i is set then the load index corresponding to `li[i]` caused the host to be busy. An integer can be used to for 32 load indices. If number of load indices on the host, both built-in and user defined, are more than 32, more than one integer will be used.

Programmers can use macros to test the status of a host. The most commonly used macros include:

```
LS_ISUNAVAIL(status)
LS_ISBUSY(status)
LS_ISBUSYON(status, index)
LS_ISLOCKEDU(status)
LS_ISLOCKEDW(status)
LS_ISLOCKED(status)
LS_ISRESDOWN(status)
LS_ISUNLICENSED(status)
LS_ISOK(status)
```

In the `hostLoad` data structure, the `li` vector contains load information on various resources on a host. The elements of the load vector are determined by the `namelist` parameter.

The returned hostLoad array is sorted according to the order section of the resource requirements, `resreq` (or, if not specified, the 1-minute average CPU queue length and paging rate), with the lightest loaded host being the first.

`hostlist` is an array of `listsize` host or cluster names. If not `NULL`, then only load information about hosts in this list will be returned.

SYNOPSIS

```
#include <lsf/lsf.h>

struct hostLoad *ls_loadinfo(char *resreq, int *numhosts,
    int options, char *fromhost, char **hostlist,
    int listsize, char ***namelist)

struct hostLoad {
    char hostName[MAXHOSTNAMELEN];
    int *status;
    float *li;
};
```

PARAMETERS

***resreq** `resreq` is a character string describing resource requirements. Only the load vectors of the hosts satisfying the requirements will be returned. If `resreq` is `NULL`, the load vectors of all hosts will be returned.

***numhosts** `numhosts` is the address of an integer which initially contains the number of hosts requested. If `*numhosts` is 0, request information on as many hosts as satisfy `resreq`. If `numhosts` is `NULL`, requests load information on one (1) host. If `numhosts` is not `NULL`, then `*numhosts` will contain the number of `hostLoad` records returned on success.

options `options` is constructed from the bitwise inclusive OR of zero or more of the following flags, as defined in `<lsf/lsf.h>`.

EXACT

Exactly `*numhosts` hosts are desired. If `EXACT` is set, either exactly `*numhosts` hosts are returned, or the call returns an error. If `EXACT` is not set, then up to `*numhosts` hosts are returned. If `*numhosts` is zero, then the `EXACT` flag is ignored and as many hosts in the load sharing system as are eligible (that is, those that satisfy the resource requirements) are returned.

OK_ONLY

Return only those hosts that are currently in the ‘`ok`’ state. If `OK_ONLY` is set, those hosts that are busy, locked, or unavail are not returned. If `OK_ONLY` is not set, then some or all of the hosts whose status are not ‘`ok`’ may also be returned, depending on the value of `*numhosts` and whether the `EXACT` flag is set.

NORMALIZE

Normalize CPU load indices. If `NORMALIZE` is set, then the CPU run queue length load indices `r15s`, `r1m`, and `r15m` of each host returned are normalized. See *Administering Platform LSF* for the concept of normalized queue length. Default is to return the raw queue length. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

EFFECTIVE

If `EFFECTIVE` is set, then the CPU run queue length load indices of each host returned are effective load. See *Administering Platform LSF* for the concept of effective queue length. Default is to return the raw queue length. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

IGNORE_RES

Ignore the status of RES when determining the hosts that are considered to be ‘`ok`’. If `IGNORE_RES` is specified, then hosts with RES not running are also considered to be ‘`ok`’ during host selection.

DFT_FROMTYPE

Return hosts with the same type as the `fromhost` which satisfy the resource requirements. By default all host types are considered.

__**fromhost__ `fromhost` is the name of the host from which a task might be transferred. This parameter affects the host selection in such a way as to give preference to `fromhost` if the load on other hosts is not much better. If `fromhost` is `NULL`, the local host is assumed.

__**hostlist__ An array of `listsize` host or cluster names.

listsize

__***namelist__ `namelist` is an input/output parameter. On input it points to a null-terminated list of names of indices whose values will be returned in the `li` vector for each host selected. Setting `namelist` to point to `NULL` returns all indices. On return it points to a null-terminated list of the names of the indices returned in the `li` load vector for each host. Each element of the load vector is a floating point number between 0.0 and `INFINIT_LOAD` (defined in `lsf.h`). The index value is set to `INFINIT_LOAD` to indicate an invalid or unknown value for an index. The indices in the `li` vector are ordered such that `li[i]` contains the value of index `namelist[i]`. If index `namelist[i]` is causing the host to be busy, then `LS_ISBUSYON(status, i)` will be `TRUE`. When the input `namelist` is `NULL` the output `namelist` is ordered such that the `li` vector can be indexed using constants defined in `<lsf/lsf.h>` as listed below:

li[R15S]

15-second exponentially averaged CPU run queue length.

li[R1M]

1-minute exponentially averaged CPU run queue length.

li[R15M]

15-minute exponentially averaged CPU run queue length.

li[UT]

CPU utilization exponentially averaged over the last minute (from 0.0 to 1.0).

li[IO]

Disk I/O rate exponentially averaged over the last minute, in KBytes per second.

li[PG]

Memory paging rate exponentially averaged over the last minute, in pages per second.

li[LS]

Number of current login users.

li[IT]

Idle time of the host (keyboard not touched on all logged in sessions), in minutes.

li[TMP]

Available free disk space in `/tmp`, in MBytes.

li[SWP]

Amount of currently available swap space, in MBytes.

li[MEM]

Amount of currently available memory, in MBytes.

RETURN VALUES

character:	
character:NULL	Depends on which parameter is returned with NULL.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_loadofhosts()`

`ls_load()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_loadofhosts()

Returns all load indices except those specified.

DESCRIPTION

This routine returns the dynamic load information of qualified hosts.

`ls_loadofhosts()` returns all load indices except for the ones excluded b the parameters. The result of this call is an array of `hostLoad` data structures as defined in `<lsf/lsf.h>`. The `status` component of the `hostLoad` structure is an array of integers. The high order 16 bits of the first integer are used to mark the operation status of the host. Possible states defined in `<lsf/lsf.h>` are as follows:

LIM_UNAVAIL	The host Load Information Manager (LIM) is unavailable (e.g. the host is down or there is no LIM). If LIM is unavailable the other information in the `hostLoad` structure is meaningless.
LIM_BUSY	The host is busy (overloaded).
LIM_LOCKEDU	The host's LIM is locked by the root, LSF administrator or a user.
LIM_LOCKEDW	The host's LIM is locked by its run windows.
LIM_RESDOWN	The host's Remote Execution Server (RES) is not available.
LIM_UNLICENSED	The host has no software license.

The low order 16 bits of the first integer are reserved. The other `integer()` of the status array is used to indicate the load status of the host. If any of these bits is set, then the host is considered to be busy (overloaded). Each bit (starting from bit 0) in `integer()` represents one load index that caused the host to be busy. If bit i is set then the load index corresponding to `li[i]` caused the host to be busy. An integer can be used to for 32 load indices. If number of load indices on the host, both built-in and user defined, are more than 32, more than one integer will be used.

Programmers can use macros to test the status of a host. The most commonly used macros include:

```
LS_ISUNAVAIL(status)
LS_ISBUSY(status)
LS_ISBUSYON(status, index)
LS_ISLOCKEDU(status)
LS_ISLOCKEDW(status)
LS_ISLOCKED(status)
LS_ISRESDOWN(status)
LS_ISUNLICENSED(status)
LS_ISOK(status)
```

SYNOPSIS

```
#include <lsf/lsf.h>
struct hostLoad *ls_loadofhosts(char *resreq, int *numhosts,
    int options, char *fromhost, char **hostlist,
    int listsize)
```

```
struct hostLoad {
    char hostName[MAXHOSTNAMELEN];
    int *status;
    float *li;
};
```

PARAMETERS

***resreq** `resreq` is a character string describing resource requirements. Only the load vectors of the hosts satisfying the requirements will be returned. If `resreq` is `NULL`, the load vectors of all hosts will be returned.

***numhosts** `numhosts` is the address of an integer which initially contains the number of hosts requested. If `*numhosts` is 0, request information on as many hosts as satisfy `resreq`. If `numhosts` is `NULL`, requests load information on one (1) host. If `numhosts` is not `NULL`, then `*numhosts` will contain the number of `hostLoad` records returned on success.

options `options` is constructed from the bitwise inclusive OR of zero or more of the following flags, as defined in `<lsf/lsf.h>`.

EXACT

Exactly `*numhosts` hosts are desired. If `EXACT` is set, either exactly `*numhosts` hosts are returned, or the call returns an error. If `EXACT` is not set, then up to `*numhosts` hosts are returned. If `*numhosts` is zero, then the `EXACT` flag is ignored and as many hosts in the load sharing system as are eligible (that is, those that satisfy the resource requirements) are returned.

OK_ONLY

Return only those hosts that are currently in the ‘`ok`’ state. If `OK_ONLY` is set, those hosts that are busy, locked, or unavail are not returned. If `OK_ONLY` is not set, then some or all of the hosts whose status are not ‘`ok`’ may also be returned, depending on the value of `*numhosts` and whether the `EXACT` flag is set.

NORMALIZE

Normalize CPU load indices. If `NORMALIZE` is set, then the CPU run queue length load indices `r15s`, `r1m`, and `r15m` of each host returned are normalized. See *Administering Platform LSF* for the concept of normalized queue length. Default is to return the raw queue length. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

EFFECTIVE

If `EFFECTIVE` is set, then the CPU run queue length load indices of each host returned are effective load. See *Administering Platform LSF* for the concept of effective queue length. Default is to return the raw queue length. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

IGNORE_RES

Ignore the status of RES when determining the hosts that are considered to be ‘`ok`’. If `IGNORE_RES` is specified, then hosts with RES not running are also considered to be ‘`ok`’ during host selection.

DFT_FROMTYPE

Return hosts with the same type as the `fromhost` which satisfy the resource requirements. By default all host types are considered.

***fromhost** `fromhost` is the name of the host from which a task might be transferred. This parameter affects the host selection in such a way as to give preference to `fromhost` if the load on other hosts is not much better. If `fromhost` is `NULL`, the local host is assumed.

****hostlist** `hostlist` is an array of `listsize` host or cluster names. If not `NULL`, then only load information about hosts in this list will be returned.

listsize

RETURN VALUES

character:

character:NULL Depends on which parameter is returned with `NULL`.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_loadinfo()`

`ls_load()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_lockhost()

Locks the local host for a specified number of seconds.

DESCRIPTION

`ls_lockhost()` prevents a host from being selected by the master LIM for task or job placement. If the host is locked for 0 seconds, it remains locked until it is explicitly unlocked by `ls_unlockhost()`. Indefinitely locking a host is useful if a job or task must run exclusively on the local host, or if machine owners want private control over their machines.

A program using `ls_lockhost()` must be setuid to root in order for the LSF administrator or any other user to lock a host.

To lock a host, use the setuid function (int setuid(uid_t uid)) to set the effective user id of the calling process to root or LSF administrator. On success, this API changes the status of the local host to indicate that it has been locked by the user.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_lockhost(time_t duration)
```

PARAMETERS

duration	The number of seconds the local host is locked. 0 seconds locks a host indefinitely.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

On failure, `lserrno` is set to indicate the error. If the host is already locked, `ls_lockhost()` sets `lserrno` to LSE_LIM_ALOCKED.

SEE ALSO

Related APIs:

`ls_limcontrol()` - shuts down or reboots a host's LIM

`ls_unlockhost()` - unlocks a locked host

Equivalent line command

```
lsadmin limlock
```

Files:

```
${LSF_ENVDIR-/etc}/lsf.conf
$LSF_CONFDIR/lsf.shared
$LSF_CONFDIR/lsf.cluster.cluster_name
```

ls_perror()

Prints LSF error messages.

DESCRIPTION

`ls_perror()` is a LSLIB library routine for printing LSF error messages. The global variable `lserrno`, maintained by LSLIB, indicates the error number of the most recent LSLIB call that caused an error.

`ls_perror()` prints on the standard error output the character string `usrMsg`, if it is not a null pointer, followed by a colon and a space, then an error message that describes the error that corresponds to the value of `lserrno`, followed by a newline. If an LSLIB call failed due to a system error, then the system error message as indicated by `errno` is included in the error message.

SYNOPSIS

```
#include <lsf/lsf.h>
void ls_perror(char *usrMsg)
```

PARAMETERS

***usrMsg** The standard output error string.

RETURN VALUES

void: There is no return value.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_sysmsg()`

`ls_errlog()`

Equivalent line command

none

Files

`lsf/lsf.h`

ls_placeofhosts()

Returns the most suitable `host()` for the `task()` from a set of candidate hosts with regards to current load conditions and the task's resource requirements.

DESCRIPTION

`ls_placeofhosts()` sends a task placement request to the LIM. The LIM returns a set of most suitable `host()` for the `task()`, taking into account the current load conditions and the task's resource requirements. Hostnames may be duplicated for hosts that have sufficient resources to accept multiple tasks (for example, multiprocessors).

If `ls_placeofhosts()` is successful, an array of host names is returned and `*num` is set to reflect the number of returned hosts. Otherwise, `ls_placeofhosts()` returns `NULL` and sets `lserrno` to indicate the error.

The routine returns a pointer to a dynamically allocated array of strings which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
```

```
char **ls_placeofhosts(char *resreq, int *num, int options,
    char *fromhost, char **hostlist, int listsize)
```

PARAMETERS

***resreq** `resreq` is a resource requirement expression that characterizes the resource needs of a single task. You can retrieve this parameter by calling `ls_eligible()` or the application can supply its own. See *Administering Platform LSF* for more information about resource requirement expressions. The names used for resource requirements are defined by the LSF administrator in the configuration file `LSF_CONFDIR/lsf.shared`. You can obtain the available resource names by calling `ls_info()` or running the LSF utility program `lsinfo()`. If `resreq` is `NULL`, then the default is assumed, which is to require a host of the same type as the local host with low 1-minute average CPU queue length and paging rate.

***num** `*num` is the number of hosts requested. If `*num` is zero, then all eligible hosts are requested. If `*num` is `NULL`, then a single host is requested.

options `options` is constructed from the bitwise inclusive OR of zero or more of the flags that are defined in `<lsf/lsf.h>`. These flags are documented in `ls_load()`.

***fromhost** `fromhost` is the host from which the task originates when LIM makes the placement decision. Preference is given to `fromhost` over remote hosts that do not have significantly lighter loads or greater resources. This preference avoids unnecessary task transfer and reduces overhead. If `fromhost` is `NULL`, then the local host is assumed.

****hostlist** Specifies a list of candidate hosts from which `ls_placeofhosts()` can choose suitable hosts.

listsize `listsize` gives the number of host or cluster names in `hostlist`.

RETURN VALUES

character:Array	An array of host names is returned.
character:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_placereq()`

`ls_loadadj()`

`ls_eligible()`

`ls_info()` – Returns a pointer to an `lsInfo` structure

`ls_load()`

Equivalent line command

none

Files

none

ls_placereq()

Returns the most suitable `host()` for the `task()` with regards to current load conditions and the task's resource requirements.

DESCRIPTION

`ls_placereq()` sends a task placement request to the LIM. The LIM returns a set of most suitable `host()` for the `task()`, taking into account the current load conditions and the task's resource requirements. Hostnames may be duplicated for hosts that have sufficient resources to accept multiple tasks (for example, multiprocessors).

If `ls_placereq()` is successful, an array of host names is returned and `*num` is set to reflect the number of returned hosts. Otherwise, `NULL` is returned and `lserrno` is set to indicate the error.

The routine returns a pointer to a dynamically allocated array of strings which can be freed in subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
char **ls_placereq(char *resreq, int *num, int options,
                                char *fromhost)
```

PARAMETERS

***resreq** The input parameter `resreq` is a resource requirement expression that characterizes the resource needs of a single task. You can retrieve this parameter by calling `ls_eligible()` or the application can supply its own. See *Administering Platform LSF* for more information about resource requirement expressions. The names used for resource requirements are defined by the LSF administrator in the configuration file `LSF_CONFDIR/lsf.shared`. You can obtain the available resource names by calling `ls_info()` or running the LSF utility program `lsinfo()`. If `resreq` is `NULL`, then the default is assumed, which is to require a host of the same type as the local host with low 1-minute average CPU queue length and paging rate.

***num** The input parameter `*num` is the number of hosts requested. If **num* is zero, then all eligible hosts are requested. If `*num` is `NULL`, then a single host is requested.

options The input parameter `options` is constructed from the bitwise inclusive OR of zero or more of the flags that are defined in `<lsf/lsf.h>`. These flags are documented in `ls_load()`.

***fromhost** `fromhost` is the host from which the task originates when LIM makes the placement decision. Preference is given to `fromhost` over remote hosts that do not have significantly lighter loads or greater resources. This preference avoids unnecessary task transfer and reduces overhead. If `fromhost` is `NULL`, then the local host is assumed.

RETURN VALUES

character:Array An array of host names is returned.

ERRORS

character:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_placeofhosts()`

`ls_loadadj()`

`ls_eligible()`

`ls_info()` – Returns a pointer to an `lsInfo` structure.

`ls_load()`

Equivalent line command

none

Files

none

ls_rclose()

Performs a close operation on a file on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `close()` system calls.

`ls_rclose()` performs a close operation on an opened file that is referenced by `rfd`. The file has been previously opened using `ls_ropen()`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_rclose(int rfd)
```

PARAMETERS

rfd	References the file that is to be closed.

RETURN VALUES

integer:0	Response is the same from its UNIX counterpart.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rread()`

`ls_rwrite()`

`ls_rlseek()`

`ls_rstat()`

SEE ALSO

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_readconfenv()

Reads the LSF configuration parameters from the `*confPath/lsf.conf` file.

DESCRIPTION

`ls_readconfenv()` reads the LSF configuration parameters from the `*confPath/lsf.conf` file. If `confPath` is `NULL`, the LSF configurable parameters are read from the `${LSF_ENVDIR-/etc}/lsf.conf` file. See `lsf.conf`. The input `paramList` is an array of data structures that are defined in `<lsf/lsf.h>`. The `paramName` parameter in the `config_param` data structure should be the pointer to the configuration parameter name defined in the `*confPath/lsf.conf` or `/etc/lsf.conf` file if `confPath` is `NULL`. The `paramValue` parameter in the `config_param` data structure must initially contain `NULL` and is then modified to point to a result string if a matching paramName is found in the `lsf.conf` file. A typical data structure declaration is as follows:

```
struct config_param paramList[] =
{
#define LSF_CONFDIR  0
    {"LSF_CONFDIR", NULL},
#define LSF_LOGDIR   1
    {"LSF_LOGDIR", NULL},
#define LSF_SERVERDIR 2
    {"LSF_SERVERDIR", NULL},
#define LSF_RES_DEBUG 3
    {"LSF_RES_DEBUG", NULL},
#define LSF_NPARAMS  4
    {NULL, NULL},
};
```

By calling `ls_readconfenv(paramList, "/localpath/etc")`, it is possible to read in the required parameters and use the defined constants as indices for referencing the parameters when needed. If a certain parameter name is not found in the `lsf.conf` file, then its `paramValue` will remain `NULL` on return.

SYNOPSIS

```
#include <lsf/lsf.h>

int ls_readconfenv(struct config_param *paramList,
                               char *confPath)

struct config_param {
    char *paramName;
    char *paramValue;
};
```

PARAMETERS

***paramlist**	An array of data structures that are defined in `<lsf/lsf.h>`.
paramname	The pointer to the configuration parameter name defined in the `*confPath/lsf.conf` or `/etc/lsf.conf` file if `confPath` is `NULL`.
***paramValue**	The `config_param` data structure must initially contain `NULL` and is then modified to point to a result string if a matching paramName is found in the `lsf.conf` file.

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_readrexlog()

Reads the next record from the opened log file created by RES.

DESCRIPTION

`ls_readrexlog()` reads the next record from the opened log file created by RES (see `lsf.acct`). It returns a pointer to the `lsfAcctRec` structure.

Memory for the `lsfAcctRec` structure is statically allocated and will be overwritten by the next `ls_readrexlog()` call.

The meaning of the fields in the `lsfAcctRec` structure is:

pid The process ID of the task. If a task contains a tree of processes, the root process ID is logged.

username The login name of the user who issued the task.

exitStatus The exit status of the task (see `wait()` for details).

dispTime The start time of the task.

termTime The termination time of the task.

fromHost The name of the host from which the task was submitted.

execHost The name of the host on which the task was executed.

cwd The current working directory of the task.

cmdln The task command line.

lsfRu Resource usage statistics. The `lsfRusage` structure is defined in `<lsf/lsf.h>`. Note that the availability of certain fields depends on the platform on which the RES runs. The fields that do not make sense on the platform will be logged as -1.0.

SYNOPSIS

```
#include <stdio.h>
#include <lsf/lsf.h>

struct lsfAcctRec *ls_readrexlog(FILE *fp)

struct lsfAcctRec {
    int pid;
    char *username;
    int exitStatus;
    time_t dispTime;
    time_t termTime;
    char *fromHost;
    char *execHost;
    char *cwd;
    char *cmdln;
    struct lsfRusage lsfRu;
}
```

PARAMETERS

***fp**

RETURN VALUES

FILE:1	Function was successful.
integer:0	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

none

ls_rescontrol()

Controls and maintains the Remote Execution Server.

DESCRIPTION

This library routine is used by the LSF administrator or authorized users to control and maintain the Remote Execution Server (RES).

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_rescontrol(char *host, int opCode, int data)
```

PARAMETERS

***host** The `host` argument is used to specify the host name of the machine whose RES is to be operated upon.

opCode The command is specified by the `opCode` argument and the `data` argument is used to supply an extra parameter for a particular `opCode`. The supported values are:

RES_CMD_REBOOT

Restart the RES. If the RES is in service, it will keep serving until all remote tasks exit, meanwhile starting another RES to serve new clients.

RES_CMD_SHUTDOWN

Shutdown the RES. The RES will not accept new tasks and will die after all current remote tasks exit.

RES_CMD_LOGON

Enable task logging, so that resource usage information can be logged to a file (see `lsf.acct`).

RES_CMD_LOGOFF

Disable task logging.

data The `data` argument is optionally used with `RES_CMD_LOGON` to specify a CPU time threshold in msec, so that RES will log resource information only for tasks that consumed more than the specified CPU time.

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, then `lserrno` is set to indicate the error. In particular, `ls_rescontrol()` will set `lserrno` to `LSE_BAD_OPCODE` if the `opCode` is not from the list above.

LSLIB calls for remote execution services

These routines allow programs to make use of LSF remote execution services. Such services include support for maintaining standard I/O transparency to and from remote machines, establishing, using, and terminating remote connections, transferring terminal and environment variable settings to remote processes, executing remote tasks and so on.

All rex routines require that the header `<lsf/lsf.h>` is included.

The following routines are supported:

ls_initrex()	Initiate remote execution
ls_connect()	Establish a remote connection
ls_isconnected()	Check for an established connection
ls_findmyconnections()	List hosts with open remote connections
ls_rexecv()	Remote execv
ls_rexecve()	Remote execve
ls_rtask()	Start a remote task
ls_rtaske()	Start a remote task with a new environment
ls_stdinmode()	Assign stdin to local or remote tasks
ls_getstdin()	List the remote task IDs that receive (or do not receive) standard input
ls_setstdin()	Specify how stdin is assigned to remote tasks.
ls_rwait()	Wait for a remote task to exit
ls_chdir()	Change the remote current working directory
ls_rsetenv()	Set environment on remote host
ls_rkill()	Kill a remote task

ls_donerex()

Restore terminal settings after remote execution

ls_fdbusy()

Test if a specified file descriptor is in use or reserved by LSF

ls_stoprex()

Stop the network I/O server

ls_conntaskport()

Connect to the remote task port.

LIMITATIONS

Although the level of transparency for remote execution in LSF is high, minor parts of the UNIX execution environment are not propagated to remote hosts. One such example is the UNIX process group.

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

none

ls_resreq()

Searches a remote task list for a task name and returns the resource requirements.

DESCRIPTION

LSLIB library routine for manipulating task lists stored by LSLIB. Task lists contain information about the eligibility of tasks for remote execution and their resource requirement characteristics. LSLIB maintains two task lists: local and remote. The local list contains tasks (i.e. UNIX processes) that must be executed on the local host (for example, `ps`, `uptime`, `hostname`). The remote list contains tasks that are suitable for remote execution (for example, `compress`), together with their resource requirements.

Task lists are generated and stored in memory by reading the system task `file()` and the `.lsftask` file in the user's home directory. The task lists can be updated and displayed using the command `lsrtasks()`. See the *LSF User's Guide* for detailed information on the use of task lists and resource requirements.

`ls_resreq()` is a simplified version of `ls_eligible()` which searches the remote task list for `taskname` and returns the resource requirements associated with the task if found otherwise NULL is returned.

`ls_resreq()` returns a pointer to static data which can be overwritten by subsequent calls.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_resreq(char *taskname)
```

PARAMETERS

***taskname**	The name of the task being sought.

RETURN VALUES

character:Requirements

Returns the resource requirements associated with the taskname.

character:NULL

Unable to find the taskname on the remote task list.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_eligible()`

`ls_listrtask()`

`ls_listltask()`

`ls_insertrtask()`

`ls_inesrtltask()`

`ls_deletertask()`

`ls_deleteltask()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.task`

`$LSF_CONFDIR/lsf.task.`*cluster_name*

`$HOME/.lsftask`

ls_rexecv()

Executes a program on a specified remote host.

DESCRIPTION

This routine is for executing remote tasks. It is modeled after the UNIX `fork` and `execv` system calls.

`ls_rexecv()` executes a program on the specified remote host. The program name is given in `argv[0]` and the arguments are listed in `argv`. This routine is basically a remote `execv`. If a connection with the Remote Execution Server (RES) on `host` has not been set up previously, `ls_connect()` is invoked to automatically establish the connection. The remote execution environment is set up to be exactly the same as the local one and is cached by the remote RES server.

The `options` value is constructed by ORing flags from the following list:

REXF_USEPTY	Use a remote pseudo-terminal.
REXF_CLNTDIR	Use the local client's current working directory as the current working directory for remote execution (see `ls_chdir()`).
REXF_TASKPORT	Request the remote RES to create a task port and return its number to the LSLIB. The application program can later call `ls_conntaskport()` to connect to the port.
REXF_SHMODE	Enable shell mode support if the `REXF_USEPTY` flag is also given. This flag is ignored if `REXF_USEPTY` is not given. This flag should be specified for submitting interactive shells, or applications which redefine the ctrl-C and ctrl-Z keys (e.g., `jove`).

The caller of this routine is typically a child process which terminates when the remote task is over. This routine does not return if successful. It returns -1 on failure.

Any program using this routine must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

The remote file operations documented in `ls_rfs()` make use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_rexecv(char *host, char **argv, int options)
```

PARAMETERS

***host**	The remote host where the program is executed.
****argv**	The program being used.
options	

RETURN VALUES

None	Function was successful.
integer:-1	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_rexecve()`

`ls_rtask()`

`ls_rtaske()`

`ls_control()`

`ls_chdir()`

`ls_conntaskport()`

`ls_initrex()`

`ls_rfs()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rexecve()

Executes a program on a specified remote host.

DESCRIPTION

This routine is for executing remote tasks. It is modeled after the UNIX `fork` and `execv` system calls.

`ls_rexecve()` executes a program on the specified remote host. The program name is given in `argv[0]` and the arguments are listed in `argv`. This routine is basically a remote `execv`. If a connection with the Remote Execution Server (RES) on `host` has not been set up previously, `ls_connect()` is invoked to automatically establish the connection. The remote execution environment is set up to be exactly the same as the local one and is cached by the remote RES server.

`ls_rexecve()` is the same as `ls_rexecv()` except that it provides the support of setting up a new environment specified by the string array `**envp`. When `envp` is a `NULL` pointer, it means using the remote RES server's cached environment, otherwise using the new one. A minimal default environment (`HOME`, `SHELL`, `USER`, and `PATH`) is initially cached when a remote execution connection is established and the cached environment is updated whenever the remote execution environment is changed by `ls_rsetenv()` or any of the routines on this man page.

The caller of this routine is typically a child process which terminates when the remote task is over. This routine does not return if successful. It returns -1 on failure.

Any program using this routine must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

The remote file operations documented in `ls_rfs()` make use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <lsf/lsf.h>
```

```
int ls_rexecve(char *host, char **argv, int options,
                           char **envp)
```

PARAMETERS

***host** The remote host where the program is executed.

****argv** The program being used.

options The `options` value is constructed by ORing flags from the following list:

REXF_USEPTY

Use a remote pseudo-terminal.

REXF_CLNTDIR

Use the local client's current working directory as the current working directory for remote execution (see `ls_chdir()`).

REXF_TASKPORT

Request the remote RES to create a task port and return its number to the LSLIB. The application program can later call `ls_conntaskport()` to connect to the port.

REXF_SHMODE

Enable shell mode support if the `REXF_USEPTY` flag is also given. This flag is ignored if `REXF_USEPTY` is not given. This flag should be specified for submitting interactive shells, or applications which redefine the ctrl-C and ctrl-Z keys (e.g., `jove`).

****envp**

RETURN VALUES

None	Function was successful.
integer:-1	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_rexecv()`

`ls_rtask()`

`ls_rtaske()`

`ls_control()`

`ls_chdir()`

`ls_conntaskport()`

`ls_rstenv()`

`ls_initrex()`

`ls_rfs()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rfcontrol()

Controls the behavior of remote file operations.

DESCRIPTION

This routine performs operations on files located on remote hosts.

`ls_rfcontrol()` controls the behavior of remote file operations. Possible commands are:

RF_CMD_MAXHOSTS Allows the caller to specify the number of connected hosts. When a remote file operation is being serviced at a host for the first time, a connection is made to the Remote Execution Server's (RES) file server process on the remote host. If the number of connections reaches `RF_MAXHOSTS` defined in `lsf.h`, the least recently used connection which does not have any files open on that host is broken. The `RF_CMD_MAXHOSTS` command allows you to change the maximum number of connections. The new maximum is specified in arg.

RF_CMD_TERMINATE Terminates the connection with the RES's file server process on host `arg`. `arg` is a pointer to the remote host name cast to `int`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` must be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_rfcontrol(int command, int arg)
```

PARAMETERS

command

arg A pointer to the remote host name cast to `int`.

RETURN VALUES

integer:0 The function was successful.

integer:-1 Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rread()`

`ls_rwrite()`

`ls_rlseek()`

`ls_rclose()`

`ls_rstat()`

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_initrex()`

Equivalent line command

none

Files

none:

ls_rfstat()

Obtains information about a file located on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `fstat()` system calls.

`ls_rfstat()` obtains information about a file located on the remote host. Because different platforms have different fields in the `stat` structure, only the following fields are updated: `st_dev`, `st_ino`, `st_mode`, `st_nlink`, `st_uid`, `st_gid`, `st_rdev`, `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_rfstat(int rfd, struct stat *buf)
```

PARAMETERS

rfd	References the file that is to be accessed.
***buf**	

RETURN VALUES

Response is the same from its UNIX counterpart.

integer:-1 Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rread()`

`ls_rwrite()`

`ls_rlseek()`

`ls_rclose()`

`ls_rstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_rgetmnthost()

Obtains the name of the file server that exports a specified file system.

DESCRIPTION

This routine performs operations on files located on remote hosts.

`ls_rgetmnthost()` obtains the name of the file server that exports the file system mounted on `host` that contains `file`, where file is a relative or absolute path name. If `host` is `NULL`, the local host name is assumed. This call corresponds to the `ls_getmnthost()` call.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
char *ls_rgetmnthost(char *host, char *file)
```

PARAMETERS

***host**	The host containing the `file`.
***file**	The file to be accessed.

RETURN VALUES

character:Hostname	The function was successful.
character:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rread()`

`ls_rwrite()`

`ls_rlseek()`

`ls_rclose()`

`ls_rstat()`

`ls_rfstat()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_rkill()

Signals a remote task

DESCRIPTION

`ls_rkill()` sends the signal `sig` to the remote task `tid` and all its children that belong to the same UNIX process group. `tid` is the remote task ID that is returned by `ls_rtask()` or `ls_rtaske()`. If `sig` is zero, then this call only polls the existence of the remote task `tid`.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
#include <signal.h>
```

```
int ls_rkill(int tid, int sig)
```

PARAMETERS

sig	The signal sent to the `tid`.
tid	The remote task ID returned by `ls_rtask()` or `ls_rtaske()`.

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_rtask()`

`ls_rtaske()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rlseek()

Performs a seek operation on a file on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `lseek()` system calls.

`ls_rlseek()` performs a read operation on an opened file that is referenced by `rfd`. The file has been previously opened using `ls_ropen()`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
off_t ls_rlseek(int rfd, off_t offset, int whence)
```

PARAMETERS

rfd	References the file that is to be sought.
offset	
whence	

RETURN VALUES

Response is the same from its UNIX counterpart.

integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rread()`

`ls_rwrite()`

SEE ALSO

`ls_rclose()`

`ls_rstat()`

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_ropen()

Opens a file on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `open()` system calls.

`ls_ropen()` opens the named file located on the remote `host`. A remote file descriptor, `rfd`, is returned on success. You can use this descriptor in subsequent remote file calls as an argument.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_ropen(char *host, char *path, int flags, int mode)
```

PARAMETERS

***host**	The host where the file to be opened is located.
***path**	The path name to the file to be opened.
flags	
mode	

RETURN VALUES

Response is the same from its UNIX counterpart.

integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_rread()`

SEE ALSO

`ls_rwrite()`

`ls_rlseek()`

`ls_rclose()`

`ls_rstat()`

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_rread()

Performs a read operation on a file on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `read()` system calls.

`ls_rread()` performs a read operation on an opened file that is referenced by `rfd`. The file has been previously opened using `ls_ropen()`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_rread(int rfd, char *buf, int len)
```

PARAMETERS

rfd References the file that is to be read.

***buf**

len

RETURN VALUES

Response is the same from its UNIX counterpart.

integer:-1 Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rwrite()`

`ls_rlseek()`

SEE ALSO

`ls_rclose()`

`ls_rstat()`

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_rstat()

Obtains information about a file located on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `stat()` system calls.

`ls_rstat()` obtains information about a file located on the remote host. Because different platforms have different fields in the `stat` structure, only the following fields are updated: `st_dev`, `st_ino`, `st_mode`, `st_nlink`, `st_uid`, `st_gid`, `st_rdev`, `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_rstat(char *host, char *path, struct stat *buf)
```

PARAMETERS

***host**	The remote host containing the file to be analyzed.
***path**	
***buf**	

RETURN VALUES

Response is the same from its UNIX counterpart.

integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

SEE ALSO

`ls_rread()`

`ls_rwrite()`

`ls_rlseek()`

`ls_rclose()`

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_rsetenv()

Sets up environment variables on a remote host.

DESCRIPTION

`ls_rsetenv()` sets up the environment variables given in envp on the specified remote host. `envp` is a pointer to an array of strings of the form `variable=value`. When the environment variables are set, all remote tasks on the remote host acquire the environment setting until another call to this routine overrides it. A default set of environment variables is set up for the remote host if this routine is never called (see `ls_rtask()`). This call is typically used to propagate changes in the local environment to the remote hosts to which the application has connections.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_rsetenv(char *host, char **envp)
```

PARAMETERS

***host**	The remote host upon which the environment is being set.
****envp**	A pointer to an array of strings of the form `variable=value`.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_rtask()`

`ls_initrex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rtask()

Starts a remote task on a specified host.

DESCRIPTION

This routine is for executing remote tasks. It is modeled after the UNIX `fork` and `execv` system calls.

`ls_rtask()` starts a remote task on the specified host. This routine is basically a remote `fork` followed by an `execv`. The arguments are identical to those of `ls_rexecv()`. `ls_rtask()` is typically used by a parallel application to execute multiple remote tasks efficiently. When a remote task finishes, a SIGUSR1 signal is delivered back to the application, and its status can be collected by calling `ls_rwait()` or `ls_rwaittid()`. `ls_rtask()` returns a unique task ID to be used by the application to differentiate outstanding remote tasks. It returns -1 on failure.

Any program using this routine must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

The remote file operations documented in `ls_rfs()` make use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_rtask(char *host, char **argv, int options)
```

PARAMETERS

***host**	The remote host where the program is executed.
****argv**	The program being used.
options	

RETURN VALUES

integer:Unique TaskID	Function was successful.
integer:-1	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_rexecv()`

`ls_rexecve()`

`ls_rtaske()`

`ls_control()`

`ls_chdir()`

`ls_conntaskport()`

`ls_rstenv()`

`ls_initrex()`

`ls_rfs()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rtaske()

Starts a remote task on a specified host.

DESCRIPTION

This routine is for executing remote tasks. It is modeled after the UNIX `fork` and `execv` system calls.

`ls_rtaske()` starts a remote task on the specified host. This routine is basically a remote `fork` followed by an `execv`. The arguments are identical to those of `ls_rexecv()`. `ls_rtask()` is typically used by a parallel application to execute multiple remote tasks efficiently. When a remote task finishes, a SIGUSR1 signal is delivered back to the application, and its status can be collected by calling `ls_rwait()` or `ls_rwaittid()`. `ls_rtask()` returns a unique task ID to be used by the application to differentiate outstanding remote tasks. It returns -1 on failure.

`ls_rtaske()` is the same as `ls_rtask()` except that it provides the support of setting up a new environment specified by the string array `**envp`. When `envp` is a `NULL` pointer, it means using the remote RES server's cached environment, otherwise using the new one. A minimal default environment (`HOME`, `SHELL`, `USER`, and `PATH`) is initially cached when a remote execution connection is established and the cached environment is updated whenever the remote execution environment is changed by `ls_rsetenv()` or any of the routines on this man page.

Any program using this routine must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

The remote file operations documented in `ls_rfs()` make use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_rtaske(char *host, char **argv, int options,
                     char **envp)
```

PARAMETERS

***host**	The remote host where the program is executed.
****argv**	The program being used.
options	
***envp**	

RETURN VALUES

integer:Unique TaskID	Function was successful.
integer:-1	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_rexecv()`

`ls_rexecve()`

`ls_rtask()`

`ls_control()`

`ls_chdir()`

`ls_conntaskport()`

`ls_rstenv()`

`ls_initrex()`

`ls_rfs()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rwait()

Collects the status of a remote task started by `ls_rtask()` or `ls_rtaske()`.

DESCRIPTION

`ls_rwait()` collects the status of a remote child (task) that has been started by `ls_rtask()` or `ls_rtaske()`. This call is similar to the UNIX `wait3()` call, except that the child is located on a remote host.

If a remote child's status is successfully obtained, then the remote task ID (which is returned by an earlier `ls_rtask()` or `ls_rtaske()` call) is returned. Also, if `status` is not `NULL`, the status of the exited child is stored in the structure pointed to by `status`. If `ru` is not `NULL`, and the remote child's machine supports the `rusage` structure in its `wait3()` call, the resource usage information of the exited child is stored in the structure pointed to by `ru`. Only the `ru_utime` and `ru_stime` fields are set in the structure if the remote child's machine does not support the `rusage` structure in the `wait3()` call. If the remote child is run on a different platform than the parent, then only the fields in the resource structure that are common between the two platforms are filled in (the `rusage` structure is not identical across all platforms). If the child runs on a 64-bit machine, and the parent runs on a 32-bit machine, each of the values in the `rusage` structure that will overflow on a 32-bit machine are set to `LONG_MAX`.

The `ls_rwait()` call are automatically restarted when the parent receives a signal while awaiting termination of a remote child process, unless the `SV_INTERRUPT` bit has been set for the signal (see `sigaction()`).

LSLIB defines some new return status values related to load sharing. These values are returned by `ls_rwait()`. They include:

STATUS_TIMEOUT	Timeout trying to connect to the remote RES.
STATUS_IOERR	The remote task failed with an I/O error.
STATUS_EXCESS	Too many tasks are currently executing.
STATUS_REX_NOMEM	RES failed to allocate memory
STATUS_REX_FATAL	Fatal error, check RES err log
STATUS_REX_CWD	Cannot change to current working directory
STATUS_REX_PTY	RES cannot allocate a pty
STATUS_REX_SP	RES cannot allocate a socket pair
STATUS_REX_FORK	RES failed to fork the task
STATUS_REX_UNKNOWN	Internal error in RES

Use the blocking mode of `ls_rwait()` with care. If there are both local and remote children, `ls_rwait()` take care only of remote children; none of them will return even though a local child has exited. In such cases, you can call `wait()`, `ls_rwait()` and/or `ls_rwaittid()` via signal handlers (for `SIGCHLD` and `SIGUSR1`, respectively) to process local and remote children.

When a remote child terminates, SIGUSR1 is sent to the parent process. Thus, `ls_rwait()` is typically called from inside the SIGUSR1 signal handler of the parent process.

Any program using these routines must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

The remote file operations documented in `ls_rfs()` make use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <lsf/lsf.h>
```

```
int ls_rwait(LS_WAIT_T *status, int options, struct rusage *ru)
```

PARAMETERS

***status**

options If specified as 0, and there is at least one remote child, then the calling host is blocked until a remote child exits. If `options` is specified to be `WNOHANG`, the routine checks for any exited (remote) child and returns immediately. The `options` parameter may be extended to provide more options in the future.

***ru** The structure where the resource usage information of the exited child is stored.

RETURN VALUES

character:remote task ID

The function was successful.

integer:-1

Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_rwaittid()`

`ls_rtask()`

`ls_rtaske()`

`ls_rfs()`

SEE ALSO

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rwaittid()

Provides support for collecting the status of a specified remote task.

DESCRIPTION

`ls_rwaittid()` is modelled after the UNIX `waitpid()` system call. It provides support for collecting the status of the remote task whose task ID is `tid`.

If a remote child's status is successfully obtained, then the remote task ID (which is returned by an earlier `ls_rtask()` or `ls_rtaske()` call) is returned. Also, if `status` is not `NULL`, the status of the exited child is stored in the structure pointed to by `status`. If `ru` is not `NULL`, and the remote child's machine supports the `rusage` structure in its `wait3()` call, the resource usage information of the exited child is stored in the structure pointed to by `ru`. Only the `ru_utime` and `ru_stime` fields are set in the structure if the remote child's machine does not support the `rusage` structure in the `wait3()` call. If the remote child is run on a different platform than the parent, then only the fields in the resource structure that are common between the two platforms are filled in (the `rusage` structure is not identical across all platforms). If the child runs on a 64-bit machine, and the parent runs on a 32-bit machine, each of the values in the `rusage` structure that will overflow on a 32-bit machine are set to `LONG_MAX`.

`ls_rwaittid()` behaves identically to `ls_rwait()` if `tid` has a value of zero. If `tid` is less than 0, it returns -1 and sets `lserrno` to `LSE_BAD_ARGS`.

The `ls_rwaittid()` call are automatically restarted when the parent receives a signal while awaiting termination of a remote child process, unless the `SV_INTERRUPT` bit has been set for the signal (see `sigaction()`).

LSLIB defines some new return status values related to load sharing. These values are returned by `ls_rwaitidt()`. They include:

STATUS_TIMEOUT	Timeout trying to connect to the remote RES.
STATUS_IOERR	The remote task failed with an I/O error.
STATUS_EXCESS	Too many tasks are currently executing.
STATUS_REX_NOMEM	RES failed to allocate memory
STATUS_REX_FATAL	Fatal error, check RES err log
STATUS_REX_CWD	Cannot change to current working directory
STATUS_REX_PTY	RES cannot allocate a pty
STATUS_REX_SP	RES cannot allocate a socket pair
STATUS_REX_FORK	RES failed to fork the task
STATUS_REX_UNKNOWN	Internal error in RES

Use the blocking mode of `ls_rwaittid()` with care. If there are both local and remote children, `ls_rwaittid()` take care only of remote children; none of them will return even though a local child has exited. In such cases, you can call `wait()`, `ls_rwait()` and/or `ls_rwaittid()` via signal handlers (for `SIGCHLD` and `SIGUSR1`, respectively) to process local and remote children.

Any program using these routines must call `ls_initrex()` first.

Any program using these routines must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

The remote file operations documented in `ls_rfs()` make use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <lsf/lsf.h>
```

```
int ls_rwaittid(int tid, LS_WAIT_T *status, int options,
    struct rusage *ru)
```

PARAMETERS

tid The ID of the remote task being accessed.

***status**

options If `options` is set to 0, and there is at least one remote task, the calling host is blocked until the specific task exits. If `options` is `WNOHANG` (non-blocking), it reads the child's status if the child is dead, otherwise it returns immediately with 0. If the status of the child is successfully read, the remote task ID is returned.

ru The structure where the resource usage information of the exited child is stored.

RETURN VALUES

character:remote task ID

The function was successful.

integer:-1

Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error. If `tid` is less than 0, it returns -1 and sets `lserrno` to `LSE_BAD_ARGS`.

SEE ALSO

Related APIs

`ls_rwait()`

`ls_rtask()`

`ls_rtaske()`

`ls_rfs()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_rwrite()

Performs a write operation on a file on a remote host.

DESCRIPTION

This routine performs operations on files located on remote hosts. This call corresponds to the UNIX `write()` system calls.

`ls_rwrite()` performs a read operation on an opened file that is referenced by `rfd`. The file has been previously opened using `ls_ropen()`.

Either the RES must be running at the remote host to service any remote file operation or `rcp()` be available.

`ls_initrex()` must be called before calling any remote file operation.

This remote file operation makes use of a Remote File Server on the remote host. When this RFS shuts down, its status will be reported to its client. The client should ignore this status.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <lsf/lsf.h>
```

```
int ls_rwrite(int rfd, char *buf, int len)
```

PARAMETERS

rfd	References the file that is to be written.
***buf**	
len	

RETURN VALUES

Response is the same from its UNIX counterpart.

integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_ropen()`

`ls_rread()`

`ls_rlseek()`

`ls_rclose()`

`ls_rstat()`

`ls_rfstat()`

`ls_rgetmnthost()`

`ls_rfcontrol()`

`ls_initrex()`

Equivalent line command

none

Files

none

ls_setstdin()

Allows an application program to query and specify how stdin is assigned to a specific subset of remote tasks.

DESCRIPTION

`ls_setstdin()` gives an application program the ability to query and specify how stdin is assigned to remote tasks. It allows you to assign stdin to a specific subset of remote tasks. You can change this setting at any time.

`ls_setstdin()` turns on or off the delivery of standard input to specific remote tasks. Other remote tasks are not affected by this call.

By default, a remote task is set to receive standard input. Note that remote tasks only receive standard input if the current stdin mode is remote. Hence, if the application is running in local stdin mode (see the description of `ls_stdinmode()`), `ls_setstdin()` is not effective.

Upon success, `ls_setstdin()` returns zero. On failure, -1 is returned, and the error code is stored in `lserrno`.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_setstdin(int on, int *tidlist, int len)
```

PARAMETERS

on If `on` is non-zero and the current stdin mode is remote, then the tasks given by `tidlist` receive the standard input. If `on` is zero, the tasks will not receive standard input.

***tidlist** `tidlist` gives the list of task IDs of the remote tasks to be operated upon.

len The number of entries.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_stdinmodel()`

`ls_getstdin()`

`ls_initrex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_sharedresourceinfo()

Returns shared resource information in dynamic values.

DESCRIPTION

`ls_sharedresourceinfo()` returns the requested shared resource information in dynamic values. The result of this call is a chained data structure as defined in `<lsf/lsf.h>`, which contains requested information.

SYNOPSIS

```
#include <lsf/lsf.h>
LS_SHARED_RESOURCE_INFO_T *ls_sharedresourceinfo(
                           char **resources,
                           int *numResources, char *hostName,
                           int options)

typedef struct lsSharedResourceInfo {
/* resource name */
    char *resourceName;
/* number of instances */
    int nInstances;
/* pointer to the next instance */
    LS_SHARED_RESOURCE_INST_T *instances;
} LS_SHARED_RESOURCE_INFO_T;

typedef struct lsSharedResourceInstance {
/* Value associated with the resource */
    char *value;
    int nHosts;
/* Hosts associated with the resource. */
    char **hostList;
} LS_SHARED_RESOURCE_INST_T;
```

PARAMETERS

****resources**	`resources` is an array of `NULL` terminated strings storing requesting resource names. If set to `NULL`, the call returns all shared resources defined in the cluster.
***numResources**	`numResources` is an input/output parameter. On input it indicates how many resources are requested. Value 0 means requesting all shared resources. On return it contains qualified number of resources.
***hostName**	`hostName` is a string containing a host name. Only shared resource available on the specified host will be returned. If `hostName` is set to NULL, shared resource available on all hosts will be returned.
options	`options` is reserved for future use. Currently, it should be set to 0.

RETURN VALUES

pointer:array	The function was successful.
character:NULL	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.cluster_name`

ls_stdinmode()

Allows an application program to query and specify how stdin is assigned to remote tasks on a local application.

DESCRIPTION

`ls_stdinmode()` gives an application program the ability to query and specify how stdin is assigned to remote tasks. It allows you to assign stdin to the local program only. You can change this setting at any time.

`ls_stdinmode()` specifies whether standard input is read by the calling (local) application or its remote children.

This routine returns 0 on success; otherwise, it returns -1 and sets `lserrno` to indicate the error.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_stdinmode(int remote)
```

PARAMETERS

remote If `remote` is non-zero, then the application will not read subsequent standard input, and the remote children will read standard input. This mode of operation is called the remote stdin mode. Remote stdin mode is the default. In remote stdin mode, standard input is read by the Network I/O Server (NIOS) and forwarded to the appropriate remote tasks. If `remote` is zero, then the application reads the subsequent standard input, and it is not forwarded to remote children. This mode of operation is called the local stdin mode.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_getstdin()`

`ls_setstdin()`

`ls_initrex()`

Equivalent line command

none

Files

```
${LSF_ENVDIR-/etc}/lsf.conf
```

ls_stoprex()

Stops the Networks I/O Server and restores the local tty environment.

DESCRIPTION

`ls_stoprex()` stops the Network I/O Server (NIOS) and restores the local tty environment. This routine is necessary only for those LSF applications that explicitly catch job control signals (that is, SIGTSTP) and eventually suspend themselves. For most applications, the default SIGTSTP handler in the LSF library provides the desired behavior, without the requirement of calling this routine.

When NIOS receives a SIGTSTP signal, it sends the signal to all its remote tasks. If the local client also needs to be stopped, then `ls_stoprex()` must be called to stop the NIOS and restore the tty environment for the parent application first.

Any program using this routine must call `ls_initrex()` first.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_stoprex(void)
```

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`ls_initrex()`

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

ls_sysmsg()

Obtains LSF error messages.

DESCRIPTION

`ls_sysmsg()` is a LSLIB library routine for obtaining LSF error messages. The global variable `lserrno`, maintained by LSLIB, indicates the error number of the most recent LSLIB call that caused an error.

`ls_sysmsg()` returns a character string that contains an error message that corresponds to the current value of `lserrno`. If an LSLIB call failed due to a system call, then the system error message as indicated by `errno` is included in the error message.

SYNOPSIS

```
#include <lsf/lsf.h>
char *ls_sysmsg(void)
```

RETURN VALUES

char:	Function was successful.
char:NULL	Function failed.

ERRORS

Systems that conform to the Single UNIX specification are not required to detect error conditions for this function. – Error handling

SEE ALSO

Related APIs

`ls_perror()`

`ls_errlog()`

Equivalent line command

none

Files

`lsf/lsf.h`

ls_unlockhost()

Unlocks a locked local host.

DESCRIPTION

`ls_unlockhost()` unlocks a host locked by `ls_lockhost()`. On success, `ls_unlockhost()` changes the status of the local host to indicate that it is no longer locked by the user.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsf.h>
int ls_unlockhost(void)
```

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error. In particular, `ls_unlockhost()` sets lserrno to `LSE_LIM_NLOCKED` if the host is not locked.

SEE ALSO

Related APIs:

`ls_limcontrol()` - shuts down or reboots a host's LIM

`ls_lockhost()` - locks a host

Equivalent line command

```
lsadmin limunlock
```

Files:

```
${LSF_ENVDIR-/etc}/lsf.conf
$LSF_CONFDIR/lsf.shared

$LSF_CONFDIR/lsf.cluster.cluster_name
```

lsb_addreservation()

Makes an advance reservation.

DESCRIPTION

Use `lsb_addreservation()` to send a reservation request to `mbatchd`. If `mbatchd` grants the reservation, it issues the reservation ID. If `mbatchd` rejects the request, it issues `NULL` as the reservation ID.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_addreservation (struct addRsvRequest *request char *rsvId)

struct addRsvRequest {
    int options;
    char *name;
    struct {
        int minNumProcs;
        int maxNumProcs;
    } procRange;
    int numAskedHosts;
    char **askedHosts;
    char *resReq;
    char *timeWindow;
    rsvExecCmd_t  *execCmd;
    char *desc;
    char *rsvName;
};
```

PARAMETERS

***request**	The reservation request
***rsvId**	Reservation ID returned from mbatchd. If the reservation fails, this is NULL. The memory for `rsvid` is allocated by the caller.

addRsvRequest structure

options	Reservation options.
name	LSF user group name for the reservation. See the `-g` option of `brsvadd`.
minNumProcs	Minimum number of processors the required to run the job. See the `-g` option of `brsvadd`.
maxNumProcessors	Maximum number of processors the required to run the job.
numAskedHosts	The number of invoker specified hosts for the reservation. If `numAskedHosts` is 0, all qualified hosts will be considered.

askedHosts	The array of names of invoker specified hosts hosts for the reservation. The number of hosts is given by `numAskedHosts`. See the `-m` option of `brsvadd`.
resReq	The resource requirements of the reservation. See the `-R` option of `brsvadd`.
timeWindow	Active time window for a recurring reservation. See the `-t` option of `brsvadd`.
rsvExecCmd_t *execCmd;	Info for the -exec option.
desc	description for the reservation to be created. The description must be provided as a double quoted text string. The maximum length is 512 characters. Equivalent to the value of `brsvadd -d`.
rsvName	User-defined advance reservation name unique in an LSF cluster. The name is a string of letters, numeric characters, underscores, and dashes beginning with a letter. The maximum length of the name is 39 characters. Equivalent to the value of `brsvadd -N`.

RETURN VALUES

integer:0	The reservation was successful.
integer:-1	The reservation failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_removereservation()` - Removes a reservation

`lsb_modreservation()` - Modifies a reservation

`lsb_reservationinfo()` - Retrieves reservation information

Equivalent line command

`brsvadd`

Files:

none

lsb_calendarinfo()

Gets information about calendars defined in the batch system.

DESCRIPTION

`lsb_calendarinfo()` gets information about calendars defined in the batch system.

On success, this routine returns a pointer to an array of `calendarInfoEnt` structures which stores the information about the returned calendars and `*numCalendars` gives number of calendars returned. On failure `NULL` is returned and `lsberrno` is set to indicate the error.

In `calendarInfoEnt` structure:

name	A pointer to the name of the calendar.
desc	A description string associated with the calendar
timeEvents	The time expression list used to generate the time events of the calendar (see `bcaladd()` for a description of time events and expressions.)
lastEvent	The time of the last event which occurred in the calendar.
lastDuration	The duration of the last time event.
nextEvent	The time of the next event which is expected to occur in the calendar.
nextDuration	The duration of the next time event.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct calendarInfoEnt *lsb_calendarinfo(char **calendars,
                         int *numCalendars, char *user)
struct calendarInfoEnt {
    char *name;
    char *desc;
    char *calExpr;
    char *userName;
    int status;
    int options;
    int lastDay;
    int nextDay;
    time_t creatTime;
    time_t lastModifyTime;
    intflags;
};
```

PARAMETERS

****calendars**	`calendars` is a pointer to an array of calendar names.

RETURN VALUES

***numCalendars**	`*numCalendars` gives the number of calendar names. If `*numCalendars` is 0, then information about all calendars is returned. By default, only the invokers calendars are considered.
***user**	Setting the `user` parameter will cause the given users calendars to be considered. Use the reserved user name `all` to get calendars of all users.

RETURN VALUES

character:POINTER	Sends an array about the calendars and their info .
character:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_calendarop()`

none

Equivalent line command

none

Files

none

lsb_calendarop()

Adds, modifies or deletes a calendar.

DESCRIPTION

`lsb_calendarop()` is used to add, modify or delete a calendar. The `oper` parameter is one of `CALADD`, `CALMOD`, or `CALDEL`. When the operation `CALADD` is specified, the first element of the `names` array is used as the name of the calendar to add. The `desc` and `timeEvents` parameters should point to the description string and the time expression list, respectively. See `bcaladd()` for a description of time expressions. `CALMOD` permits the modification of the description or time expression list associated with an existing calendar. The first name in the `names` array indicates the calendar to be modified. The `desc` and `timeEvents` parameters can be set to the updated value or to `NULL` to indicate that the existing value should be maintained. If the operation is `CALDEL` then the `names` parameter points to an array of calendar names to be deleted. `numNames` gives the number of names in the array. `options` is reserved for the future use.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int *lsb_calendarop(int oper, int numNames, char **names, char
*desc, char *timeEvents, int options, char **badStar)

#define CALADD    1
#define CALMOD    2
#define CALMOD    3
```

PARAMETERS

oper	One of `CALADD`, `CALMOD`, or `CALDEL`. Depending on which one is chosen, adds, modifies, or deletes a calendar.
***names**	Depending on oper, it defines the name of the calendar is going to be added, modified or deleted.
***desc**	The calendar's description list.
****timeEvents**	The calendar's time events list.
numNames	The number of names in the array.
options	Currently unused.
****badStar**	Need description

RETURN VALUES

character:POINTER	Sends an array about the calendars and their info .
character:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error. If error is related to bad calendar name or time expression, the routine returns the name or expression in badStr.

SEE ALSO

Related APIs

`lsb_calendarinfo()`

Equivalent line command

none

Files

none

lsb_chkpntjob()

Checkpoints a job.

DESCRIPTION

Checkpoints a job.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_chkpntjob(jobId, period, options)
```

PARAMETERS

LS_LONG_INT jobId	The job to be checkpointed.
time_t period;	The checkpoint period in seconds. The value 0 disables periodic checkpointing.
int options;	The bitwise inclusive OR of some of the following: **LSB_CHKPNT_KILL** Checkpoint and kill the job as an atomic action. **LSB_CHKPNT_FORCE** Checkpoint the job even if non-checkpointable conditions exist.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

NOTES

Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs:

none

Equivalent line command

`bchkpnt`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_closejobinfo()

Closes job information connection with the master batch daemon.

DESCRIPTION

Use `lsb_closejobinfo()` to close the connection to the master batch daemon after opening a job information connection with `lsb_openjobinfo()` and reading job records with `lsb_readjobinfo()`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
void lsb_closejobinfo(void)
```

RETURN VALUES

Returns void.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs:

lsb_openjobinfo() - Opens a connection to the master batch daemon

lsb_openjobinfo_a() - Provides the name and number of jobs and hosts in the master batch daemon

lsb_readjobinfo() - Returns the next job information record in master batch daemon

Equivalent line command

none

Files:

none

lsb_closestream()

Close an lsb_stream file.

DESCRIPTION

lsb_closestream() closes the streamFile.

This API function is inside `liblsbstream.so`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_closestream(const char *config)
```

PARAMETERS

*** config**	Pointer to the handle of the stream file.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

lsb_openstream(): Open the stream file.

lsb_readstreamline(): Read a line from the stream file.

lsb_writestream(): Write an event to the stream file.

lsb_readstream(): Read from the stream file.

lsb_streamversion(): Version of the current event type supported by mbatchd.

Equivalent line command

None

Files

`lsb.params`

lsb_deletejob()

Kills a job in a queue

DESCRIPTION

Use `lsb_deletejob()` to send a signal to kill a running, user-suspended, or system-suspended job. The job can be requeued or deleted from the batch system. If the job is requeued, it retains its submit time but it is dispatched according to its requeue time. When the job is requeued, it is assigned the `PEND` status and re-run. If the job is deleted from the batch system, it is no longer available to be requeued.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_deletejob (LS_LONG_INT jobId, int times, int options)
```

PARAMETERS

jobId	The job to be killed. If an element of a job array is to be killed, use the array form `jobID[i]` where `jobID` is the job array name, and `i` is the index value.
times	Original job submit time.
options	If the preprocessor macro `LSB_KILL_REQUEUE` in `lsbatch.h` is compared with options and found true, then requeue the job using the same job ID. If the preprocessor macro `LSB_KILL_REQUEUE` in `lsbatch.h` is compared with options and found false, then the job is deleted from the batch system.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

NOTES

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`lsb_signaljob()` - Signals a job

`lsb_chkpntjob()` - Checkpoints a job

Equivalent line command

`bkill`

SEE ALSO

`brequeue -J`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_freeLimitInfoEnt()

Frees the memory allocated by `lsb_limitInfo()`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
void lsb_freeLimitInfoEnt(limitInfoEnt * ent, int size)
typedef struct _limitInfoEnt {
char *         name;
limitItem   confInfo;
int              usageC;
limitItem   usageInfo;
} limitInfoEnt;
```

PARAMETERS

ent	input, the array of limit information
size	input, the size of the limit information array
_limitInfoEnt	The structure limitInfoEnt contains the following fields:

name

Limit policy name given by the user.

confInfo

Limit configuration.

usageC

Size of limit dynamic usage info array.

usageInfo

Limit dynamic usage info array.

RETURN VALUES

LSBE_NO_ERROR	Suceess; others, errors happened.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`lsb_limitInfo()`

Equivalent command

`blimits`

Files

`lsb.queues`, `lsb.users`, `lsb.hosts`, `lsb.resources`

lsb_getalloc()

Allocates memory for a host list to be used for launching parallel tasks through `blaunch` and the `lsb_launch()` API.

DESCRIPTION

It is the responsibility of the caller to free the host list when it is no longer needed. On success, the host list will be a list of strings. Before freeing host list, the individual elements must be freed.

An application using the `lsb_getalloc()` API is assumed to be part of an LSF job, and that LSB_MCPU_HOSTS is set in the environment.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_getalloc(char ***hostlist)
```

PARAMETERS

hostlist	[OUT] A null-terminated list of host names

RETURN VALUES

> 0	Function was successful. Returns length of `hostlist`, not including the null last element.
< 0	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

none

Equivalent line command

none

Files

none

lsb_geteventrec()

Get an event record from a log file

DESCRIPTION

`lsb_geteventrec()` returns an `eventRec` from a log file.

The storage for the `eventRec` structure returned by `lsb_geteventrec()` will be reused by the next call.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct eventRec *lsb_geteventrec(log_fp, lineNum)
FILE *log_fp;
int *lineNum;
struct eventRec {
    char  version[MAX_VERSION_LEN];
    int   type;
    time_t eventTime;
    union eventLog eventLog;
};
union  eventLog {
    struct jobNewLog jobNewLog;
    struct jobStartLog jobStartLog;
    struct jobStatusLog jobStatusLog;
    struct sbdJobStatusLog sbdJobStatusLog;
    struct jobSwitchLog jobSwitchLog;
    struct jobMoveLog jobMoveLog;
    struct queueCtrlLog queueCtrlLog;
    struct newDebugLog  newDebugLog;
    struct hostCtrlLog hostCtrlLog;
    struct mbdStartLog mbdStartLog;
    struct mbdDieLog mbdDieLog;
    struct unfulfillLog unfulfillLog;
    struct jobFinishLog jobFinishLog;
    struct loadIndexLog loadIndexLog;
    struct migLog migLog;
    struct calendarLog calendarLog;
    struct jobForwardLog jobForwardLog;
    struct jobAcceptLog jobAcceptLog;
    struct statusAckLog statusAckLog;
    struct signalLog signalLog;
    struct jobExecuteLog jobExecuteLog;
    struct jobMsgLog jobMsgLog;
```

SYNOPSIS

```
    struct jobMsgAckLog jobMsgAckLog;
    struct jobRequeueLog jobRequeueLog;
    struct chkpntLog chkpntLog;
    struct sigactLog sigactLog;
    struct jobOccupyReqLog jobOccupyReqLog;
    struct jobVacatedLog jobVacatedLog;
    struct jobStartAcceptLog jobStartAcceptLog;
    struct jobCleanLog jobCleanLog;
    struct jobExceptionLog jobExceptionLog;
    struct jgrpNewLog jgrpNewLog;
    struct jgrpCtrlLog jgrpCtrlLog;
    struct jobForceRequestLog jobForceRequestLog;
    struct logSwitchLog logSwitchLog;
    struct jobModLog jobModLog;
    struct jgrpStatusLog jgrpStatusLog;
    struct jobAttrSetLog jobAttrSetLog;
    struct jobExternalMsgLog jobExternalMsgLog;
    struct jobChunkLog jobChunkLog;
    struct sbdUnreportedStatusLog sbdUnreportedStatusLog
    struct rsvFinishLog rsvFinishLog;
    struct hgCtrlLog hgCtrlLog;
    struct cpuProfileLog cpuProfileLog;
    struct dataLoggingLog dataLoggingLog;
    struct jobRunRusageLog   jobRunRusageLog;
    struct eventEOSLog eventEOSLog;
    struct slaLog slaLog;
    struct perfmonLog perfmonLog;
    struct taskFinishLog taskFinishLog;
};
struct xFile {
    char *subFn;
    char *execFn;
    int options;
};
struct jobAttrSetLog {   /* Structure for log_jobattrset() and other calls */
    int       jobId;
    int       idx;
    int       uid;
    int       port;
    char      *hostname;
};
struct logSwitchLog { /* Records of logged events */
    int lastJobId;
```

```
};
struct dataLoggingLog {          /* Records of job cpu data logged event */
    time_t loggingTime;
};
struct jgrpNewLog {
    int    userId;
    time_t submitTime;
    char   userName[MAX_LSB_NAME_LEN];
    char   *depCond;
    char   *timeEvent;
    char   *groupSpec;
    char   *destSpec;
    int    delOptions;
    int    delOptions2;
    int    fromPlatform;
    char   *sla;
    int  maxJLimit;
    int options;
};
struct jgrpCtrlLog {
   int    userId;
   char   userName[MAX_LSB_NAME_LEN];
    char   *groupSpec;
    int    options;
    int    ctrlOp;
};
struct jgrpStatusLog {
    char   *groupSpec;
    int    status;
    int    oldStatus;
};
struct jobNewLog {         /* logged in lsb.events when a job is submitted */
    int    jobId;
    int    userId;
    char   userName[MAX_LSB_NAME_LEN];
    int    options;
    int    options2;
    int    numProcessors;
    time_t submitTime;
    time_t beginTime;
    time_t termTime;
    int    sigValue;
    int    chkpntPeriod;
```

SYNOPSIS

```
int    restartPid;
int    rLimits[LSF_RLIM_NLIMITS];
char   hostSpec[MAXHOSTNAMELEN];
float  hostFactor;
int    umask;
char   queue[MAX_LSB_NAME_LEN];
char   *resReq;
char   fromHost[MAXHOSTNAMELEN];
char   *cwd;
char   *chkpntDir;
char   *inFile;
char   *outFile;
char   *errFile;
char   *inFileSpool;
char   *commandSpool;
char   *jobSpoolDir;
char   *subHomeDir;
char   *jobFile;
int    numAskedHosts;
char   **askedHosts;
char   *dependCond;
char   *timeEvent;
char   *jobName;
char   *command;
int    nxf;
struct xFile *xf;
char   *preExecCmd;
char   *mailUser;
char   *projectName;
int    niosPort;
int    maxNumProcessors;
char   *schedHostType;
char   *loginShell;
char   *userGroup;
char   *exceptList;
int    idx;
int    userPriority;
char   *rsvId;
char   *jobGroup;
char   *extsched;
int    warningTimePeriod;
char   *warningAction;
char   *sla;
```

```
    int    SLArunLimit;
    char   *licenseProject;
    int    options3;
    char   *app;
    char   *postExecCmd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
};
struct jobModLog {
    char    *jobIdStr;
    int     options;
    int     options2;
    int     delOptions;
    int     delOptions2;
    int     userId;
    char    *userName;
    int     submitTime;
    int     umask;
    int     numProcessors;
    int     beginTime;
    int     termTime;
    int     sigValue;
    int     restartPid;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    char    *dependCond;
    char    *timeEvent;
    char    *subHomeDir;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *inFileSpool;
    char    *commandSpool;
    int     chkpntPeriod;
    char    *chkpntDir;
```

SYNOPSIS

```
    int     nxf;
    struct  xFile *xf;
    char    *jobFile;
    char    *fromHost;
    char    *cwd;
    char    *preExecCmd;
    char    *mailUser;
    char    *projectName;
    int     niosPort;
    int     maxNumProcessors;
    char    *loginShell;
    char    *schedHostType;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *jobGroup;
    char    *sla;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    char    *apsString;
    char    *postExecCmd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
};
struct jobStartLog {      /* logged in lsb.events when a job is started */
    int    jobId;
    int    jStatus;
    int    jobPid;
    int    jobPGid;
    float  hostFactor;
    int    numExHosts;
    char   **execHosts;
    char   *queuePreCmd;
    char   *queuePostCmd;
    int    jFlags;
    char   *userGroup;
    int    idx;
    char   *additionalInfo;
    int    duration4PreemptBackfill;
};
struct jobStartAcceptLog {  /* logged in lsb.events when a job start request is accepted
*/
    int    jobId;
    int    jobPid;
    int    jobPGid;
    int    idx;
};
struct jobExecuteLog {   /* logged in lsb.events when a job is executed */
    int    jobId;
    int    execUid;
    char   *execHome;
    char   *execCwd;
    int    jobPGid;
    char   *execUsername;
    int    jobPid;
    int    idx;
    char   *additionalInfo;
    int    SLAscaledRunLimit;
    int    position;
    char   *execRusage;
    int    duration4PreemptBackfill;
};
struct jobStatusLog {     /* logged when a job's status is changed */
    int    jobId;
    int    jStatus;
    int    reason;
    int    subreasons;
    float  cpuTime;
    time_t endTime;
    int    ru;
    struct lsfRusage lsfRusage;
    int   jFlags;
    int   exitStatus;
    int    idx;
    int    exitInfo;
};
struct sbdJobStatusLog {     /* logged when a job's status is changed */
    int    jobId;
```

SYNOPSIS

```
    int    jStatus;
    int    reasons;
    int    subreasons;
    int    actPid;
    int    actValue;
    time_t actPeriod;
    int    actFlags;
    int    actStatus;
    int    actReasons;
    int    actSubReasons;
    int    idx;
    int    sigValue;
    int    exitInfo;
};
struct sbdUnreportedStatusLog {   /* job status that we could send to MBD */
    int    jobId;
    int    actPid;
    int    jobPid;
    int    jobPGid;
    int    newStatus;
    int    reason;
    int    subreasons;
    struct lsfRusage lsfRusage;
    int    execUid;
    int    exitStatus;
    char   *execCwd;
    char   *execHome;
    char   *execUsername;
    int    msgId;
    struct jRusage runRusage;
    int    sigValue;
    int    actStatus;
    int    seq;
    int    idx;
    int    exitInfo;
};
struct jobSwitchLog {     /* logged when a job is switched to another queue */
    int    userId;
    int    jobId;
    char   queue[MAX_LSB_NAME_LEN];
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
```

```
};
struct jobMoveLog {       /* logged when a job is moved to another position */
    int    userId;
    int    jobId;
    int    position;
    int    base;
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
};
struct chkpntLog {
    int jobId;
    time_t period;
    int pid;
    int ok;
    int flags;
    int    idx;
};
struct jobRequeueLog {
    int   jobId;
    int   idx;
};
struct jobCleanLog {
    int   jobId;
    int   idx;
};
struct jobExceptionLog {
    int jobId;
    int    exceptMask;
    int    actMask;
    time_t timeEvent;
    int    exceptInfo;
    int    idx;
};
struct sigactLog {
    int      jobId;
    time_t   period;
    int      pid;
    int      jStatus;
    int      reasons;
    int      flags;
    char     *signalSymbol;
    int      actStatus;
    int      idx;
```

SYNOPSIS

```
};
struct migLog {
    int    jobId;
    int    numAskedHosts;
    char   **askedHosts;
    int    userId;
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
};
struct signalLog {
    int    userId;
    int    jobId;
    char   *signalSymbol;
    int    runCount;
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
};
struct queueCtrlLog { /* logged when bqc command is invoked */
    int  opCode;
    char queue[MAX_LSB_NAME_LEN];
    int  userId;
    char userName[MAX_LSB_NAME_LEN];
    char message[MAXLINELEN];
};
struct newDebugLog {
    int opCode;
    int level;
    int logclass;
    int turnOff;
    char logFileName[MAXLSFNAMELEN];
    int userId;
 };
struct hostCtrlLog { /* logged when bhc command is invoked */
    int  opCode;
    char host[MAXHOSTNAMELEN];
    int  userId;
    char userName[MAX_LSB_NAME_LEN];
    char message[MAXLINELEN];
};
struct hgCtrlLog {       /* logged when dynamic hosts are added to group */
    int    opCode;
    char   host[MAXHOSTNAMELEN];
    char   grpname[MAXHOSTNAMELEN];
    int    userId;
    char   userName[MAX_LSB_NAME_LEN];
    char   message[MAXLINELEN];
};
struct mbdStartLog {
    char master[MAXHOSTNAMELEN];
    char cluster[MAXLSFNAMELEN];
    int  numHosts;
    int  numQueues;
};
struct mbdDieLog {
    char master[MAXHOSTNAMELEN];
    int  numRemoveJobs;
    int  exitCode;
    char   message[MAXLINELEN];
};
struct unfulfillLog { /* logged before mbatchd dies */
    int  jobId;
    int  notSwitched;
    int  sig;
    int  sig1;
    int  sig1Flags;
    time_t chkPeriod;
    int  notModified;
    int  idx;
    int  miscOpts4PendSig;
};
struct jobFinishLog {          /* logged in lsb.acct when a job finished */
    int    jobId;
    int    userId;
    char   userName[MAX_LSB_NAME_LEN];
    int    options;
    int    numProcessors;
    int    jStatus;
    time_t submitTime;
    time_t beginTime;
    time_t termTime;
    time_t startTime;
    time_t endTime;
    char   queue[MAX_LSB_NAME_LEN];
    char   *resReq;
    char   fromHost[MAXHOSTNAMELEN];
    char   *cwd;
```

SYNOPSIS

```
    char   *inFile;
    char   *outFile;
    char   *errFile;
    char   *inFileSpool;
    char   *commandSpool;
    char   *jobFile;
    int    numAskedHosts;
    char   **askedHosts;
    float  hostFactor;
    int    numExHosts;
    char   **execHosts;
    float  cpuTime;
    char   *jobName;
    char   *command;
    struct  lsfRusage lsfRusage;
    char   *dependCond;
    char   *timeEvent;
    char   *preExecCmd;
    char   *mailUser;
    char   *projectName;
    int    exitStatus;
    int    maxNumProcessors;
    char   *loginShell;
    int    idx;
    int    maxRMem;
    int    maxRSwap;
    char   *rsvId;
    char   *sla;
    int    exceptMask;
    char   *additionalInfo;
    int    exitInfo;
    int    warningTimePeriod;
    char   *warningAction;
    char   *chargedSAAP;
    char   *licenseProject;
    char   *app;
    char   *postExecCmd;
    int    runtimeEstimation;
    char   *jgroup;
    char   *requeueEValues;
};
struct loadIndexLog {
   int  nIdx;
```

```
    char **name;
};
struct calendarLog {
    int    options;
    int    userId;
    char   *name;
    char   *desc;
    char   *calExpr;
};
struct jobForwardLog {
    int    jobId;
    char   *cluster;
    int    numReserHosts;
    char   **reserHosts;
    int    idx;
    int    jobRmtAttr;
};
struct jobAcceptLog {
    int         jobId;
    LS_LONG_INT remoteJid;
    char        *cluster;
    int         idx;
    int         jobRmtAttr;
};
struct statusAckLog {
    int jobId;
    int statusNum;
    int    idx;
};
struct jobMsgLog {
    int   usrId;
    int   jobId;
    int   msgId;
    int   type;
    char  *src;
    char  *dest;
    char  *msg;
    int   idx;
};
struct jobMsgAckLog {
    int usrId;
    int jobId;
    int msgId;
```

SYNOPSIS

```
    int type;
    char *src;
    char *dest;
    char *msg;
    int    idx;
};
struct jobOccupyReqLog {
    int userId;
    int jobId;
    int numOccupyRequests;
    void *occupyReqList;
    int    idx;
    char userName[MAX_LSB_NAME_LEN];
};
struct jobVacatedLog {
    int userId;
    int jobId;
    int    idx;
    char userName[MAX_LSB_NAME_LEN];
};
struct jobForceRequestLog {
    int     userId;
    int     numExecHosts;
    char**  execHosts;
    int     jobId;
    int     idx;
    int     options;
    char    userName[MAX_LSB_NAME_LEN];
    char   *queue;
};
struct jobChunkLog {
    long         membSize;
    LS_LONG_INT *membJobId;
    long         numExHosts;
    char       **execHosts;
};
struct jobExternalMsgLog {
    int      jobId;
    int      idx;
    int      msgIdx;
    char     *desc;
    int      userId;
    long     dataSize;
    time_t   postTime;
    int      dataStatus;
    char     *fileName;
    char     userName[MAX_LSB_NAME_LEN];
};
struct rsvRes {
    char     *resName;
    int      count;
    int      usedAmt;
};
struct rsvFinishLog { /* for advanced reservation */
    time_t        rsvReqTime;
    int           options;
    int           uid;
    char          *rsvId;
    char          *name;
    int           numReses;
    struct rsvRes *alloc;
    char          *timeWindow;
    time_t        duration;
    char          *creator;
};
struct cpuProfileLog {
    char    servicePartition[MAX_LSB_NAME_LEN];
    int     slotsRequired;
    int     slotsAllocated;
    int     slotsBorrowed;
    int     slotsLent;
};
struct jobRunRusageLog {         /* log the running rusage of a job in the lsb.stream
file */
    int             jobid;
    int             idx;
    struct jRusage  jrusage;
};
struct slaLog { /* SLA event */
    char   *name;
    char   *consumer;
    int    goaltype;
    int    state;
    int    optimum;
    int    counters[NUM_JGRP_COUNTERS];
```

```
};
struct perfmonLogInfo { /* a wrap of structure perfmonLog for perf. metrics project */
    int samplePeriod;
    int * metrics;
    time_t startTime;
    time_t logTime;
};
struct perfmonLog {              /* performance metrics log in lsb.stream */
    int    samplePeriod;
    int    totalQueries;
    int    jobQuries;
    int    queueQuries;
    int    hostQuries;
    int    submissionRequest;
    int    jobSubmitted;
    int    dispatchedjobs;
    int    jobcompleted;
    int    jobMCSend;
    int    jobMCReceive;
    time_t startTime;
};
struct taskFinishLog {     /* task accounting record in ssched.acct */
    struct jobFinishLog jobFinishLog;
    int    taskId;
    int    taskIdx;
    char   *taskName;
    int    taskOptions;
    int    taskExitReason;
};
struct eventEOSLog {     /* Event end of stream. */
    int   eos;
};
```

PARAMETERS

log_fp	Either an event log file or a job log file.
***lineNum**	The number of the event record.
struct eventRec	The `eventRec` structure contains the following fields:

version

The `mbatchd` version number

type

The type of event—one of the following:

EVENT_JOB_NEW

New job submitted

EVENT_JOB_START

`mbatchd` is trying to start a job

EVENT_JOB_START_ACCEPT

Accept Job started

EVENT_JOB_STATUS

Job's status change event

EVENT_JOB_SWITCH

Job switched to another queue

EVENT_JOB_MOVE

Move a pending job's position within a queue

EVENT_QUEUE_CTRL

Queue's status changed by Platform LSF administrator (bhc operation)

EVENT_HOST_CTRL

Host status changed by Platform LSF administrator (bhc operation)

EVENT_MBD_START

New `mbatchd` start event

EVENT_MBD_DIE

Log parameters before `mbatchd` died

EVENT_MBD_UNFULFILL

Action that was not taken because the mbatchd was unable to contact the `sbatchd` on the job's execution host

EVENT_JOB_FINISH

Job finished (Logged in `lsb.acct`)

EVENT_LOAD_INDEX

The complete list of load indices, including external load indices

EVENT_CHKPNT

Job checkpointed.

EVENT_MIG

Job migrated

EVENT_PRE_EXEC_START

The pre-execution command started

EVENT_JOB_MODIFY

Job modification request

EVENT_JOB_MODIFY2

Job modification request

EVENT_JOB_ATTR_SET

Job attributes have been set

EVENT_CAL_NEW: deprecated

Add new calendar to the system

EVENT_CAL_MODIFY: deprecated

Calendar modified

EVENT_CAL_DELETE: deprecated

Calendar deleted

EVENT_CAL_UNDELETE: deprecated

Undo delete of Calendar

EVENT_JOB_FORWARD

Job forwarded to another cluster

EVENT_JOB_ACCEPT

Job from a remote cluster dispatched

EVENT_STATUS_ACK

Job status successfully sent to submission cluster

EVENT_JOB_SIGNAL

Signal/delete a job

EVENT_JOB_EXECUTE

Job started successfully on the execution host

EVENT_JOB_MSG

Send a message to a job

EVENT_JOB_MSG_ACK

The message has been delivered

EVENT_JOB_REQUEUE

Job is requeued

EVENT_JOB_CLEAN

Job is cleaned out of the core

EVENT_JOB_SIGACT

A signal action on a job has been initiated or finished

EVENT_JOB_EXCEPTION

Job exception was detected

EVENT_JGRP_ADD

Adding a new job group

EVENT_JGRP_MOD

Modifying a job group

EVENT_JGRP_CTRL

Controlling a job group

EVENT_JGRP_STATUS

Log job group status

EVENT_JOB_OCCUPY_REQ

Submission mbatchd logs this after sending an occupy request to execution mbatchd

EVENT_JOB_VACATED

Submission mbatchd logs this event after all execution mbatchds have vacated the occupied hosts for the job

EVENT_SBD_JOB_STATUS

sbatchd's new job status

EVENT_JOB_FORCE

Forcing a job to start on specified hosts (`brun` operation)

EVENT_LOG_SWITCH

Switching the event file lsb.events

EVENT_JOB_CHUNK

Insert one job to a chunk

EVENT_SBD_UNREPORTED_STATUS

Save unreported sbatchd status

EVENT_JOB_EXT_MSG

Send an external message to a job

EVENT_JOB_ATTA_DATA

Update data status of a message for a job

EVENT_ADRSV_FINISH

Reservation finished

EVENT_CPUPROFILE_STATUS

Saved current CPU allocation on service partition

EVENT_DATA_LOGGING

Write out data logging file

EVENT_CPUPROFILE_STATUS

Write CPU profile status

EVENT_DATA_LOGGING

Log data

EVENT_JOB_RUN_RUSAGE

Write job rusage in lsb.stream

EVENT_SLA_RECOMPUTE

SLA goal is reavaluated

EVENT_JOB_ROUTE

The job has been routed to NQS

EVENT_TASK_FINISH

Write task finish log to `ssched.acct`

JOB_RUN_RUSAGE

Writes job rusage in lsb.stream.

END_OF_STREAM

Stream closed and new stream opened.

SLA_RECOMPUTE

SLA goal is reavaluated.

eventTime

The time the event occurred

eventLog

The information for this type of event, contained in a structure corresponding to type

`struct jobNewLog` The `jobNewLog` structure contains the following fields:

jobId

The job ID that the LSF assigned to the job

userId

The user ID of the submitter

userName

The name of the submitter

options

Job submission options. See `lsb_submit()`.

options2

Job submission options. See `lsb_submit()`.

numProcessors

The number of processors requested for execution

submitTime

The job submission time

beginTime

The job should be started on or after this time

termTime

If the job has not finished by this time, it will be killed

sigValue

The signal value sent to the job 10 minutes before its run window closes

chkpntPeriod

The checkpointing period

restartPid

The process ID assigned to the job when it was restarted

rLimits

The user’s resource limits

hostSpec

The model, host name or host type for scaling `CPULIMIT` and `RUNLIMIT`

hostFactor

The CPU factor for the above model, host name or host type

umask

The file creation mask for this job

queue

The name of the queue to which this job was submitted

resReq

The resource requirements of the job

fromHost

The submission host name

cwd

The current working directory

chkpntDir

The checkpoint directory

inFile

The input file name

outFile

The output file name

errFile

The error output file name

inFileSpool

Job spool input file

commandSpool

Job spool command file

jobSpoolDir

job spool directory

subHomeDir

The home directory of the submitter

jobFile

The job file name

numAskedHosts

The number of hosts considered for dispatching this job

askedHosts

The array of names of hosts considered for dispatching this job

dependCond

The job dependency condition

timeEvent

Time event string

jobName

The job name

command

The job command

nxf

The number of files to transfer

xf

The array of file transfer specifications. (The `xFile` structure is defined in `<lsf/lsbatch.h>`)

preExecCmd

The pre-execution command

mailUser

User option mail string

projectName

Project name for the job; used for accounting purposes

niosPort

NIOS callback port to be used for interactive jobs

maxNumProcessors

Maximum number of processors

schedHostType

Execution host type

loginShell

The login shell specified by user

userGroup

The user group name for this job

exceptList

List of job exception conditions

idx

Job array index; must be 0 in JOB_NEW

userPriority

User priority

rsvId

Advance reservation ID

jobGroup

The job group under which the job runs

extsched

External scheduling options

warningTimePeriod

Job warning time period in seconds; -1 if unspecified

warningAction

Job warning action: SIGNAL | CHKPNT | command; NULL if unspecified

sla

SLA service class name under which the job runs

SLArunLimit

Absolute run time limit of the job for SLA service classes

licenseProject

LSF License Scheduler project name

options3

Extended bitwise inclusive OR of options flags. See `lsb_submit()`.

app

Application profile under which the job runs.

postExecCmd

Post-execution commands.

runtimeEstimation

Runtime estimate specified.

requeueEValues

Job-level requeue exit values.

initChkpntPeriod

Initial checkpoint period.

migThreshold

Job migration threshold.

`struct jobStartLog` The `jobStartLog` structure contains the following fields:

jobId

The unique ID for the job

jStatus

The status of the job (See `lsb_readjobinfo()`)

jobPid

The job process ID

PARAMETERS

jobPGid

The job process group ID

hostFactor

The CPU factor of the first execution host

numExHosts

The number of processors used for execution

execHosts

The array of execution host names

queuePreCmd

Pre-execution command defined in the queue

queuePostCmd

Post-execution command defined in the queue

jFlags

Job processing flags

userGroup

The user group name for this job

idx

Job array index; must be 0 in JOB_NEW

additionalInfo

Placement information of LSF HPC jobs

duration4PreemptBackfill

How long a backfilled job can run; used for preemption backfill jobs

`struct jobForceRequestLog`

The jobForceRequestLog structure contains the following fields:

userId

The user ID of the submitter

numExecHosts

Number of execution hosts

execHosts

The array of execution host names

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

options

Job run options (RUNJOB_OPT_NOSTOP | JFLAG_URGENT_NOSTOP | JFLAG_URGENT)

userName

The name of the submitter

queue

The name of the queue to which this job was submitted

`struct logSwitchLog` The logSwitchLog structure contains the following fields:

lastJobId

the last jobId so far

`struct jobModLog` The `jobModLog` structure contains the following fields:

jobIdStr

jobId or jobName in char

options

Job submission options (See `lsb_submit()`)

options2

Job submission options (See `lsb_submit()`)

delOptions

Delete options in `options` field .

delOptions2

Extended delete options in `options2` field .

userId

The user ID of the submitter

userName

The name of the submitter

submitTime

The job submission time

umask

The file creation mask for this job

numProcessors

The number of processors requested for execution

beginTime

The job should be started on or after this time

termTime

If the job has not finished by this time, it will be killed

sigValue

The signal value sent to the job 10 minutes before its run window closes

restartPid

The process ID assigned to the job when it was restarted

jobName

The job name

PARAMETERS

queue

The name of the queue to which this job was submitted

numAskedHosts

The number of hosts considered for dispatching this job

askedHosts

List of asked hosts

resReq

The resource requirements of the job

rLimits

The user's resource limits

hostSpec

The model, host name or host type for scaling `CPULIMIT` and `RUNLIMIT`

dependCond

The job dependency condition

timeEvent

Time event string.

subHomeDir

The home directory of the submitter

inFile

The input file name

outFile

The output file name

errFile

The error output file name

command

The job command

inFileSpool

Job spool input file

commandSpool

Job spool command file

chkpntPeriod

The checkpointing period

chkpntDir

The checkpoint directory

nxf

The number of files to transfer

xf

The array of file transfer specifications. (The `xFile` structure is defined in `<lsf/lsbatch.h>`)

jobFile

The job file name

fromHost

The submission host name

cwd

The current working directory

preExecCmd

The pre-execution command

mailUser

User option mail string

projectName

Project name for the job; used for accounting purposes

niosPort

NIOS callback port to be used for interactive jobs

maxNumProcessors

Maximum number of processors

loginShell

The login shell specified by user

schedHostType

Execution host type

userGroup

The user group name for this job

exceptList

List of job exception conditions

userPriority

User priority

rsvId

Advance reservation ID

extsched

External scheduling options

warningTimePeriod

Job warning time period in seconds; -1 if unspecified

warningAction

Job warning action: SIGNAL | CHKPNT | command; NULL if unspecified

jobGroup

The job group under which the job runs

sla

SLA service class name under which the job runs

licenseProject

LSF License Scheduler project name

options3

Extended bitwise inclusive OR of options flags. See `lsb_submit()`.

delOptions3

Extended delete options in options3 field.

app

Application profile under which the job runs.

apsString

Absolute priority scheduling string set by administrators to denote static system APS value or ADMIN factor APS value.

postExecCmd

Post-execution commands.

runtimeEstimation

Runtime estimate.

requeueEValues

Job-level requeue exit values.

initChkpntPeriod

Initial checkpoint period.

migThreshold

Job migration threshold.

`struct jobStatusLog`

The `jobStatusLog` structure contains the following fields:

jobId

The unique ID for the job

jStatus

The job status (See `lsb_readjobinfo()`)

reason

The reason the job is pending or suspended (See `lsb_pendreason()` and `lsb_suspreason()`)

subreasons

The load indices that have overloaded the host (See `lsb_pendreason()` and `lsb_suspreason()`)

cpuTime

The CPU time consumed before this event occurred

endTime

The job completion time

ru

Boolean indicating `lsfRusage` is logged

lsfRusage

Resource usage statistics

The `lsfRusage` structure is defined in <`lsf/lsf.h`>. Note that the availability of certain fields depends on the platform on which the sbatchd runs. The fields that do not make sense on the platform will be logged as -1.0.

exitStatus

Job exit status

idx

Job array index; must be 0 in JOB_NEW

exitInfo

Job termination reason, see `<lsf/lsbatch.h>`

struct migLog The `migLog` structure contains the following fields:

jobId

The job to be migrated

numAskedHosts

The number of candidate hosts for migration

askedHosts

The array of candidate host names

userId

The user ID of the submitter

idx

Job array index; must be 0 in JOB_NEW

userName

The user name of the submitter

struct sigactLog The sigactLog structure contains the following fields:

jobId

The unique ID of the job

period

action period

pid

action process ID

jStatus

job status

reasons

Pending reasons

flags

Action flag

signalSymbol

signal symbol from the set: DELETEJOB | KILL | KILLREQUEUE | REQUEUE_DONE | REQUEUE_EXIT | REQUEUE_PEND | REQUEUE_PSUSP_ADMIN | REQUEUE_PSUSP_USER | SIG_CHKPNT | SIG_CHKPNT_COPY

actStatus

action logging status (ACT_NO | ACT_START | ACT_PREEMPT | ACT_DONE | ACT_FAIL)

idx

Job array index; must be 0 in JOB_NEW

struct jobOccupyReqLog

The jobOccupyReqLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID for the job

numOccupyRequests

Number of Jobs Slots desired

occupyReqList

List of slots occupied

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

struct jobVacatedLog

The jobVacatedLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

struct jobCleanLog

The jobCleanLog structure contains the following fields:

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

struct jobStartAcceptLog

The jobStartAcceptLog structure contains the following fields:

jobId

The unique ID for the job

jobPid

The job process ID

jobPGid

The job process group ID

idx

Job array index; must be 0 in JOB_NEW

struct jobExceptionLog

The jobExceptionLog structure contains the following fields:

jobId

The unique ID for the job

exceptMask

Job exception handling mask

ActMask

Action Id (kill | alarm | rerun | setexcept)

timeEvent

Time event string

ExceptInfo

Except Info, pending reason for missched or cantrun exception, the exit code of the job for the abend exception, otherwise 0.

idx

Job array index; must be 0 in JOB_NEW

struct jobForceRequestLog

The jobForceRequestLog structure contains the following fields:

userId

The user ID of the submitter

numExecHosts

The number of execution hosts

ExecHosts

The array of execution host names

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

options

Job run options (RUNJOB_OPT_NOSTOP | JFLAG_URGENT_NOSTOP | JFLAG_URGENT)

userName

The name of the submitter

queue

The name of the queue to which this job was submitted

struct logSwitchLog The logSwitchLog structure contains the following fields:

lastJobId

the last jobId so far

struct jobModLog The jobModLog structure contains the following fields:

jobIdStr

Job id

options

Job submission options

options2

Job submission options – more

delOptions

Delete options in options field

delOptions2

Delete options in options2 field

userId

The user ID of the submitter

userName

The name of the submitter

submitTime

time of job submission

umask

The file creation mask for this job

numProcessors

min num of proc for the job

beginTime

mustn't start before this

termTime

kill if not done after this

sigValue

signal value

restartPid

pid of original job

jobName

Job name

queue

Queue name

numAskedHosts

number of user specified hosts

askedHosts

user specified execution hosts

resReq

resource request

rLimits

user's resource limits (soft)

hostSpec

host/model name for CPU scale

dependCond

depend_cond expression string

timeEvent

Time event string

subHomeDir

job's homedir at submission host

inFile

input file

outFile

output file

errFile

error file

command

command description - this is really a job description field

inFileSpool

spool input file

commandSpool

spool command file

chkpntPeriod

The checkpointing period

chkpntDir

The checkpointing directory

nxf

Number of files to be copied

xf

Files to be copied

jobFile

The job file name. ‘\0’ indicate let mbatchd make up name, otherwise, mbatchd will use given name. It is '\0' if it is a regular job, non-nil means it is a restart job.

fromHost

The submission host name

cwd

The current working directory

preExecCmd

Command string to be pre_executed

mailUser

Specified user results mailed to

projectName

project name for acct purposes

niosPort

nios port for interactive job

maxNumProcessors

max num of proc for the job

loginShell

login shell specified by user

schedHostType

restart job's submission host type

userGroup

user group

exceptList

exceptions to be detected

userPriority

user priority

rsvId

reservation ID

extsched

extsched option

warningTimePeriod

warning time period in seconds, -1 if unspecified

warningAction

warning action, SIGNAL | CHKPNT | command, NULL if unspecified

jobGroup

job group

sla

service class

licenseProject

License Project

options3

Job submission options – more and more

delOptions3

Delete options in options3 field

app

Application

apsString

aps value set by admin

postExecCmd

Post-execution commands specified by -Ep option of bsub and bmod

runtimeEstimation

Runtime estimate specified by -We option of bsub and bmod

struct sigactLog The sigactLog structure contains the following fields:

jobId

The unique ID for the job

period

action period

pid

action process ID

jStatus

job status

reasons

pending reasons

flags

action flag

signalSymbol

signal symbol from the set: DELETEJOB | KILL | KILLREQUEUE | REQUEUE_DONE | REQUEUE_EXIT | REQUEUE_PEND | REQUEUE_PSUSP_ADMIN | REQUEUE_PSUSP_USER | SIG_CHKPNT | SIG_CHKPNT_COPY

actStatus

action logging status (ACT_NO | ACT_START | ACT_PREEMPT | ACT_DONE | ACT_FAIL)

idx

Job array index; must be 0 in JOB_NEW

struct jobOccupyReqLog

The jobOccupyReqLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID for the job

numOccupyRequests

Number of Jobs Slots desired

occupyReqList

list of slots occupied

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

struct jobVacatedLog

The jobVacatedLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

struct jobStartAcceptLog

The jobStartAcceptLog structure contains the following fields:

jobId

The unique ID for the job

jobPid

The job process ID

jobPGid

The job process group ID

idx

Job array index; must be 0 in JOB_NEW

`struct jobCleanLog`

The jobCleanLog structure contains the following fields:

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

struct jobExceptionLog The jobExceptionLog structure contains the following fields:

jobId

The unique ID for the job

exceptMask

Job exception handling mask

ActMask

Action Id (kill | alarm | rerun | setexcept)

timeEvent

Time event string

ExceptInfo

Except Info, pending reason for missched or cantrun exception, the exit code of the job for the abend exception, otherwise 0.

idx

Job array index; must be 0 in JOB_NEW

struct jgrpStatusLog The `jgrpStatusLog` structure contains the following fields:

groupSpec

The full group path name for the job group

status

Job group status

oldStatus

Prior status

struct jgrpNewLog The `jgrpNewLog` structure contains the following fields:

userId

The user ID of the submitter

submitTime

The job submission time

userName

The name of the submitter

dependCond

The job dependency condition

timeEvent

Time event string

groupSpec

Job group name

destSpec

New job group name

delOptions

Delete options in `options` field .

delOptions2

Extended delete options in `options2` field .

fromPlatform

Platform type e.g. Unix, Windows

sla

SLA service class name under which the job runs

maxJLimit

Job group slot limit

options

Job group creation method: implicit or explicit

struct jgrpCtrlLog The `jgrpCtrlLog` structure contains the following fields:

userId

The user ID of the submitter

userName

The name of the submitter

groupSpec

Job group name

options

Options

ctrlOp

Job control JGRP_RELEASE, JGRP_HOLD, JGRP_DEL

struct jobAttrSetLog The jobAttrSetLog structure contains the following fields:

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

uid

The user who requested the action

port

job attributes

hostname

Name of the host

struct jobExternalMsgLog The jobExternalMsgLog structure contains the following fields:

jobId

The unique ID for the job

idx

Job array index; must be 0 in JOB_NEW

msgIdx

The message index

desc

Message description

userId

The user ID of the submitter

dataSize

Size of the message

postTime

The time the author posted the message.

dataStatus

The status of the message

fileName

Name of attached data file. If no file is attached, use NULL.

userName

The author of the message

struct perfmonLog The perfmonLog structure contains the following fields:

samplePeriod

sample rate

totalQueries

Number of Queries

jobQueries

Number of Job Query

queueQuries

Number of Queue Query

hostQuries

Number of Host Query

SubmissionRequest

Number of Submission Requests

jobSubmitted

Number of Jobs Submitted

dispatchedjobs

Number of Dispatched Jobs

jobcompleted

Number of Job Completed

jobMCSend

Number of MultiCluster Jobs Sent

jobMCReceive

Number of MultiCluster Jobs Received

startTime

Start Time

structure jobChunkLog

The `jobChunkLog` structure contains the following fields:

membSize

Size of array `membJobId`

membJobId

Job IDs of jobs in the chunk

numExHosts

The number of processors used for execution

execHosts

The array of names of execution hosts

struct sbdJobStatusLog

The sbdJobStatusLog structure contains the following fields:

jobId

The unique ID for the job

jStatus

The status of the job (See lsb_readjobinfo())

reason

The reason the job is pending or suspended (See lsb_pendreason() and lsb_suspreason())

subreasons

The load indices that have overloaded the host (See lsb_pendreason() and lsb_suspreason())

actPid

Action process ID

actValue

Action Value SIG_CHKPNT | SIG_CHKPNT_COPY | SIG_WARNING

actPeriod

Action period

actFlags

Action flag

actStatus

Action logging status

actReasons

Action Reason SUSP_MBD_LOCK | SUSP_USER_STOP | SUSP_USER_RESUME | SUSP_SBD_STARTUP

ActSubReasons

Sub Reason SUB_REASON_RUNLIMIT | SUB_REASON_DEADLINE | SUB_REASON_PROCESSLIMIT | SUB_REASON_MEMLIMIT | SUB_REASON_CPULIMIT

idx

Job array index; must be 0 in JOB_NEW

sigValue

The signal value

exitInfo

The termination reason of a job

`struct jobSwitchLog`

The `jobSwitchLog` structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID of the job

queue

The name of the queue the job has been switched to

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

`struct jobMoveLog`

The `jobMoveLog` structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID of the job

position

The new position of the job

base

The operation code for the move (See `lsb_movejob()`)

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

`struct queueCtrlLog` The `queueCtrlLog` structure contains the following fields:

opCode

The queue control operation (See `lsb_queuecontrol()`)

queue

The name of the queue

userId

The user ID of the submitter

userName

The name of the submitter

message

Queue control message

struct newDebugLog The newDebugLog structure contains the following fields:

opCode

The queue control operation (See lsb_queuecontrol())

level

Debug level

logclass

Class of log

turnOff

Log enabled, disabled

logFileName

Name of log file

userId

The user ID of the submitter

struct hostCtrlLog The `hostCtrlLog` structure contains the following fields:

opCode

The host control operation (See `lsb_hostcontrol()`)

host

The name of the host

userId

The user ID of the submitter

userName

The name of the submitter

message

Host control message

struct hgCtrlLog The `hgCtrlLog` structure contains the following fields:

opCode

The host control operation (See `lsb_hostcontrol()`)

host

The name of the host

grpname

The name of the host group

userId

The user ID of the submitter

userName

The name of the submitter

message

Host group control message

`struct mbdStartLog` The `mbdStartLog` structure contains the following fields:

master

The master host name

cluster

The cluster name

numHosts

The number of hosts in the cluster

numQueues

The number of queues in the cluster

struct mbdDieLog The `mbdDieLog` structure contains the following fields:

master

The master host name

numRemoveJobs

The number of finished jobs that have been removed from the system and logged in the current event file

exitCode

The exit code from the master batch daemon

message

`mbatchd` administrator control message

struct unfulfillLog The `unfulfillLog` structure contains the following fields:

jobId

The job ID.

notSwitched

The mbatchd has switched the job to a new queue but the sbatchd has not been informed of the switch

PARAMETERS

sig

This signal was not sent to the job

sig1

The job was not signaled to checkpoint itself

sig1Flags

Checkpoint flags. See the chkpntLog structure below.

chkPeriod

The new checkpoint period for the job

notModified

Flag for modifying job parameters of a running job

idx

Job array index

miscOpts4PendSig

Option flags for pending job signals

struct jobFinishLog The `jobFinishLog` structure contains the following fields:

jobId

The unique ID for the job

userId

The user ID of the submitter

userName

The user name of the submitter

options

Job submission options (See `lsb_submit()`)

numProcessors

The number of processors requested for execution

jStatus

The status of the job (See `lsb_readjobinfo()`)

submitTime

Job submission time

beginTime

The job started at or after this time

termTime

If the job was not finished by this time, it was killed

startTime

Job dispatch time

endTime

The time the job finished

queue

The name of the queue to which this job was submitted

resReq

Resource requirements

fromHost

Submission host name

cwd

Current working directory

inFile

Input file name

outFile

Output file name

errFile

Error output file name

inFileSpool

Job spool input file

commandSpool

Job spool command file

jobFile

Job file name

numAskedHosts

The number of hosts considered for dispatching this job

askedHosts

The array of names of hosts considered for dispatching this job

hostFactor

The CPU factor of the first execution host

numExHosts

The number of processors used for execution

execHosts

The array of names of execution hosts

cpuTime

The total CPU time consumed by the job

jobName

Job name

command

Job command

lsfRusage

Resource usage statistics

The `lsfRusage` structure is defined in `<lsf/lsf.h>`. Note that the availability of certain fields depends on the platform on which the sbatchd runs. The fields that do not make sense on this platform will be logged as -1.0.

dependCond

The job dependency condition

timeEvent

Time event string

preExecCmd

The pre-execution command

mailUser

Name of the user to whom job related mail was sent

projectName

The project name, used for accounting purposes

exitStatus

Job exit status

maxNumProcessors

Maximum number of processors specified for the job

loginShell

The login shell specified by user

idx

Job array index

maxRMem

Maximum memory used by job

maxRSwap

Maximum swap used by job

rsvId

Advanced reservation ID

sla

SLA service class name for the job

exceptMask

Job exception handling mask

additionalInfo

Placement information of LSF HPC jobs

exitInfo

Job termination reason, see `<lsf/lsbatch.h>`

warningTimePeriod

Job warning time period in seconds; -1 if unspecified

warningAction

Job warning action, SIGNAL | CHKPNT | command; NULL if unspecified

chargedSAAP

SAAP charged for job

licenseProject

LSF License Scheduler project name

app

Application profile under which the job runs.

postExecCmd

Post-execution commands.

runtimeEstimation

Runtime estimate specified.

jgroup

Job group name

requeueEValues

Job-level requeue exit values

struct loadIndexLog The `loadIndexLog` structure contains the following fields:

nIdx

The number of load indices

name

The array of load index names

struct calendarLog The calendarLog structure contains the following fields

options

Reserved for future use

userId

The user ID of the submitter

name

The name of the calendar

desc

Description

calExpr

calendar expression

struct jobForwardLog The jobForwardLog structure contains the following fields:

jobId

The unique ID of the job

cluster

PARAMETERS

The cluster name

numReserHosts

Number of Reserved Hosts

reserHosts

Reserved Host Names

idx

Job array index; must be 0 in JOB_NEW

jobRmtAttr

Remote job attributes from:

JOB_FORWARD	Remote batch job on submit side
JOB_LEASE	Lease job on submit side
JOB_REMOTE_BATCH	Remote batch job on exec. side
JOB_REMOTE_LEASE	Lease job on exec. side
JOB_LEASE_RESYNC	Lease job resync during restart
JOB_REMOTE_RERUNNABLE	Remote batch job rerunnable on execution cluster

struct jobAcceptLog The jobAcceptLog structure contains the following fields:

jobId

The unique ID of the job

remoteJid

The unique ID of the remote job

cluster

The cluster name

idx

Job array index; must be 0 in JOB_NEW

jobRmtAttr

Remote job attributes from:

JOB_FORWARD	Remote batch job on submit side
JOB_LEASE	Lease job on submit side
JOB_REMOTE_BATCH	Remote batch job on exec. side
JOB_REMOTE_LEASE	Lease job on exec. side
JOB_LEASE_RESYNC	Lease job resync during restart
JOB_REMOTE_RERUNNABLE	Remote batch job rerunnable on execution cluster

statusAckLog

The statusAckLog structure contains the following fields:

jobId

The unique ID of the job

statusNum

Line number of Status

idx

Job array index; must be 0 in JOB_NEW

struct signalLog The signalLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID of the job

signalSymbol

signal symbol from the set: DELETEJOB | KILL | KILLREQUEUE | REQUEUE_DONE | REQUEUE_EXIT | REQUEUE_PEND | REQUEUE_PSUSP_ADMIN | REQUEUE_PSUSP_USER | SIG_CHKPNT | SIG_CHKPNT_COPY

runCount

the number of running times

idx

Job array index; must be 0 in JOB_NEW

userName

The name of the submitter

struct jobExecuteLog The jobExecuteLog structure contains the following fields:

jobId

The unique ID of the job

execUid

User ID under which the job is running

execHome

Home directory of the user denoted by execUid

execCwd

Current working directory where job is running

jobPGid

The job process group ID

execUsername

User name under which the job is running

jobPid

The job process ID

idx

Job array index; must be 0 in JOB_NEW

additionalInfo

Placement information of LSF HPC jobs

SLAscaledRunLimit

Run time limit for the job scaled by the execution host

position

The position of the job

execRusage

The rusage satisfied at job runtime

duration4PreemptBackfill

The duration for preemptive backfill class in seconds

struct jobMsgLog The jobMsgLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID of the job

msgId

message index

type

message type

src

message source

dest

message destination

msg

message

idx

Job array index; must be 0 in JOB_NEW

struct jobMsgAckLog The jobMsgAckLog structure contains the following fields:

userId

The user ID of the submitter

jobId

The unique ID of the job

msgId

message index

type

message type

src

message source

dest

message destination

msg

message

idx

Job array index; must be 0 in JOB_NEW

struct jobRequeueLog The jobRequeueLog structure contains the following fields:

jobId

The unique ID of the job

idx

Job array index; must be 0 in JOB_NEW

struct chkpntLog The `chkpntLog` structure contains the following fields:

jobId

The unique ID of the job

period

The new checkpointing period

pid

The process ID of the checkpointing process (a child sbatchd)

ok

0: checkpoint started; 1: checkpoint succeeded

flags

One of the following:

LSB_CHKPNT_KILL : Kill process if checkpoint successful

LSB_CHKPNT_FORCE : Force checkpoint even if non-checkpointable conditions exist

LSB_CHKPNT_MIG : Checkpoint for the purpose of migration

idx

Job array index; must be 0 in JOB_NEW

struct rsvRes The `rsvRes` structure contains the following fields:

resName

Name of the resource (currently: host)

count

Reserved counter (currently: cpu number)

usedAmt

Amount of reserved counter used (currently: not used)

struct sbdUnreportedStatusLog The sbdUnreportedStatusLog structure contains the following fields:

jobId

PARAMETERS

The unique ID for the job

jobPid

The job process ID

jobPGid

The job process group ID

newStatus

New status of the job

reason

Pending or suspending reason code

subreasons

Pending or suspending subreason code

lsfRusage

Resource usage information for the job (see jobFinishLog)

execUid

User ID under which the job is running

exitStatus

Job exit status

execCwd

Current working directory where job is running

execHome

Home directory of the user denoted by execUid

execUsername

User name under which the job is running

msgId

Message index

runRusage

Job's resource usage

sigValue

Signal value

actStatus

Action logging status

seq

Sequence status of the job

idx

Job array index

exitInfo

Job termination reason

struct rsvFinishLog The `rsvFinishLog` structure contains the following fields:

rsvReqTime

Time when the reservation is required

options

Same as the options field in the struct addRsvRequest in `<lsbatch.h>`

uid

The user who created the reservation

rsvId

Reservation ID

name

User the reservation is for

numReses

Number of resources reserved

alloc

Allocation vector

timeWindow

Time window within which the advance reservation is active:

- time_t1-time_t2, or
- [day1]:hour1:0-[day2]:hour2:0

duration

Duration in seconds. (1) duration = to - from : when the reservation expired

creator

Creator of the reservation

struct cpuProfileLog The cpuProfileLog structure contains the following fields:

servicePartition

Queue name

slotsRequired

The number of CPUs required

slotsAllocated

The number of CPUs actually allocated

slotsBorrowed

The number of CPUs borrowed

slotsLent

The number of CPUs lent

struct dataLoggingLog The dataLoggingLog structure contains the following field:

loggingTime

The time of last job cpu data logging

PARAMETERS

struct jobRunRusageLog The jobRunRusageLog structure contains the following fields:

jobId

The unique ID of the job

idx

Job array index; must be 0 in JOB_NEW

struct jRusage The jRusage structure contains the following fields:

mem

Total resident memory usage in kilobytes of all currently running processes in given process groups

swap

Total virtual memory usage in kilobytes of all currently running processes in given proces groups.

utime

Cumulative total user time in seconds

stime

Cumulative total system time in seconds

npids

Number of currently active processesin given process groups

pidInfo

Structure containing information about an active process

struct eventEOSLog The eventEOSLog structure contains the following field:

eos

Event end of stream

struct slaLog The `slaLog` structure contains the following fields:

name

Service class name

consumer

Consumer name associated with the service class

goaltype

Objectives

state

The service class state (ontine, delayed)

optimum

Optimum number of job slots (or concurrently running jobs) needed for the service class to meet its service-level goals

counters

Job counters for the service class

struct taskFinishLog The taskFinishLog structure contains the following fields:

taskId

Task ID

taskIdx

Task index

taskName

Name of task

TaskOptions

Bit mask of task options:

TASK_IN_FILE (0x01)-specify input file

TASK_OUT_FILE (0x02)-specify output file

TASK_ERR_FILE (0x04)-specify error file

TASK_PRE_EXEC (0x08)-specify pre-exec command

TASK_POST_EXEC (0x10)-specify post-exec command

TASK_NAME (0x20)-specify task name

taskExitReason

Task Exit Reason

TASK_EXIT_NORMAL = 0- normal exit

TASK_EXIT_INIT = 1-generic task initialization failure

TASK_EXIT_PATH = 2-failed to initialize path

TASK_EXIT_NO_FILE = 3-failed to create task file

TASK_EXIT_PRE_EXEC = 4- task pre-exec failed

TASK_EXIT_NO_PROCESS = 5-fork failed

TASK_EXIT_XDR = 6-xdr communication error

TASK_EXIT_NOMEM = 7- no memory

TASK_EXIT_SYS = 8-system call failed

TASK_EXIT_TSCHILD_EXEC = 9-failed to run sschild

TASK_EXIT_RUNLIMIT = 10-task reaches run limit

TASK_EXIT_IO = 11-I/O failure

TASK_EXIT_RSRC_LIMIT = 12-set task resource limit failed

Struct jobFinishLog See jobFinishLog structure above

RETURN VALUES

character:Pointer Points to an `eventRec` which contains information on a job event and updates `*lineNum` to point to the next line of the log file.

character:NULL Function failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

If there are no more records, returns `NULL` and sets `lsberrno` to LSBE_EOF.

SEE ALSO

Related APIs

`lsb_hostcontrol()` - Opens or closes a host, or restarts or shuts down its slave batch daemon

`lsb_movejob()` - Changes the position of a pending job in a queue

`lsb_pendreason()` - Explains why a job is pending

`lsb_puteventrec()` - Puts information of an `eventRec` structure pointed to by `logPtr` into a log file

`lsb_queuecontrol()`- Changes the status of a queue

`lsb_readjobinfo()`- Returns the next job information record in mbatchd

`lsb_submit()` - Submits or restarts a job in the batch system

`lsb_suspreason()`- Explains why a job was suspended

Equivalent line command

none

FILES

`$LSB_SHAREDIR/cluster/logdir/lsb.acct`

`$LSB_SHAREDIR/cluster/logdir/lsb.events`

`$LSB_SHAREDIR/cluster/logdir/lsb.rsv.ids`

`$LSB_SHAREDIR/cluster/logdir/lsb.rsv.state`

lsb_geteventrecbyline()

Parse an event line and put the result in an event record structure

SYNOPSIS

```
#include <lsf/lsf.h>
int lsb_geteventrecbyline(char *line, struct eventRec *logRec)
```

PARAMETERS

char *line

Buffer containing a line of event text string

struct eventRec *logRec

Pointer to an eventRec structure

PRE-CONDITIONS

The event record structure must have been initialized outside the `lsb_geteventrecbyline()` function.

DESCRIPTION

The `lsb_geteventrecbyline()` function parses an event line and puts the result in an event record structure.

If the line to be parsed is a comment line, `lsb_geteventrecbyline()` sets errno to bad event format and logs an error.

RETURN

- 0 on success
- -1 on failure and set lserrno

lsb_getjobdepinfo()

Returns the job dependency information.

DESCRIPTION

`lsb_getjobdepinfo()` returns information about jobs (including job arrays) when a job has one or more dependencies on it.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct jobDependInfo *lsb_getjobdepinfo(struct jobDepRequest
*jobdepReq)
struct dependJobs {
    LS_LONG_INT jobId;
    char    *jobname;
    int     level;
    int     jobstatus;
    char    hasDependency;
    char    *condition;
    int     satisfied;
    LS_LONG_INT depjobid;
};
struct queriedJobs {
    LS_LONG_INT jobId;
    char    *dependcondition;
    int     satisfied;
};
struct jobDependInfo {
    int    options;
    int     numQueriedJobs;
    struct quieriedJobs *queriedJobs;
    int     level;
    int     numJobs;
    struct dependJobs *depJobs;
};
struct jobDepRequest {
    LS_LONG_INT jobId;
    int     options;
    int     level;
};
```

struct dependJobs

The `dependJobs` structure contains the following fields:

jobId Job ID. By default, it is the parent job of the queried job. Modify to child job by setting `QUERY_DEPEND_CHILD` in options of `JobDepRequest`.

jobname The job name associated with the job ID.

jobstatus Job status of the job.

level The number of degrees of separation from the original job.

hasDependency Whether the job ID has a dependency or not. When you set `QUERY_DEPEND_RECURSIVELY` in options of `JobDepRequest`, 0 indicates job ID does not have a dependency. Otherwise, one or more of the following bits displays:

- JOB_HAS_DEPENDENCY: Job has a dependency.
- JOB_HAS_INDIVIDUAL_CONDITION: Job has an individual dependency condition when it is an element of job array.

condition When you set "QUERY_DEPEND_DETAIL" into options, it is dependency condition of jobId. It is "" when you do not set "QUERY_DEPEND_DETAIL".

satisfied Whether the condition is satisfied.

depJobId Job ID. By default, it is the child job. Modify to parent job by setting `QUERY_DEPEND_CHILD` in options of `JobDepRequest`

struct queriedJobs

The `queriedJobs` structure contains the following fields:

jobId Job ID of the queried job or job array.

dependcondition The whole dependency condition of the job.

satisfied Whether the condition is satisfied.

struct jobDependInfo

The `jobDependInfo` structure contains the following fields:

options You can set the following bits into this field:

QUERY_DEPEND_RECURSIVELY

Query the dependency information recursively.

QUERY_DEPEND_DETAIL

Query the detailed dependency information.

QUERY_DEPEND_UNSATISFIED

Query the jobs that cause this job pend.

QUERY_DEPEND_CHILD

Query child jobs.

numQueriedJobs The number of jobs you queried. By default, the value is 1. However, when you set `QUERY_DEPEND_DETAIL` in the options and you query a job array where some elements have a dependency condition that has changed, the value is the number of the changed element + 1.

queriedJobs The jobs you queried.

level The number of levels returned.

SYNOPSIS

numJobs The number of jobs returned.

depJobs The returned dependency jobs.

structure jobDepRequest

The `jobDepRequest` structure contains the following fields:

jobid Job ID of the queried job or job array.

options You can set the following bits into this field:

QUERY_DEPEND_RECURSIVELY

Query the dependency information recursively.

QUERY_DEPEND_DETAIL

Query the detailed dependency information.

QUERY_DEPEND_UNSATISFIED

Query the jobs that cause this job pend.

QUERY_DEPEND_CHILD

Query child jobs.

level The level when you set `QUERY_DEPEND_RECURSIVELY`.

lsb_hostcontrol()

Opens or closes a host, or restarts or shuts down its slave batch daemon.

DESCRIPTION

`lsb_hostcontrol()` opens or closes a host, or restarts or shuts down its slave batch daemon. Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

To restart the master batch daemon, mbatchd, in order to use updated batch LSF configuration files, use `lsb_reconfig()`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_hostcontrol (struct hostCtrlReq)
struct hostCtrlReq {
    char  *host;
    int    opCode;
    char  *message;
};
```

PARAMETERS

***host**	The host to be controlled. If `host` is `NULL`, the local host is assumed.
opCode	One of the following:

HOST_CLOSE

Closes the host so that no jobs can dispatched to it.

HOST_OPEN

Opens the host to accept jobs.

HOST_REBOOT

Restart the sbatchd on the host. The sbatchd will receive a request from the mbatchd and re-execute itself. This permits the sbatchd binary to be updated. This operation will fail if no sbatchd is running on the specified host.

HOST_SHUTDOWN

The sbatchd on the host will exit.

***message**	Message attached by the administrator.

RETURN VALUES

int:0	The function was successful.
int:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_reconfig()`

Equivalent line command

none

Files

`lsf.conf`

lsb_hostgrpinfo()

Returns LSF host group membership.

DESCRIPTION

`lsb_hostgrpinfo()` gets LSF host group membership.

LSF host group is defined in the configuration file `lsb.hosts`.

The storage for the array of `groupInfoEnt` structures will be reused by the next call.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct groupInfoEnt *lsb_hostgrpinfo (char **groups,
                     int *numGroups, int options)

struct groupInfoEnt {
    char *group;
    char *memberList;
    char *adminMemberList;
    int  numUserShares;
    struct userShares  *userShares;
    int  options;
    char *pattern;
    char *neg_pattern;
    int  cu_type;
};
```

PARAMETERS

****groups**	An array of group names.
***numGroups**	The number of group names. `*numGroups` will be updated to the actual number of groups when this call returns.
options	The bitwise inclusive OR of some of the following flags:

GRP_RECURSIVE

Expand the group membership recursively. That is, if a member of a group is itself a group, give the names of its members recursively, rather than its name, which is the default.

GRP_ALL

Get membership of all groups.

groupInfoEnt structure fields

group	Group name.
memberList	ASCII list of member names.

RETURN VALUES

adminMemberList	ASCII list of admin member names.
numUserShares	The number of users with shares.
userShares	The user shares representation.
options	The bitwise inclusive OR of some of the following: **GRP_NO_CONDENSE_OUTPUT** 0x01 Group output is in regular (uncondensed) format. **GRP_CONDENSE_OUTPUT** 0x02 Group output is in condensed format. **GRP_HAVE_REG_EXP** 0x04 **GRP_SERVICE_CLASS** 0x08 Group is a service class. **GRP_IS_CU** 0x10 Group is a compute unit.
pattern	Host membership pattern.
neg_pattern	Negation membership pattern.
cu_type	Compute unit type.

RETURN VALUES

array:groupInfoEnt	On success, returns an array of `groupInfoEnt` structures which hold the group name and the list of names of its members. If a member of a group is itself a group (i.e., a subgroup), then a '/' is appended to the name to indicate this. `*numGroups` is the number of `groupInfoEnt` structures returned.
char:NULL	Function failed.

ERRORS

On failure, returns `NULL` and sets `lsberrno` to indicate the error. If there are invalid groups specified, the function returns the groups up to the invalid ones and then sets `lsberrno` to `LSBE_BAD_GROUP`, which means that the specified `(*groups)[*numGroups]` is not a group known to the LSF system. If the first group specified is invalid, the function returns `NULL`.

SEE ALSO

Related APIs

`lsb_usergrpinfo()`

Equivalent line command

none

Files

`$LSB_CONFDIR/cluster_name/lsb.hosts`

```
$LSB_CONFDIR/cluster_name/lsb.users
```

lsb_hostinfo()

Returns information about job server hosts.

DESCRIPTION

`lsb_hostinfo()` returns information about job server hosts.

The `hostInfoEnt` structure contains the following fields:

SYNOPSIS

```
#include <lsf/lsbatch.h>

struct hostInfoEnt *lsb_hostinfo(char **hosts, int *numHosts)
struct hostInfoEnt {
   char   *host;
   int     hStatus;
   int    *busySched;
   int    *busyStop;
   float  cpuFactor;
   int    nIdx;
   float  *load;
   float  *loadSched;
   float  *loadStop;
   char   *windows;
   int    userJobLimit;
   int    maxJobs;
   int    numJobs;
   int    numRUN;
   int    numSSUSP;
   int    numUSUSP;
   int    mig;
   int    attr;
   float  *realLoad;
   int    numRESERVE;
   int    chkSig;
   float  cnsmrUsage;
   float  prvdrUsage;
   float  cnsmrAvail;
   float  prvdrAvail;
   float  maxAvail;
   float  maxExitRate;
   float  numExitRate;
   char   *hCtrlMsg;
};
```

PARAMETERS

****hosts**

An array of host or cluster names.

***numHosts**

The number of host names.

To get information on all hosts, set `*numHosts` to 0; `*numHosts` will be set to the actual number of `hostInfoEnt` structures when this call returns.

If `*numHosts` is 1 and `hosts` is `NULL`, information on the local host is returned.

host

The name of the host.

hStatus

The status of the host. It is the bitwise inclusive OR of some of the following:

HOST_STAT_BUSY

The host load is greater than a scheduling threshold. In this status, no new job will be scheduled to run on this host.

HOST_STAT_WIND

The host dispatch window is closed. In this status, no new job will be accepted.

HOST_STAT_DISABLED

The host has been disabled by the LSF administrator and will not accept jobs. In this status, no new job will be scheduled to run on this host.

HOST_STAT_LOCKED

The host is locked by a exclusive task. In this status, no new job will be scheduled to run on this host.

HOST_STAT_FULL

The host has reached its job limit. In this status, no new job will be scheduled to run on this host.

HOST_STAT_UNREACH

The sbatchd on this host is unreachable.

HOST_STAT_UNAVAIL

The LIM and sbatchd on this host are unavailable.

HOST_STAT_UNLICENSED

The host does not have an LSF license.

HOST_STAT_NO_LIM

The host is running an sbatchd but not a LIM.

HOST_STAT_EXCLUSIVE

The host is running an sbatchd but not a LIM.

HOST_STAT_LOCKED_MASTER

LIM locked by master LIM.

HOST_STAT_REMOTE_DISABLED

Close a remote lease host. This flag is used together with HOST_STAT_DISABLED.

HOST_STAT_LEASE_INACTIVE

Close a remote lease host due to the lease is renewing or terminating.

HOST_STAT_DISABLED_RES

Host is disabled by RES.

HOST_STAT_LOCKED_EGO

The host is disabled by RMS.

HOST_CLOSED_BY_ADMIN

If none of the above hold, `hStatus` is set to `HOST_STAT_OK` to indicate that the host is ready to accept and run jobs.

busySched & busyStop

If `hStatus` is `HOST_STAT_BUSY`, these indicate the host `loadSched` or `loadStop` busy reason. If none of the thresholds have been exceeded, the value is `HOST_BUSY_NOT`. Otherwise the value is the bitwise inclusive OR of some of the following:

HOST_BUSY_R15S

The 15 second average CPU run queue length is too high.

HOST_BUSY_R1M

The 1 minute average CPU run queue length is too high.

HOST_BUSY_R15M

The 15 minute average CPU run queue length is too high.

HOST_BUSY_UT

The CPU utilization is too high.

HOST_BUSY_PG

The paging rate is too high.

HOST_BUSY_IO

The I/O rate is too high.

HOST_BUSY_LS

There are too many login sessions.

HOST_BUSY_IT

The host has not been idle long enough.

HOST_BUSY_TMP

There is not enough free space in the file system containing /tmp.

HOST_BUSY_SWP

There is not enough free swap space.

HOST_BUSY_MEM

There is not enough free memory.

The external load indices are designated by the constants from `B 1 << HOST_BUSY_MEM + 1 to 1 << nIdx - 1`. The names of these indices can be obtained from `ls_info()`.

cpuFactor

The host CPU factor used to scale CPU load values to account for differences in CPU speeds. The faster the CPU, the larger the CPU factor.

nIdx

The number of load indices in the `load`, `loadSched` and `loadStop` arrays.

load

Load information array on a host. This array gives the load information that is used for scheduling batch jobs. This load information is the effective load information from `ls_loadofhosts()` (see `ls_loadofhosts()`) plus the load reserved for running jobs (see `lsb.queues` for details on resource reservation). The `load` array is indexed the same as `loadSched` and `loadStop` (see `loadSched` and `loadStop` below).

loadSched & loadStop

The `loadSched` and `loadStop` arrays control batch job scheduling, suspension, and resumption.

The values in the `loadSched` array specify the scheduling thresholds for the corresponding load indices. Only if the current values of all specified load indices of this host are within (below or above, depending on the meaning of the load index) the corresponding thresholds of this host, will jobs be scheduled to run on this host.

Similarly, the values in the `loadStop` array specify the stop thresholds for the corresponding load indices. If any of the load index values of the host goes beyond its stop threshold, the job will be suspended.

The `loadSched` and `loadStop` arrays are indexed by the following constants:

R15S

15-second average CPU run queue length.

R1M

1-minute average CPU run queue length.

R15M

15-minute average CPU run queue length.

UT

CPU utilization over the last minute.

PG

Average memory paging rate, in pages per second.

IO

Average disk I/O rate, in KB per second.

LS

Number of current login users.

IT

	Idle time of the host in minutes. **TMP** The amount of free disk space in the file system containing /tmp, in MB. **SWP** The amount of swap space available, in MB. **MEM** The amount of available user memory on this host, in MB.
windows	One or more time windows in a week during which batch jobs may be dispatched to run on this host . The default is no restriction, or always open (i.e., 24 hours a day, seven days a week). These windows are similar to the dispatch windows of batch job queues. See `lsb_queueinfo()`.
userJobLimit	The maximum number of job slots any user is allowed to use on this host.
maxJobs	The maximum number of job slots that the host can process concurrently.
numJobs	The number of job slots running or suspended on the host.
numRUN	The number of job slots running on the host.
numSSUSP	The number of job slots suspended by the batch daemon on the host.
numUSUSP	The number of job slots suspended by the job submitter or the LSF system administrator.
mig	The migration threshold in minutes after which a suspended job will be considered for migration.
attr	The host attributes; the bitwise inclusive OR of some of the following: **H_ATTR_CHKPNTABLE** This host can checkpoint jobs. **H_ATTR_CHKPNT_COPY** This host provides kernel support for checkpoint copy.
realLoad	The effective load of the host.
numRESERVE	The number of job slots reserved by LSF for the PEND jobs.
chkSig	If attr has an `H_ATTR_CHKPNT_COPY` attribute, chkSig is set to the signal which triggers checkpoint and copy operation on the host. Otherwise, chkSig is set to the signal which triggers checkpoint operation on the host.
cnsmrUsage	The number of resources used by the consumer.
prvdrUsage	The number of resources used by the provider.
cnsmrAvail	The number of resources available for the consumer to use.
prvdrAvail	The number of resources available for the provider to use.
maxAvail	The maximum number resources available in total.
maxExitRate	The job exit rate threshold on the host.
***hCtrlMsg**	AdminAction: host control message.

RETURN VALUES

array:hostInfoEnt	On success, return an array of `hostInfoEnt` structures which hold the host information and sets `*numHosts` to the number of `hostInfoEnt` structures.
char:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error. If `lsberrno` is `LSBE_BAD_HOST`, `(*hosts)[*numHosts]` is not a host known to the batch system. Otherwise, if `*numHosts` is less than its original value, `*numHosts` is the actual number of hosts found.

SEE ALSO

Related APIs

`lsb_hostinfo_ex()`

`ls_info()` - Returns a pointer to an `lsInfo` structure

`ls_loadofhosts()`

`lsb_queueinfo()` - Get information about job queues

`lsb_userinfo()` - Get information about users and user groups

Equivalent line command

`bhosts`

Files

`$LSB_CONFDIR/cluster_name/lsb.hosts`

lsb_hostinfo_cond()

Returns condensed information about job server hosts.

DESCRIPTION

`lsb_hostinfo_cond()` returns condensed information about job server hosts. While `lsb_hostinfo()` returns specific information about individual hosts, `lsb_hostinfo_cond()` returns the number of jobs in each state within the entire host group. The `condHostInfoEnt` structure contains counters that indicate how many hosts are in the `ok`, `busy`, `closed`, `full`, `unreach`, and `unavail` states and an array of `hostInfoEnt` structures that indicate the status of each host in the host group.

SYNOPSIS

```
#include <lsf/lsbatch.h>
```

```
struct condHostInfoEnt * lsb_hostinfo_cond
                        (char **hosts, int *numHosts,
                         char *resReq, int options)

struct condHostInfoEnt {
    char *name;
    int howManyOk;
    int howManyBusy;
    int howManyClosed;
    int howManyFull;
    int howManyUnreach;
    int howManyUnavail;
    struct hostInfoEnt *hostInfo;
};
```

PARAMETERS

****hosts**	An array of host names belonging to the host group.
***numHosts**	The number of host names in the host group. To get information on all hosts in the host group, set *numHosts to 0; *numHosts will be set to the actual number of hostInfoEnt structures in the host group when this call returns.
***resReq**	Any resource requirements called with the function.
options	Any options called with the function.

RETURN VALUES

***condHostInfoEnt**	The condHostInfoEnt structure contains condensed information about the status of job server hosts in the host group. If there is no condensed host group matching the specified host, `*name` is set to `NULL` and `*hostInfo` contains specific host information to be displayed instead of the condensed host group information.

ERRORS

If the function fails, lsberrno is set to indicate the error.

SEE ALSO

Related APIs:

`lsb_hostinfo()` - Returns information about job server hosts

lsb_hostinfo_ex()

Returns informaton about job server hosts that satisfy specified resource requirements.

DESCRIPTION

`lsb_hostinfo_ex()` returns information about job server hosts that satisfy the specified resource requirements.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct hostInfoEnt *lsb_hostinfo_ex(char **hosts,
                    int *numHosts, char *resReq, int options)
struct hostInfoEnt {
    char   *host;
    int    hStatus;
    int    *busySched;
    int    *busyStop;
    float  cpuFactor;
    int    nIdx;
    float  *load;
    float  *loadSched;
    float  *loadStop;
    char   *windows;
    int    userJobLimit;
    int    maxJobs;
    int    numJobs;
    int    numRUN;
    int    numSSUSP;
    int    numUSUSP;
    int    mig;
    int    attr;
    float  *realLoad;
    int    numRESERVE;
    int    chkSig;
    float  cnsmrUsage;
    float  prvdrUsage;
    float  cnsmrAvail;
    float  prvdrAvail;
    float  maxAvail;
    float  maxExitRate;
    float  numExitRate;
    char   *hCtrlMsg;
};
```

PARAMETERS

****hosts** An array of host or cluster names.

***numHosts** The number of host names.

To get information on all hosts, set `*numHosts` to 0; `*numHosts` will be set to the actual number of `hostInfoEnt` structures when this call returns.

If `*numHosts` is 1 and `hosts` is `NULL`, information on the local host is returned.

***resReq** Resource requirements.

If this option is specified, then only host information for those hosts that satisfy the resource requirements is returned. Returned hosts are sorted according to the load on the `resource()` given in resReq, or by default according to CPU and paging load.

options Options is reserved for the future use.

The `hostInfoEnt` structure contains the following fields:

host The name of the host.

hStatus The status of the host. It is the bitwise inclusive OR of some of the following:

HOST_STAT_BUSY

The host load is greater than a scheduling threshold. In this status, no new job will be scheduled to run on this host.

HOST_STAT_WIND

The host dispatch window is closed. In this status, no new job will be accepted.

HOST_STAT_DISABLED

The host has been disabled by the LSF administrator and will not accept jobs. In this status, no new job will be scheduled to run on this host.

HOST_STAT_LOCKED

The host is locked by a exclusive task. In this status, no new job will be scheduled to run on this host.

HOST_STAT_FULL

The host has reached its job limit. In this status, no new job will be scheduled to run on this host.

HOST_STAT_UNREACH

The sbatchd on this host is unreachable.

HOST_STAT_UNAVAIL

The LIM and sbatchd on this host are unavailable.

HOST_STAT_UNLICENSED

The host does not have an LSF license.

HOST_STAT_NO_LIM

The host is running an sbatchd but not a LIM.

HOST_STAT_EXCLUSIVE

The host is running an sbatchd but not a LIM.

HOST_STAT_LOCKED_MASTER

LIM locked by master LIM.

HOST_STAT_REMOTE_DISABLED

Close a remote lease host. This flag is used together with HOST_STAT_DISABLED.

HOST_STAT_LEASE_INACTIVE

Close a remote lease host due to the lease is renewing or terminating.

HOST_STAT_DISABLED_RES

Host is disabled by RES.

HOST_STAT_LOCKED_EGO

The host is disabled by RMS.

HOST_CLOSED_BY_ADMIN

If none of the above hold, `hStatus` is set to `HOST_STAT_OK` to indicate that the host is ready to accept and run jobs.

`busySched` **&** `busyStop`

If `hStatus` is `HOST_STAT_BUSY`, these indicate the host `loadSched` or `loadStop` busy reason. If none of the thresholds have been exceeded, the value is `HOST_BUSY_NOT`. Otherwise the value is the bitwise inclusive OR of some of the following:

HOST_BUSY_R15S

The 15 second average CPU run queue length is too high.

HOST_BUSY_R1M

The 1 minute average CPU run queue length is too high.

HOST_BUSY_R15M

The 15 minute average CPU run queue length is too high.

HOST_BUSY_UT

The CPU utilization is too high.

HOST_BUSY_PG

The paging rate is too high.

HOST_BUSY_IO

The I/O rate is too high.

HOST_BUSY_LS

There are too many login sessions.

HOST_BUSY_IT

The host has not been idle long enough.

HOST_BUSY_TMP

There is not enough free space in the file system containing /tmp.

HOST_BUSY_SWP

There is not enough free swap space.

HOST_BUSY_MEM

There is not enough free memory.

The external load indices are designated by the constants from `B 1 << HOST_BUSY_MEM + 1 to 1 << nIdx - 1`. The names of these indices can be obtained from `ls_info()`.

cpuFactor The host CPU factor used to scale CPU load values to account for differences in CPU speeds. The faster the CPU, the larger the CPU factor.

nIdx The number of load indices in the `load`, `loadSched` and `loadStop` arrays.

load Load information array on a host. This array gives the load information that is used for scheduling batch jobs. This load information is the effective load information from `ls_loadofhosts()` (see `ls_loadofhosts()`) plus the load reserved for running jobs (see `lsb.queues` for details on resource reservation). The `load` array is indexed the same as `loadSched` and `loadStop` (see `loadSched` and `loadStop` below).

loadSched & loadStop The `loadSched` and `loadStop` arrays control batch job scheduling, suspension, and resumption.

The values in the `loadSched` array specify the scheduling thresholds for the corresponding load indices. Only if the current values of all specified load indices of this host are within (below or above, depending on the meaning of the load index) the corresponding thresholds of this host, will jobs be scheduled to run on this host.

Similarly, the values in the `loadStop` array specify the stop thresholds for the corresponding load indices. If any of the load index values of the host goes beyond its stop threshold, the job will be suspended.

The `loadSched` and `loadStop` arrays are indexed by the following constants:

R15S

15-second average CPU run queue length.

R1M

1-minute average CPU run queue length.

R15M

15-minute average CPU run queue length.

UT

CPU utilization over the last minute.

PG

Average memory paging rate, in pages per second.

IO

Average disk I/O rate, in KB per second.

LS

Number of current login users.

IT

	Idle time of the host in minutes. **TMP** The amount of free disk space in the file system containing /tmp, in MB. **SWP** The amount of swap space available, in MB. **MEM** The amount of available user memory on this host, in MB.
windows	One or more time windows in a week during which batch jobs may be dispatched to run on this host . The default is no restriction, or always open (i.e., 24 hours a day, seven days a week). These windows are similar to the dispatch windows of batch job queues. See `lsb_queueinfo()`.
userJobLimit	The maximum number of job slots any user is allowed to use on this host.
maxJobs	The maximum number of job slots that the host can process concurrently.
numJobs	The number of job slots running or suspended on the host.
numRUN	The number of job slots running on the host.
numSSUSP	The number of job slots suspended by the batch daemon on the host.
numUSUSP	The number of job slots suspended by the job submitter or the LSF system administrator.
mig	The migration threshold in minutes after which a suspended job will be considered for migration.
attr	The host attributes; the bitwise inclusive OR of some of the following: **H_ATTR_CHKPNTABLE** This host can checkpoint jobs. **H_ATTR_CHKPNT_COPY** This host provides kernel support for checkpoint copy.
realLoad	The effective load of the host.
numRESERVE	The number of job slots reserved by LSF for the PEND jobs.
chkSig	If attr has an `H_ATTR_CHKPNT_COPY` attribute, chkSig is set to the signal which triggers checkpoint and copy operation on the host. Otherwise, chkSig is set to the signal which triggers checkpoint operation on the host. The storage for the array of `hostInfoEnt` structures will be reused by the next call.
cnsmrUsage	The number of resources used by the consumer.
prvdrUsage	The number of resources used by the provider.
cnsmrAvail	The number of resources available for the consumer to use.
prvdrAvail	The number of resources available for the provider to use.
maxAvail	The maximum number resources available in total.
maxExitRate	The job exit rate threshold on the host.
***hCtrlMsg**	AdminAction: host control message.

RETURN VALUES

array:hostInfoEnt	On success, return an array of `hostInfoEnt` structures which hold the host information and sets `*numHosts` to the number of `hostInfoEnt` structures.
char:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error. If `lsberrno` is `LSBE_BAD_HOST`, `(*hosts)[*numHosts]` is not a host known to the batch system. Otherwise, if `*numHosts` is less than its original value, `*numHosts` is the actual number of hosts found.

SEE ALSO

Related APIs

`ls_info()` - Returns a pointer to an `lsInfo` structure

`ls_loadofhosts()`

`lsb_hostinfo()` - Get information about job server hosts

`lsb_queueinfo()` - Get information about job queues

`lsb_userinfo()` - Get information about users and user groups

Equivalent line command

none

Files

`$LSB_CONFDIR/cluster_name/lsb.hosts`

lsb_hostpartinfo()

Returns informaton about host partitions.

DESCRIPTION

`lsb_hostpartinfo()` gets information about host partitions.

The `hostPartInfoEnt` structure has the following fields:

hostPart The name of the host partition.

hostList A blank-separated list of names of hosts and host groups which are members of the host partition. The name of a host group has a '/' appended. (See `lsb_hostgrpinfo()`.)

numUsers The number of users in this host partition. i.e., the number of `hostPartUserInfo` structures.

users An array of `hostPartUserInfo` structures which hold information on users in this host partition.

The `hostPartUserInfo` structure has the following fields:

user The user name or user group name. (See `lsb_userinfo()` and `lsb_usergrpinfo()`.)

shares The number of shares assigned to the user or user group, as configured in the file `lsb.hosts`. (See `lsb.hosts`.)

numStartJobs The number of job slots belonging to the user or user group that are running or suspended in the host partition.

numReserveJobs The number of job slots that are reserved for the `PEND` jobs belonging to the user or user group in the host partition.

histCpuTime The normalized CPU time accumulated in the host partition during the recent period by finished jobs belonging to the user or user group. The period may be configured in the file `lsb.params` (see `lsb.params`), with a default value of five (5) hours.

priority The priority of the user or user group to use the host partition. Bigger values represent higher priorities. Jobs belonging to the user or user group with the highest priority are considered first for dispatch when resources in the host partition are being contended for. In general, a user or user group with more `shares`, fewer `numStartJobs` and less `histCpuTime` has higher priority.

The storage for the array of `hostPartInfoEnt` structures will be reused by the next call.

runTime The time unfinished jobs spend in the RUN state.

shareAdjustment The fairshare adjustment value from the fairshare plugin (`libfairshareadjust.*`). The adjustment is enabled and weighted by setting the value of `FAIRSHARE_ADJUSTMENT_FACTOR` in `lsb.params`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct hostPartInfoEnt *lsb_hostpartinfo (char **hostParts,
                        int *numHostParts)

struct hostPartInfoEnt {
    char hostPart[MAX_LSB_NAME_LEN];
    char *hostList;
    int numUsers;
    struct hostPartUserInfo *users;
};

struct hostPartUserInfo {
    char user[MAX_LSB_NAME_LEN];
    int shares;
    float priority;
    int numStartJobs;
    float histCpuTime;
    int numReserveJobs;
    int runTime;
    float shareAdjustment;
};
```

PARAMETERS

****hostParts**	An array of host partition names.
***numHostHosts**	The number of host partition names. To get information on all host partitions, set `hostParts` to `NULL`; `*numHostParts` will be the actual number of host partitions when this call returns.

RETURN VALUES

array:hostPartInfoEnt	On success, returns an array of `hostPartInfoEnt` structures which hold information on the host partitions, and sets `*numHostParts` to the number of `hostPartInfoEnt` structures.
char:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error. If `lsberrno` is `LSBE_BAD_HPART`, `(*hostParts)[*numHostParts]` is not a host partition known to the LSF system. Otherwise, if `*numHostParts` is less than its original value, `*numHostParts` is the actual number of host partitions found.

SEE ALSO

Related APIs

`lsb_usergrpinfo()`

`lsb_hostgrpinfo()`

SEE ALSO

Equivalent line command

none

Files

`$LSB_CONFDIR/`*cluster*`_name/lsb.hosts`

lsb_init()

Initializes the LSF batch library (LSBLIB), and gets the configuration environment.

DESCRIPTION

You must use lsb_init() before any other LSBLIB library routine in your application.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_init(char *appname)
```

PARAMETERS

***appName** The name of your application.

If appName holds the name of your application, a logfile with the same name as your application receives LSBLIB transaction information.

If appName is NULL, the logfile $LSF_LOGDIR/bcmd receives LSBLIB transaction information.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

If the function fails, lsberrno is set to indicate the error.

SEE ALSO

Related APIs:

none

Equivalent line command

none

Files:

none

lsb_jsdl2submit()

Accepts a JSDL job submission file as input and converts the file for use with LSF.

DESCRIPTION

`lsb_jsdl2submit()` converts parameters specified in the JSDL file and merges them with the other command line and job script options. The merged submit request is then sent to `mbatchd` for processing.

Code must link to `LSF_LIBDIR/libbat.jsdl.lib`

SYNOPSIS

```
extern int lsb_jsdl2submit(struct submit* req, char *template);
```

PARAMETERS

submit* req	Reads the specified JSDL options and maps them to the `submitReq` structure. Code must specify either `jsdl` or `jsdl_strict`.
***template**	The default template, which contains all of the `bsub` submission options.

RETURN VALUES

0	Function completed successfully.
-1	Function failed.

ERRORS

On failure, sets `lsberrno` to indicate the error.

SEE ALSO

Related API

`lsb_submit()` - Submits or restarts a job in the batch system

`lsb_modify()` - Modifies a submitted job's parameters

Equivalent line command

`bsub` with options

Files

`LSF_LIBDIR/jsdl.xsd`

`LSF_LIBDIR/jsdl-posix.xsd`

`LSF_LIBDIR/jsdl-lsf.xsd`

lsb_killbulkjobs()

Kills bulk jobs as soon as possible.

DESCRIPTION

Use `lsb_killbulkjobs()` to kill bulk jobs on a local host immediately, or to kill other jobs as soon as possible. If `mbatchd` rejects the request, it issues `NULL` as the reservation ID.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_killbulkjobs(struct signalBulkJobs *s)

struct signalBulkJobs {
    int signal;
    int njobs;
    LS_LONG_INT *jobs;
    int flags;
};
```

PARAMETERS

***signalBulkJobs**	The signal to a group of jobs.

RETURN VALUES

integer:0	The bulk jobs were successfully killed.
integer:-1	The bulk jobs were not killed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs:

none

Equivalent line command

```
bkill -b
```

Files:

none

lsb_launch()

Launch commands on remote hosts in parallel.

DESCRIPTION

`lsb_launch()` is a synchronous API call to allow source level integration with vendor MPI implementations. This API will launch the specified command (`argv`) on the remote nodes in parallel.

LSF must be installed before integrating your MPI implementation with `lsb_launch()`. The `lsb_launch()` API requires the full set of `liblsf.so`, `libbat.so` (or `liblsf.a`, `libbat.a`).

SYNOPSIS

```
#include <lsf/lsbatch.h>
int
lsb_launch (char** where, char** argv, int userOptions, char** envp)
```

PARAMETERS

where [IN] A NULL-terminated list of hosts. A task will be launched for each slot.

If this parameter is NULL then the environment variable LSB_MCPU_HOSTS will be used.

argv [IN] The command to be executed

userOptions [IN] Options to modify the behavior of `lsb_launch()`

Multiple option values can be specified. For example option values can be separated by OR (|):

```
lsb_launch(where, argv, LSF_DJOB_REPLACE_ENV | LSF_DJOB_DISABLE_STDIN, envp);
```

Valid options are:

- LSF_DJOB_DISABLE_STDIN—Disable standard input and redirect input from the special device `/dev/null`. This is equivalent to `blaunch -n`.
- LSF_DJOB_REPLACE_ENV—Replace existing enviornment variable values with `envp`.
- LSF_DJOB_NOWAIT—Non-blocking mode; the parallel job does not wait once all tasks start. This forces `lsb_launch()` not to wait for its tasks to finish.
- LSF_DJOB_STDERR_WITH_HOSTNAME—Display standard error messages with a corresponding host name where the message was generated. Cannot be specified with LSF_DJOB_NOWAIT.
- LSF_DJOB_STDOUT_WITH_HOSTNAME—Display standard output messages with a corresponding host name where the message was generated. Cannot be specified with LSF_DJOB_NOWAIT.
- LSF_DJOB_USE_LOGIN_SHELL—Launch commands through user's login shell.

- LSF_DJOB_USE_BOURNE_SHELL—Launch commands through Bourne shell (`/bin/sh`). If LSF_DJOB_USE_LOGIN_SHELL is also specified, LSF_DJOB_USE_LOGIN_SHELL is used.
- LSF_DJOB_STDERR—Separates stderr from stdout.

envp	[IN] A NULL-terminated list of environment variables specifying the environment to set for each task.

If `envp` is NULL, `lsb_launch()` uses the same environment used to start the first task on the first execution host. If non-NULL, `envp` values are appended to the environment used for the first task.

If the LSF_DJOB_REPLACE_ENV option is specified, `envp` entries will overwrite all existing environment values except those needed by LSF.

RETURN VALUES

> 0	Function was successful (the number of tasks launched).
< 0	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

none

Equivalent line command

`blaunch`

Files

none

lsb_limitInfo()

gets resource allocation limit configuration and dynamic usage information

DESCRIPTION

Displays current usage of resource allocation limits configured in Limit sections in `lsb.resources`:

- Configured limit policy name
- Users
- Queues
- Hosts
- Project names

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_limitInfo(limitInfoReq *req, limitInfoEnt **entRef, int * size, struct lsInfo*
lsInfo)
typedef struct _limitInfoReq {
char * name;
int consumerC;
limitConsumer  *consumerV;
} limitInfoReq;
typedef struct _limitConsumer {
consumerType type; char * name;
} limitConsumer
typedef struct _limitInfoEnt {
char * name; limitItem confInfo; int usageC;
limitItem   usageInfo;
} limitInfoEnt;
typedef struct _limitItem {
int consumerC; limitConsumer *consumerV; int resourceC; limitResource  *resourceV;
} limitItem;
typedef struct _ limitResource {
char * rsrcName;
int type;
float val;
} limitResource;
```

PARAMETERS

req	input, the user request for limit information
entRef	output, the limit information array
size	output, the size of the limit information array

_limitInfoReq The structure limitInfoReq contains the following fields:

name

Limit policy name given by the user.

jobid

Job ID of jobs with resource usage.

consumerC

consumerV

Consumer name, queue/host/user/project.

_limitConsumer The structure limitConsumer contains the following fields:

type

Consumer type:

- Queues per-queue
- Users and per-user
- Hosts and per-host
- Projects and per-project

name

Consumer name

_limitInfoEnt The structure limitInfoEnt contains the following fields:

name

limit policy name given by the user

confInfo

limit configuration

usageC

size of limit dynamic usage info array

usageInfo

limit dynamic usage info array

_limitItem The structure limitItem contains the following fields:

consumerC

consumerV

queue/host/user/project

resourceC

resourceV

_limitResource The structure limitResource contains the following fields:

rsrcName

val

RETURN VALUES

LSBE_NO_ERROR	Suceess; others, errors happened.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`lsb_freeLimitInfoEnt()`

Equivalent command

`blimits`

Files

`lsb.queues`, `lsb.users`, `lsb.hosts`, `lsb.resources`

lsb_mig()

Migrates a job from one host to another.

DESCRIPTION

`lsb_mig()` migrates a job from one host to another.

The `submig` structure contains the following fields:

jobId	The job ID of the job to be migrated.
options	See `lsb_submit()`.
numAskedHosts	The number of hosts supplied as candidates for migration.
askedHosts	An array of pointers to the names of candidate hosts for migration.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_mig(struct submig *mig, int *badHostIdx)

struct submig {
    LS_LONG_INT jobId;
    int options;
    int numAskedHosts;
    char **askedHosts;
};
```

PARAMETERS

***mig**	The job to be migrated.
***badHostIdx**	If the call fails, `(**askedHosts)[*badHostIdx]` is not a host known to the LSF system.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error and `badHostIdx` indicates which `askedHost` is not acceptable.

SEE ALSO

Related APIs

`lsb_submit()` - Submits or restarts a job in the batch system

SEE ALSO

Equivalent line command

none

Files

`${LSF_ENVDIR}/lsf.conf`

lsb_modify()

Modifies a submitted job's parameters.

DESCRIPTION

`lsb_modify()` allows for the modification of a submitted job's parameters.

SYNOPSIS

```
#include <lsf/lsbatch.h>
LS_LONG_INT lsb_modify (struct submit *jobSubReq,
                        struct submitReply *jobSubReply, int jobId)
struct submit {
    int     options;
    int     options2;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    int     numProcessors;
    char    *dependCond;
    char    *timeEvent;
    time_t  beginTime;
    time_t  termTime;
    int     sigValue;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *newCommand;
    time_t  chkpntPeriod;
    char    *chkpntDir;
    int     nxf;
    struct xFile *xf;
    char    *preExecCmd;
    char    *mailUser;
    int     delOptions;
    int     delOptions2;
    char    *projectName;
    int     maxNumProcessors;
    char    *loginShell;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *jobGroup;
    char    *sla;
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    int     jsdlFlag;
    char    *jsdlDoc;
    void    *correlator;
    char    *apsString;
    char    *postExecCmd;
    char    *cwd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
    char    *notifyCmd;
};
struct submitReply {
    char *queue;
    LS_LONG_INT badJobId;
    char *badJobName;
    int badReqIndx;
};
```

PARAMETERS

***jobSubReq** Describes the requirements for job modification to the batch system. A job that does not meet these requirements is not submitted to the batch system and an error is returned.

See lsb_submit() on page 377 for descriptions of the `submit` structure fields.

***jobSubReply** Describes the results of the job modification to the batch system.

See lsb_submit() on page 377 for descriptions of the `submitReply` structure fields.

***jobId** The job to be modified. If an element of a job array is to be modified, use the array form `jobID[i]` where `jobID` is the job array name, and `i` is the index value.

RETURN VALUES

character:job ID	The function was successful, and sets the `queue` field of *jobSubReply* to the name of the queue that the job was submitted to.
integer:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`lsb_submit()` - Submits or restarts a job in the batch system

`ls_info()` - Returns a pointer to an `lsInfo` structure

`ls_task()`

`lsb_queueinfo()` - Returns information about batch queues

Equivalent line command

`bmod`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_modreservation()

Modifies an advance reservation.

DESCRIPTION

Use `lsb_modreservation()` to modify an advance reservation. `mbatchd` receives the modification request and modifies the reservation with the specified reservation ID.

SYNOPSIS

```
#include <lsf/lsbatch
int lsb_modreservation(struct modRsvRequest *request)
struct modRsvRequest {
    char    *rsvId;
    struct addRsvRequest   fieldsFromAddReq;
    char    *disabledDuration;
};
struct addRsvRequest {
   int     options;
    char    *name;
    struct {
    int     minNumProcs;
    int     maxNumProcs;
    } procRange;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    char    *timeWindow;
    rsvExecCmd_t  *execCmd;
    char   *desc;
    char    *rsvName;
};
```

PARAMETERS

***rsvId**	Reservation ID of the reservation that you wish to modify.
addRsvRequest	Fields in the reservation request addRsvRequest structure that you wish to modify.
***disabledDuration**	Disabled time duration.

RETURN VALUES

integer:0	The reservation was modified successfully.
integer:-1	The reservation modification failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_addreservation()` - Makes a reservation

`lsb_removereservation()` - Removes a reservation

`lsb_reservationinfo()` - Retrieves reservation information

Equivalent line command

`brsvmod`

Files:

none

lsb_movejob()

Changes the position of a pending job in a queue.

DESCRIPTION

Use lsb_movejob() to move a pending job to a new position that you specify in a queue. Position the job in a queue by first specifying the job ID. Next, count, beginning at 1, from either the top or the bottom of the queue, to the position you want to place the job.

To position a job at the top of a queue, choose the top of a queue parameter and a postion of 1.

To position a job at the bottom of a queue, choose the bottom of the queue parameter and a position of 1.

By default, LSF dispatches jobs in a queue in order of their arrival (e.g., first-come-first-served), subject to the availability of suitable server hosts.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_movejob (LS_LONG_INT jobId, int *position, int opCode)
```

PARAMETERS

jobId The job ID that the LSF system assigns to the job. If a job in a job array is to be moved, use the array form jobID[i] where jobID is the job array name, and `i` is the index value.

position The new position of the job in a queue. position must be a value of 1 or more.

opCode The top or bottom position of a queue.

TO_TOP

The top position of a queue.

TO_BOTTOM

The bottom position of a queue.

If an opCode is not specified for the top or bottom position, the function fails.

RETURN VALUES

integer:0 The function is successful.

integer:-1 The function failed.

ERRORS

If the function fails, lsberrno is set to indicate the error.

SEE ALSO

Related APIs:

`lsb_pendreason()` - Explains why a job is pending

Equivalent line command

```
btop
```

```
bbot
```

```
bjobs -q
```

Files:

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_openjobinfo()

Returns the number of jobs in the master batch daemon.

DESCRIPTION

`lsb_openjobinfo()` accesses information about pending, running and suspended jobs in the master batch daemon. Use `lsb_openjobinfo()` to create a connection to the master batch daemon. Next, use `lsb_readjobinfo()` to read job records. Close the connection using `lsb_closejobinfo()`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
```

```
int lsb_openjobinfo(LS_LONG_INT jobId, char *jobName,
                    char *userName, char *queueName, char *hostName,
                     int options)
```

PARAMETERS

`lsb_openjobinfo()` opens a connection with mbatchd and returns the total number of records in the connection on success.

jobId Passes information about jobs with the given job ID. If jobId is 0, `lsb_openjobinfo()` looks to another parameter to return information about jobs. If a member of a job array is to be passed, use the array form jobID[i] where jobID is the job array name, and `i` is the index value.

jobName Passes information about jobs with the given job name. If jobName is NULL, `lsb_openjobinfo()` looks to another parameter to return information about jobs.

userName Passes information about jobs submitted by the named user or user group, or by all users if user is all. If user is NULL, `lsb_openjobinfo()` assumes the user is invoking this call.

queueName Passes information about jobs belonging to the named queue. If queue is NULL, jobs in all the queues of the batch system are counted.

hostName Passes information about jobs on the named host, host group or cluster name. If host is NULL, jobs on all hosts of the batch system will be considered.

options `<lsf/lsbatch.h>` defines the following flags constructed from bits. Use the bitwise OR to set more than one flag.

ALL_JOB

Information about all jobs, including unfinished jobs (pending, running or suspended) and recently finished jobs. LSF remembers jobs finished within the preceding period. This period is set by the parameter CLEAN_PERIOD in the `lsb.params` file. The default is 3600 seconds (1 hour). (See `lsb.params`). The command line equivalent is `bjobs -a`.

DONE_JOB

Information about recently finished jobs.

PEND_JOB

Information about pending jobs.

SUSP_JOB

Information about suspended jobs.

CUR_JOB

Information about all unfinished jobs.

LAST_JOB

Information about the last submitted job.

RUN_JOB

Information about all running jobs

JOBID_ONLY

Information about JobId only.

HOST_NAME

Internal use only.

NO_PEND_REASONS

Exclude pending jobs.

JGRP_INFO

Information about job groups.

JGRP_RECURSIVE

Information about job group arrays.

JOB_ID_ONLY_ALL

Information about all jobs in the core.

ZOMBIE_JOB

Information about all zombie jobs.

TRANSPARENT_MC

Display remote jobs by submission jobid.

EXCEPT_JOB

Information about unfinished jobs that have triggered a job exception (overrun, underrun, idle).

MUREX_JOB

Information about all murex jobs.

TO_SYM_UA

Information about all jobs to Symphony UA.

SYM_TOP_LEVEL_ONLY

Internal use only.

JGRP_NAME

Information about job group structure.

COND_HOSTNAME

Uncondensed output for host groups. Option ignored in lsb_openjobinfo().

FROM_BJOBSCMD

Internal use only.

WITH_LOPTION

Internal use only.

APS_JOB

Jobs submitted to APS queue.

UGRP_INFO

Information about user group.

TIME_LEFT

Estimated time remaining based on the runtime estimate or runlimit.

FINISH_TIME

Estimated finish time based on the runtime estimate or runlimit.

COM_PERCENTAGE

Estimated completion percentage based on the runtime estimate or runlimit.

If options is 0, default to `CUR_JOB`.

RETURN VALUES

integer:value	The total number of records in the connection.
integer:-1	The function failed.

ERRORS

If the function fails, lsberrno is set to indicate the error.

SEE ALSO

Related APIs:

lsb_openjobinfo_a() - Provides the name and number of jobs and hosts in the master batch daemon

lsb_openjobinfo_a_ext() – Provides the name and number of jobs and hosts in the master batch daemon with additional host group information

lsb_openjobinfo_req() – Extensible API interface providing name, number of jobs and other information in the master batch daemon.

lsb_closejobinfo() - Closes a job information connection to the master batch daemon

lsb_readjobinfo() - Returns the next job information record in master batch daemon

lsb_readframejob() - Returns frame job information from the master batch daemon

Equivalent line command

```
bjobs
```

Files:

```
${LSF_ENVDIR}/lsf.conf
```

lsb_openjobinfo_a()

Provides the name and number of jobs and hosts in the master batch daemon.

DESCRIPTION

`lsb_openjobinfo_a()` provides more information on pending, running and suspended jobs than `lsb_openjobinfo()`. Use `lsb_openjobinfo_a()` to create a connection to the master batch daemon. Next, use `lsb_readjobinfo()` to read job records. Close the connection using `lsb_closejobinfo()`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct jobInfoHead *lsb_openjobinfo_a(LS_LONG_INT jobId,
                                      char *jobName,
                                      char *userName,
                                      char *queueName,
                                      char *hostName,
                                        int options)
```

PARAMETERS

`lsb_openjobinfo_a()` passes information about jobs based on the value of jobId, jobName, userName, queueName, or hostName. Only one parameter can be chosen. The other parameters must be `NULL` or 0.

jobId Passes information about jobs with the given job ID. If jobId is 0, `lsb_openjobinfo()` looks to another parameter to return information about jobs. If information about a member of a job array is to be passed, use the array form jobID[i] where jobID is the job array name, and i is the index value.

jobName Passes information about jobs with the given job name. If jobName is `NULL`, `lsb_openjobinfo()` looks to another parameter to return information about jobs.

userName Passes information about jobs submitted by the named user or user group, or by all users if userName is all. If userName is `NULL`, `lsb_openjobinfo_a()` assumes the user is invoking this call.

queueName Passes information about jobs belonging to the named queue. If queueName is `NULL`, jobs in all queues of the batch system will be considered.

hostName Passes information about jobs on the named host, host group or cluster name. If hostName is `NULL`, jobs on all hosts of the batch system will be considered.

options `<lsf/lsbatch.h>` defines the following flags constructed from bits. Use the bitwise `OR` to set more than one flag.

ALL_JOB

Information about all jobs, including unfinished jobs (pending, running or suspended) and recently finished jobs. LSF remembers jobs finished within the preceding period. This period is set by the parameter `CLEAN_PERIOD` in the `lsb.params` file. The default is 3600 seconds (1 hour). (See `lsb.params`). The command line equivalent is `bjobs -a`.

CUR_JOB

Information about all unfinished jobs.

DONE_JOB

Information about recently finished jobs.

PEND_JOB

Information about pending jobs.

SUSP_JOB

Information about suspended jobs.

LAST_JOB

Information about the last submitted job.

If options is 0, default to `CUR_JOB`.

RETURN VALUES

struct jobInfoHeadExt *

On success, returns an array of data type `struct jobInfoHeadExt *`, which represents the name and number of jobs and hosts in the master batch daemon with the host group information.

integer:-1 The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_openjobinfo()` - Opens a job information connection to the master batch daemon

`lsb_closejobinfo()` - Closes a job information connection to the master batch daemon

`lsb_readjobinfo()` - Returns the next job information record in master batch daemon

`lsb_readframejob()` - Returns frame job information from the master batch daemon

Equivalent line command

`bjobs`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

`lsb.params`

lsb_openjobinfo_a_ext()

Returns the name and number of jobs and hosts in the master batch daemon with additional host group information.

DESCRIPTION

`lsb_openjobinfo_a_ext()` is run from `lsb_openjobinfo_a()` using the same parameters and provides the same information as `lsb_openjobinfo_a()`, but with additional host group information.

SYNOPSIS

```
#include <lsf/lsbatch.h>
```

```
struct jobInfoHeadExt *
lsb_openjobinfo_a_ext (LS_LONG_INT jobId, char *jobName,
                       char *userName, char *queueName,
                       char *hostName, int options)
```

PARAMETERS

`lsb_openjobinfo_a_ext()` passes information about jobs based on the value of jobId, jobName, userName, queueName, or hostName. Only one parameter can be chosen. The other parameters must be `NULL` or 0.

jobId Passes information about jobs with the given job ID. If jobId is 0, `lsb_openjobinfo_a_ext()` looks to another parameter to return information about jobs. If information about a member of a job array is to be passed, use the array form jobID[i] where jobID is the job array name, and i is the index value.

jobName Passes information about jobs with the given job name. If jobName is `NULL`, `lsb_openjobinfo_a_ext()` looks to another parameter to return information about jobs.

userName Passes information about jobs submitted by the named user or user group, or by all users if userName is all. If userName is `NULL`, `lsb_openjobinfo_a_ext()` assumes the user is invoking this call.

queueName Passes information about jobs belonging to the named queue. If queueName is `NULL`, jobs in all queues of the batch system will be considered.

hostName Passes information about jobs on the named host, host group or cluster name. If hostName is `NULL`, jobs on all hosts of the batch system will be considered.

options `<lsf/lsbatch.h>` defines the following flags constructed from bits. Use the bitwise `OR` to set more than one flag.

ALL_JOB

Information about all jobs, including unfinished jobs (pending, running or suspended) and recently finished jobs. LSF remembers jobs finished within the preceding period. This period is set by the parameter `CLEAN_PERIOD` in the `lsb.params` file. The default is 3600 seconds (1 hour). (See `lsb.params`). The command line equivalent is `bjobs -a`.

CUR_JOB

Information about all unfinished jobs.

DONE_JOB

Information about recently finished jobs.

PEND_JOB

Information about pending jobs.

SUSP_JOB

Information about suspended jobs.

LAST_JOB

Information about the last submitted job.

If options is 0, default to `CUR_JOB`.

RETURN VALUES

struct jobInfoHeadExt *

On success, returns an array of data type `struct jobInfoHeadExt *`, which represents the name and number of jobs and hosts in the master batch daemon with the host group information.

integer:-1 The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_openjobinfo()` - Opens a job information connection to the master batch daemon

`lsb_openjobinfo_a()` - Returns the name and number of jobs and hosts in the master batch daemon

`lsb_closejobinfo()` - Closes a job information connection to the master batch daemon

`lsb_readjobinfo()` - Returns the next job information record in master batch daemon

`lsb_readframejob()` - Returns frame job information from the master batch daemon

Equivalent line command

`bjobs`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

`lsb.params`

lsb_openstream()

Open and create an lsb_stream file.

.

DESCRIPTION

lsb_openstream() opens the streamFile .

This API function is inside `liblsbstream.so`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_openstream(const struct lsbStream *params)
struct lsbStream {
    char   *streamFile;
    int    maxStreamSize;
    int    maxStreamFileNum;
    int    trace;
    int    (*trs)(const char *);
};
```

PARAMETERS

*** streamFile**	Pointer to the full path name of the stream file.
maxStreamSize	Maximium size of the stream file in bytes.
maxStreamFileNum	Maximium number of backup stream files.
trace	Set to 1 to enable tracing of the stream.
*** trs**	Pointer to a function that the library invokes, passing a trace buffer.

RETURN VALUES

integer:0	The function was successful.
integer:-1 or NULL	The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

lsb_closestream(): Close the stream file.

lsb_readstreamline(): Read a line from the stream file.

lsb_writestream(): Write an event to the stream file.

lsb_readstream(): Read from the stream file.

lsb_streamversion(): Version of the current event type supported by mbatchd.

Equivalent line command

None

Files

`lsb.params`

lsb_parameterinfo()

Returns information about the LSF cluster.

DESCRIPTION

`lsb_parameterinfo()` gets information about the LSF cluster.

The static storage for the parameterInfo structure is reused on the next call.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct parameterInfo *lsb_parameterinfo(char **names,
                      int *numUsers, int options)
struct parameterInfo {
    char *defaultQueues;
    char *defaultHostSpec;
    int  mbatchdInterval;
    int  sbatchdInterval;
    int  jobAcceptInterval;
    int  maxDispRetries;
    int  maxSbdRetries;
    int  preemptPeriod;
    int  cleanPeriod;
    int  maxNumJobs;
    float historyHours;
    int  pgSuspendIt;
    char *defaultProject;
    int  retryIntvl;
    int  nqsQueuesFlags;
    int  nqsRequestsFlags;
    int  maxPreExecRetry;
    int  eventWatchTime;
    float runTimeFactor;
    float waitTimeFactor;
    float runJobFactor;
    int  eEventCheckIntvl;
    int  rusageUpdateRate;
    int  rusageUpdatePercent;
    int  condCheckTime;
    int  maxSbdConnections;
    int  rschedInterval;
    int  maxSchedStay;
    int  freshPeriod;
    int  preemptFor;
    int  adminSuspend;
    int  userReservation;
    float cpuTimeFactor;
    int fyStart;
    int     maxJobArraySize;
    time_t  exceptReplayPeriod;
    int jobTerminateInterval;
    int disableUAcctMap;
    int enforceFSProj;
    int enforceProjCheck;
    int     jobRunTimes;
    int     dbDefaultIntval;
    int     dbHjobCountIntval;
    int     dbQjobCountIntval;
    int     dbUjobCountIntval;
    int     dbJobResUsageIntval;
    int     dbLoadIntval;
    int     dbJobInfoIntval;
    int     jobDepLastSub;
    int     maxJobNameDep;
    char   *dbSelectLoad;
    int     jobSynJgrp;
    char   *pjobSpoolDir;
    int     maxUserPriority;
    int     jobPriorityValue;
    int     jobPriorityTime;
    int     enableAutoAdjust;
    int     autoAdjustAtNumPend;
    float   autoAdjustAtPercent;
    int     sharedResourceUpdFactor;
    int     scheRawLoad;
    char   *jobAttaDir;
    int     maxJobMsgNum;
    int     maxJobAttaSize;
    int     mbdRefreshTime;
    int     updJobRusageInterval;
    char    *sysMapAcct;
    int     preExecDelay;
    int     updEventUpdateInterval;
    int     resourceReservePerSlot;
    int    maxJobId;
    char    *preemptResourceList;
    int     preemptionWaitTime;
```

SYNOPSIS

```
    int     maxAcctArchiveNum;
    int     acctArchiveInDays;
    int     acctArchiveInSize;
    float committedRunTimeFactor;
    int  enableHistRunTime;
#ifdef PS_SXNQS
    int   nqsUpdateInterval;
#endif
    int  mcbOlmReclaimTimeDelay;
    int  chunkJobDuration;
    int  sessionInterval;
    int  publishReasonJobNum;
    int  publishReasonInterval;
    int  publishReason4AllJobInterval;
    int  mcUpdPendingReasonInterval;
    int  mcUpdPendingReasonPkgSize;
    int noPreemptRunTime;
    int noPreemptFinishTime;
    char *  acctArchiveAt;
    int  absoluteRunLimit;
    int  lsbExitRateDuration;
    int  lsbTriggerDuration;
    int  maxJobinfoQueryPeriod;
    int jobSubRetryInterval;
    int pendingJobThreshold;
    int maxConcurrentJobQuery;
    int minSwitchPeriod;
    int condensePendingReasons;
    int slotBasedParallelSched;
    int disableUserJobMovement;
    int detectIdleJobAfter;
    int useSymbolPriority;
    int JobPriorityRound;
    char* priorityMapping;
    int maxInfoDirs;
    int minMbdRefreshTime;
    int enableStopAskingLicenses2LS;
    int expiredTime;
    char* mbdQueryCPUs;
    char *defaultApp;
    int  enableStream;
    char *streamFile;
    int  streamSize;
    int syncUpHostStatusWithLIM;
    char   *defaultSLA;
    int    slaTimer;
     int    mbdEgoTtl;
    int    mbdEgoConnTimeout;
    int    mbdEgoReadTimeout;
    int    mbdUseEgoMXJ;
    int    mbdEgoReclaimByQueue;
    int    defaultSLAvelocity;
    char  *exitRateTypes;
    float  globalJobExitRate;
    int    enableJobExitRatePerSlot;
    int  enableMetric;
    int  schMetricsSample;
    float maxApsValue;
    int  newjobRefresh;
    int  preemptJobType;
    char *defaultJgrp;
    int    jobRunlimitRatio;
    int    jobIncludePostproc;
    int    jobPostprocTimeout;
    int    sschedUpdateSummaryInterval;
    int    sschedUpdateSummaryByTask;
    int    sschedRequeueLimit;
    int    sschedRetryLimit;
    int    sschedMaxTasks;
    int    sschedMaxRuntime;
    char *sschedAcctDir;
    int     jgrpAutoDel;
    int     maxJobPreempt;
    int     maxJobRequeue;
    int     noPreemptRunTimePercent;
    int     maxStreamFileNum;
    int     PrivilegedUserForceBkill;
    int     intersectCandidateHosts;
    int     enforceOneUGLimit;
    int     logRuntimeESTExceeded;
    char*   computeUnitTypes;
    float   fairAdjustFactor;
    int     noPreemptFinishTimePercent;
    int     slotReserveQueueLimit;
    int     maxJobPercentagePerSession;
    int     useSuspSlots;
```

```
};
```

PARAMETERS

****names**	Reserved but not used; supply `NULL`.
***numUsers**	Reserved but not used; supply `NULL`.
options	Reserved but not used; supply 0.

parameterInfo structure

The `parameterInfo` structure contains the following fields:

defaultQueues	`DEFAULT_QUEUE`: A blank_separated list of queue names for automatic queue selection.
defaultHostSpec	`DEFAULT_HOST_SPEC`: The host name or host model name used as the system default for scaling `CPULIMIT` and `RUNLIMIT`.
mbatchdInterval	`MBD_SLEEP_TIME`: The interval in seconds at which the mbatchd dispatches jobs.
sbatchdInterval	`SBD_SLEEP_TIME`: The interval in seconds at which the sbatchd suspends or resumes jobs.
jobAcceptInterval	`JOB_ACCEPT_INTERVAL`: The interval at which a host accepts two successive jobs. (In units of `SBD_SLEEP_TIME`.)
maxDispRetries	`MAX_RETRY`: The maximum number of retries for dispatching a job.
maxSbdRetries	`MAX_SBD_FAIL`: The maximum number of retries for reaching an sbatchd.
preemptPeriod	`PREEM_PERIOD`: The interval in seconds for preempting jobs running on the same host.
cleanPeriod	`CLEAN_PERIOD`: The interval in seconds during which finished jobs are kept in core.
maxNumJobs	`MAX_JOB_NUM`: The maximum number of finished jobs that are logged in the current event file.
historyHours	`HIST_HOURS`: The number of hours of resource consumption history used for fair share scheduling and scheduling within a host partition.
pgSuspendIt	`PG_SUSP_IT`: The interval a host must be idle before resuming a job suspended for excessive paging.
defaultProject	The default project assigned to jobs.
retryIntvl	Job submission retry interval
nqsQueuesFlags	For Cray NQS compatiblilty only. Used by LSF to get the NQS queue information
maxPreExecRetry	The maximum number of times to attempt the preexecution command of a job from a remote cluster (MultiCluster only)
eventWatchTime	Event watching Interval in seconds
runTimeFactor	Run time weighting factor for fairshare scheduling
waitTimeFactor	used for calcultion of the fairshare scheduling formula
runJobFactor	Job slots weighting factor for fairshare scheduling
eEventCheckIntvl	default check interval

rusageUpdateRate	sbatchd report every sbd_sleep_time
rusageUpdatePercent	sbatchd updates jobs jRusage in mbatchd if more than 10% changes
condCheckTime	time period to check for reconfig
maxSbdConnections	The maximum number of connections between master and slave batch daemons
rschedInterval	The interval for rescheduling jobs
maxSchedStay	Max time mbd stays in scheduling routine, after which take a breather
freshPeriod	During which load remains fresh
preemptFor	The preemption behavior, GROUP_MAX, GROUP_JLP, USER_JLP, HOST_JLU, MINI_JOB, LEAST_RUN_TIME
adminSuspend	Flags whether users can resume their jobs when suspended by the LSF administrator
userReservation	Flags to enable/disable normal user to create advance reservation
cpuTimeFactor	CPU time weighting factor for fairshare scheduling
fyStart	The starting month for a fiscal year
maxJobArraySize	The maximum number of jobs in a job array
exceptReplayPeriod	Replay period for exceptions, in seconds
jobTerminateInterval	The interval to terminate a job
disableUAcctMap	User level account mapping for remote jobs is disabled
enforceFSProj	If set to TRUE, Project name for a job will be considerred when doing fairshare scheduling, i.e., as if user has submitted jobs using -G
enforceProjCheck	Enforces the check to see if the invoker of bsub is in the specifed group when the -P option is used
jobRunTimes	Run time for a job
dbDefaultIntval	Event table Job default interval
dbHjobCountIntval	Event table Job Host Count
dbQjobCountIntval	Event table Job Queue Count
dbUjobCountIntval	Event table Job User Count
dbJobResUsageIntval	Event table Job Resource Interval
dbLoadIntval	Event table Resource Load Interval
dbJobInfoIntval	Event table Job Info
jobDepLastSub	Used with job dependency scheduling
maxJobNameDep	Used with job dependency scheduling, deprecated
dbSelectLoad	select resources to be logged
jobSynJgrp	job synchronizes its group status
pjobSpoolDir	The batch jobs' temporary output directory
maxUserPriority	Maximal job priority defined for all users

PARAMETERS

jobPriorityValue	Job priority is increased by the system dynamically based on waiting time
jobPriorityTime	Waiting time to increase Job priority by the system dynamically
enableAutoAdjust	Enable internal statistical adjustment
autoAdjustAtNumPend	Start to autoadjust when the user has this number of pending jobs
autoAdjustAtPercent	If this number of jobs has been visited skip the user
sharedResourceUpdFactor	Static shared resource update interval for the cluster
scheRawLoad	schedule job based on raw load info
jobAttaDir	The batch jobs' external storage for attached data
maxJobMsgNum	Maximum message number for each job
maxJobAttaSize	Maximum attached data size to be transferred for each message
mbdRefreshTime	The life time of a child MBD to serve queries in the MT way
updJobRusageInterval	The interval of the execution cluster updating the job's resource usage
sysMapAcct	The account to which all windows workgroup users are to be mapped
preExecDelay	dispatch delay internal
updEventUpdateInterval	The interval of updating duplicated logging info
resourceReservePerSlot	Resources are reserved for parallel jobs on a per-slot basis
maxJobId	The Maximum JobId defined in the system
preemptResourceList	The list of preemption resources
preemptionWaitTime	The time for preemption wait
maxAcctArchiveNum	Max number of Acct files
acctArchiveInDays	Mbatchd Archive Interval
acctArchiveInSize	Mbatchd Archive threshold
committedRunTimeFactor	Committed run time weighting factor
enableHistRunTime	Enable the use of historical run time in the calculation of fairshare scheduling priority, Disable the use of historical run time in the calculation of fairshare scheduling priority
nqsUpdateInterval	NQS resource usage update interval
mcbOlmReclaimTimeDelay	Open lease reclaim time
chunkJobDuration	Enable chunk job dispatch for jobs with CPU limit or run limits
sessionInterval	The interval for scheduling jobs by scheduler daemon
publishReasonJobNum	The number of jobs per user per queue whose pending reason is published at the PEND_REASON_UPDATE_INTERVAL interval

publishReasonInterval	The interval for publishing job pending reason by scheduler daemon
publishReason4AllJobInterval	Interval (in seconds) of pending reason publish for all jobs
mcUpdPendingReasonInterval	MC pending reason update interval (0 means no updates)
mcUpdPendingReasonPkgSize	MC pending reason update package size (0 means no limit)
noPreemptRunTime	No preemption run time
noPreemptFinishTime	No preemption finish time
acctArchiveAt	Mbatchd Archive Time
absoluteRunLimit	Absolute run limit for job
lsbExitRateDuration	The job exit rate duration
lsbTriggerDuration	The duration to trigger eadmin
maxJobinfoQueryPeriod	Maximum time for job information query commands (for example,with bjobs) to wait
jobSubRetryInterval	Job submission retry interval
pendingJobThreshold	System wide max pending jobs
maxConcurrentJobQuery	Maximum concurrent job query
minSwitchPeriod	Minimal event switch period
condensePendingReasons	Condense pending reasons enabled
slotBasedParallelSched	Schedule Parallel jobs based on slots instead of CPUs
disableUserJobMovement	Job position control by admin
detectIdleJobAfter	Detect idle jobs after
useSymbolPriority	Use symbolic when specifing priority of session scheduler jobs
JobPriorityRound	Priority rounding for session scheduler jobs
priorityMapping	Priority rounding for session scheduler jobs
maxInfoDirs	Maximum number of info directories
minMbdRefreshTime	The minimum period of a child MBD to serve queries in the MT way
enableStopAskingLicenses2LS	Stop asking license to LS not due to lack license
mbdQueryCPUs	MBD child query processes will only run on the following CPUs
defaultApp	The default application profile assigned to jobs
enableStream	Streaming of lsbatch data is enabled
streamFile	File to which lsbatch data is streamed

PARAMETERS

streamSize	File size in MB to which lsbatch data is streamed
syncUpHostStatusWithLIM	Sync up host status with master LIM is enabled
defaultSLA	EGO Enabled SLA scheduling is enabled
slaTimer	EGO Enabled SLA scheduling timer period
mbdEgoTtl	EGO Enabled SLA scheduling time to live
mbdEgoConnTimeout	EGO Enabled SLA scheduling connection timeout
mbdEgoReadTimeout	EGO Enabled SLA scheduling read timeout
mbdUseEgoMXJ	EGO Enabled SLA scheduling use MXJ flag
mbdEgoReclaimByQueue	EGO Enabled SLA scheduling reclaim by queue
defaultSLAvelocity	EGO Enabled SLA scheduling default velocity
exitRateTypes	Type of host exit rate exception handling types: EXIT_RATE_TYPE
globalJobExitRate	Type of host exit rate exception handling types: GLOBAL_EXIT_RATE
enableJobExitRatePerSlot	Type of host exit rate exception handling types ENABLE_EXIT_RATE_PER_SLOT
enableMetric	Performance metrics monitor is enabled flag
schMetricsSample	Performance metrics monitor sample period flag
maxApsValue	used to bound: (1) factors, (2) weights, and (3) APS values
newjobRefresh	Child mbatchd gets updated information about new jobs from the parent mbatchd
preemptJobType	Job type to preempt, PREEMPT_JOBTYPE_BACKFILL, PREEMPT_JOBTYPE_EXCLUSIVE
defaultJgrp	The default job group assigned to jobs
jobRunlimitRatio	Max ratio between run limit and runtime estimation
jobIncludePostproc	Enable the post-execution processing of the job to be included as part of the job flag
jobPostprocTimeout	Timeout of post-execution processing
sschedUpdateSummaryInterval	The interval, in seconds, for updating the session scheduler status summary
sschedUpdateSummaryByTask	The number of completed tasks for updating the session scheduler status summary
sschedRequeueLimit	The maximum number of times a task can be requeued via requeue exit values
sschedRetryLimit	The maximum number of times a task can be retried after a dispatch error
sschedMaxTasks	The maximum number of tasks that can be submitted in one session
sschedMaxRuntime	The maximum run time of a single task
sschedAcctDir	The output directory for task accounting files
jgrpAutoDel	If TRUE enable the job group automatic deletion functionality (default is FALSE).
maxJobPreempt	Maximum number of job preempted times.

maxJobRequeue	Maximum number of job re-queue times.
noPreemptRunTimePercent	No preempt run time percent.
maxStreamFileNum	Maximum number of backup `stream.utc` files.
privilegedUserForceBkill	If enforced only admin can use the `bkill -r` option.
intersectCandidateHosts	Jobs run on only on hosts belonging to the intersection of the queue the job was submitted to, advance reservation hosts, and any hosts specified by bsub -m at the time of submission.
enforceOneUGLimit	Enforces the limitations of a single specified user group.
logRuntimeESTExceeded	Logging of runtime estimation events.
computeUnitTypes	Compute unit types.
fairAdjustFactor	Fairshare adjustment weighting factor.
noPreemptFinishTimePercent	No preempt finish time percent.
slotReserveQueueLimit	The reservation request being within JL/U.
maxJobPercentagePerSession	Job accept limit percentage.
useSuspSlots	The low priority job will use the slots freed by preempted jobs.

RETURN VALUES

pointer:parameterInfo

The function was a success, returns a pointer to a `parameterInfo` structure.

char:NULL

Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

`$LSB_CONFDIR/`*`cluster_name`*`/lsb.params`

lsb_peekjob()

Returns the base name of the file related to the job ID

DESCRIPTION

`lsb_peekjob()` retrieves the name of a job file.

Only the submitter can peek at job output.

The storage for the file name will be reused by the next call.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsbatch.h>
char * lsb_peekjob (LS_LONG_INT jobId)
```

PARAMETERS

jobId	The job ID that the LSF system assigned to the job. If a job in a job array is to be returned, use the array form `jobID[i]` where `jobID` is the job array name, and `i` is the index value.

RETURN VALUES

char:Name of job Output File

The function was a success.

char:NULL

Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

none

Equivalent line command

`bpeek`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_pendreason()

Explains why a job is pending.

DESCRIPTION

Use `lsb_pendreason()` to determine why a job is pending. Each pending reason is associated with one or more hosts.

SYNOPSIS

```
#include <lsf/lsbatch.h>

char *lsb_pendreason (int numReasons, int *rsTb,
                      struct jobInfoHead *jInfoH,
                      struct loadIndexLog *ld, int clusterId)
struct jobInfoHead{
    int numJobs;
    LS_LONG_INT *jobIds;
    int numHosts;
    char **hostNames;
    int   numClusters;
    char  **clusterNames;
    int   *numRemoteHosts;
    char  ***remoteHosts;
};
struct loadIndexLog {
    int nIdx;
    char **name;
};
```

PARAMETERS

numReasons The number of reasons in the rsTb reason table.

rsTb The reason table. Each entry in the table contains one of the following pending reasons:

PEND_JOB_NEW

A new job is waiting to be scheduled.

PEND_JOB_START_TIME

The job is held until its specified start time.

PEND_JOB_DEPEND

The job is waiting for its dependency condition(s) to be satisfied.

PEND_JOB_DEP_INVALID

The dependency condition is invalid or never satisfied.

PEND_JOB_MIG

The migrating job is waiting to be rescheduled.

PEND_JOB_PRE_EXEC

The job's pre-exec command exited with non-zero status.

PEND_JOB_NO_FILE

Unable to access the job file.

PEND_JOB_ENV

Unable to set the job's environment variables.

PEND_JOB_PATHS

Unable to determine the job's home or working directories.

PEND_JOB_OPEN_FILES

Unable to open the job's input and output files.

PEND_JOB_EXEC_INIT

Job execution initialization failed.

PEND_JOB_RESTART_FILE

Unable to copy restarting job's checkpoint files.

PEND_JOB_DELAY_SCHED

Scheduling of the job is delayed.

PEND_JOB_SWITCH

Waiting for the re-scheduling of the job after switching queues.

PEND_JOB_DEP_REJECT

An event is rejected by eeventd due to a syntax error.

PEND_JOB_JS_DISABLED

A JobScheduler feature is not enabled.

PEND_JOB_NO_PASSWD

Failed to get a user password.

PEND_JOB_LOGON_FAIL

The job is pending due to logon failure.

PEND_JOB_MODIFY

The job is waiting to be re-scheduled after its parameters have been changed.

PEND_JOB_TIME_INVALID

The job time event is invalid.

PEND_TIME_EXPIRED

The job time event has expired.

PEND_JOB_REQUEUED

The job has been requeued..

PEND_WAIT_NEXT

Waiting for the next time event.

PEND_JGRP_HOLD

The parent group is held.

PEND_JGRP_INACT

The parent group is inactive.

PEND_JGRP_WAIT

The parent group is waiting for scheduling.

PEND_JOB_RCLUS_UNREACH

The remote cluster(s) are unreachable.

PEND_JOB_QUE_REJECT

SNDJOBS_TO queue rejected by remote cluster(s).

PEND_JOB_RSCHED_START

Waiting for remote scheduling session.

PEND_JOB_RSCHED_ALLOC

Waiting for allocation replies from remote cluster(s).

PEND_JOB_FORWARDED

The job is forwarded to a remote cluster.

PEND_JOB_RMT_ZOMBIE

The job running remotely is in a zombie state.

PEND_JOB_ENFUGRP

The job's enforced user group share account is not selected.

PEND_SYS_UNABLE

The system is unable to schedule the job.

PEND_JGRP_RELEASE

The parent group has just been released.

PEND_HAS_RUN

The job has run since the parent group was active.

PEND_JOB_ARRAY_JLIMIT

The job has reached its running element limit.

PEND_CHKPNT_DIR

The checkpoint directory is invalid.

PEND_CHUNK_FAIL

The first job in the chunk failed (all other jobs in the chunk are set to PEND).

PEND_JOB_SLA_MET

Optimum number of running jobs for SLA has been reached.

PEND_JOB_APP_NOEXIST

Specified application profile does not exist

PEND_APP_PROCLIMIT

Job no longer satisfies application profile PROCLIMIT configuration

PEND_EGO_NO_HOSTS

No hosts for the job from EGO.

PEND_JGRP_JLIMIT

Job group's limit.

PEND_PREEXEC_LIMIT

Job pre-exec retry limit.

PEND_REQUEUE_LIMIT

Job re-queue limit.

PEND_BAD_RESREQ

Job has bad res req.

PEND_RSV_INACTIVE

job's reservation is inactive.

PEND_WAITING_RESUME

Job was in PSUSP with bad res req, after successful bmod waiting for the user to bresume.

PEND_QUE_INACT

The queue is inactivated by the administrator.

PEND_QUE_WINDOW

The queue is inactivated by its time windows.

PEND_QUE_JOB_LIMIT

The queue has reached its job slot limit.

PEND_QUE_USR_JLIMIT

The user has reached the per-user job slot limit of the queue.

PEND_QUE_USR_PJLIMIT

Not enough per-user job slots of the queue for the parallel job.

PEND_QUE_PRE_FAIL

The queue’s pre-exec command exited with non-zero status.

PEND_NQS_RETRY

The job was not accepted by the NQS host. Attempt again later.

PEND_NQS_REASONS

Unable to send the job to an NQS host.

PEND_NQS_FUN_OFF

Unable to contact NQS host.

PEND_SYS_NOT_READY

The system is not ready for scheduling after reconfiguration.

PEND_SBD_JOB_REQUEUE

The requeued job is waiting for rescheduling.

PEND_JOB_SPREAD_TASK

Not enough hosts to meet the job's spanning requirement.

PEND_QUE_SPREAD_TASK

Not enough hosts to meet the queue's spanning requirement.

PEND_QUE_PJOB_LIMIT

The queue has not enough job slots for the parallel job.

PEND_QUE_WINDOW_WILL_CLOSE

The job will not finish before queue's run window is closed.

PEND_QUE_PROCLIMIT

The job no longer satisfies queue PROCLIMIT configuration.

PEND_SBD_PLUGIN

The job requeued due to plug-in failure.

PEND_WAIT_SIGN_LEASE

Waiting for lease signing.

PEND_USER_JOB_LIMIT

The job slot limit is reached.

PEND_UGRP_JOB_LIMIT

A user group has reached its job slot limit.

PEND_USER_PJOB_LIMIT

The job slot limit for the parallel job is reached.

PEND_UGRP_PJOB_LIMIT

A user group has reached its job slot limit for the parallel job.

PEND_USER_RESUME

Waiting for scheduling after resumed by user.

PEND_USER_STOP

The job was suspended by the user while pending.

PEND_NO_MAPPING

Unable to determine user account for execution.

PEND_RMT_PERMISSION

The user has no permission to run the job on remote host/cluster.

PEND_ADMIN_STOP

The job was suspended by LSF admin or root while pending.

PEND_MLS_INVALID

The requested label is not valid.

PEND_MLS_CLEARANCE

The requested label is above user allowed range.

PEND_MLS_RHOST

The requested label rejected by /etc/rhost.conf.

PEND_MLS_DOMINATE

The requested label does not dominate current label.

PEND_MLS_FATAL

The requested label problem.

PEND_HOST_RES_REQ

The job's resource requirements not satisfied.

PEND_HOST_NONEXCLUSIVE

The job's requirement for exclusive execution not satisfied.

PEND_HOST_JOB_SSUSP

Higher or equal priority jobs already suspended by system.

PEND_HOST_PART_PRIO

The job failed to compete with other jobs on host partition.

PEND_SBD_GETPID

Unable to get the PID of the restarting job.

PEND_SBD_LOCK

Unable to lock the host for exclusively executing the job.

PEND_SBD_ZOMBIE

Cleaning up zombie job.

PEND_SBD_ROOT

Can't run jobs submitted by root. The job is rejected by the sbatchd.

PEND_HOST_WIN_WILL_CLOSE

The job will not finish on the host before queue's run window closes.

PEND_HOST_MISS_DEADLINE

The job will not finish on the host before job's termination deadline.

PEND_FIRST_HOST_INELIGIBLE

The specified first execution host is not eligible for this job at this time.

PEND_HOST_DISABLED

The host is closed by the LSF administrator.

PEND_HOST_LOCKED

The host is locked by the LSF administrator.

PEND_HOST_LESS_SLOTS

Not enough job slots for the parallel job.

PEND_HOST_WINDOW

The dispatch windows are closed.

PEND_HOST_JOB_LIMIT

The job slot limit reached.

PEND_QUE_PROC_JLIMIT

The queue’s per-CPU job slot limit is reached.

PEND_QUE_HOST_JLIMIT

The queue’s per-host job slot limit is reached.

PEND_USER_PROC_JLIMIT

The user’s per-CPU job slot limit is reached.

PEND_HOST_USR_JLIMIT

The host’s per-user job slot limit is reached.

PEND_HOST_QUE_MEMB

Not a member of the queue

PEND_HOST_USR_SPEC

Not a user specified host.

PEND_HOST_PART_USER

The user has no access to the host partition.

PEND_HOST_NO_USER

There is no such user account.

PEND_HOST_ACCPT_ONE

Just started a job recently.

PEND_LOAD_UNAVAIL

Load information unavailable.

PEND_HOST_NO_LIM

The LIM is unreachable by the sbatchd.

PEND_HOST_UNLICENSED

The host does not have a valid LSF software license.

PEND_HOST_QUE_RESREQ

The queue’s resource requirements are not satisfied.

PEND_HOST_SCHED_TYPE

The submission host type is not the same.

PEND_JOB_NO_SPAN

There are not enough processors to meet the job’s spanning requirement. The job level locality is unsatisfied.

PEND_QUE_NO_SPAN

There are not enough processors to meet the queue’s spanning requirement. The queue level locality is unsatisfied.

PEND_HOST_EXCLUSIVE

An exclusive job is running.

PEND_HOST_JS_DISABLED

Job Scheduler is disabled on the host. It is not licensed to accept repetitive jobs.

PEND_UGRP_PROC_JLIMIT

The user group's per-CPU job slot limit is reached.

PEND_BAD_HOST

Incorrect host, group or cluster name.

PEND_QUEUE_HOST

The host is not used by the queue.

PEND_HOST_LOCKED_MASTER

The host is locked by the master LIM.

PEND_HOST_LESS_RSVSLOTS

Not enough reserved job slots at this time for specified reservation ID

PEND_HOST_LESS_DURATION

Not enough slots or resources for whole duration of the job

PEND_HOST_NO_RSVID

Specified reservation has expired or has been deleted

PEND_HOST_LEASE_INACTIVE

The host is closed due to lease is inactive

PEND_HOST_ADRSV_ACTIVE

Not enough job slot(s) while advance reservation is active

PEND_QUE_RSVID_NOMATCH

This queue is not configured to send jobs to the cluster specified in the advance reservation

PEND_HOST_GENERAL

Individual host based reasons

PEND_HOST_RSV

Host does not belong to the specified advance reservation.

PEND_SBD_UNREACH

Unable to reach the sbatchd.

PEND_SBD_JOB_QUOTA

The number of jobs exceeds quota.

PEND_JOB_START_FAIL

The job failed in talking to the server to start the job.

PEND_JOB_START_UNKNWN

Failed in receiving the reply from the server when starting the job.

PEND_SBD_NO_MEM

Unable to allocate memory to run job. There is no memory on the sbatchd.

PEND_SBD_NO_PROCESS

Unable to fork process to run the job. There are no more processes on the sbatchd.

PEND_SBD_SOCKETPAIR

Unable to communicate with the job process.

PEND_SBD_JOB_ACCEPT

The slave batch server failed to accept the job.

PEND_LEASE_JOB_REMOTE_DISPATCH

Lease job remote dispatch failed.

PEND_JOB_RESTART_FAIL

Failed to restart job from last checkpoint.

PEND_HOST_LOAD

The load threshold is reached.

PEND_HOST_QUE_RUSAGE

The queue's requirements for resource reservation are not satisfied.

PEND_HOST_JOB_RUSAGE

The job's requirements for resource reservation are not satisfied.

PEND_RMT_JOB_FORGOTTEN

Remote job not recongized by remote cluster, waiting for rescheduling

PEND_RMT_IMPT_JOBBKLG

Remote import limit reached, waiting for rescheduling

PEND_RMT_MAX_RSCHED_TIME

Remote schedule time reached, waiting for rescheduling

PEND_RMT_MAX_PREEXEC_RETRY

Remote pre-exec retry limit reached, waiting for rescheduling

PEND_RMT_QUEUE_CLOSED

Remote queue is closed

PEND_RMT_QUEUE_INACTIVE

Remote queue is inactive

PEND_RMT_QUEUE_CONGESTED

Remote queue is congested

PEND_RMT_QUEUE_DISCONNECT

Remote queue is disconnected

PEND_RMT_QUEUE_NOPERMISSION

Remote queue is not configured to accept jobs from this cluster

PEND_RMT_BAD_TIME

Job's termination time exceeds the job creation time on remote cluster

PEND_RMT_PERMISSIONS

Permission denied on the execution cluster

PEND_RMT_PROC_NUM

Job's required on number of processors cannot be satisfied on the remote cluster

PEND_RMT_QUEUE_USE

User is not defined in the fairshare policy of the remote queue

PEND_RMT_NO_INTERACTIVE

Remote queue is a non-interactive queue

PEND_RMT_ONLY_INTERACTIVE

Remote queue is an interactive-only queue

PEND_RMT_PROC_LESS

Job's required maximum number of processors is less then the minimum number of processors defined on the remote queue

PEND_RMT_OVER_LIMIT

Job's required resource limit exceeds that of the remote queue

PEND_RMT_BAD_RESREQ

Job's resource requirements do not match with those of the remote queue

PEND_RMT_CREATE_JOB

Job failed to be created on the remote cluster

PEND_RMT_RERUN

Job is requeued for rerun on the execution cluster

PEND_RMT_EXIT_REQUEUE

Job is requeued on the execution cluster due to exit value

PEND_RMT_REQUEUE

Job was killed and requeued on the execution cluster

PEND_RMT_JOB_FORWARDING

Job was forwarded to remote cluster

PEND_RMT_QUEUE_INVALID

Remote import queue defined for the job in lsb.queues is either not ready or not valid

PEND_RMT_QUEUE_NO_EXCLUSIVE

Remote queue is a non-exclusive queue

PEND_RMT_UGROUP_MEMBER

Job was rejected; submitter does not belong to the specified User Group in the remote cluster or the user group does not exist in the remote cluster

PEND_RMT_INTERACTIVE_RERUN

Remote queue is rerunnable: can not accept interactive jobs

PEND_RMT_JOB_START_FAIL

Remote cluster failed in talking to server to start the job

PEND_RMT_FORWARD_FAIL_UGROUP_MEMBER

Job was rejected; submitter does not belong to the specified User Group in the remote cluster or the user group does not exist in the remote cluster

PEND_RMT_HOST_NO_RSVID

Specified remote reservation has expired or has been deleted

PEND_RMT_APP_NULL

Application profile could not be found in the remote cluster.

PEND_RMT_BAD_RUNLIMIT

Job's required RUNLIMIT exceeds RUNTIME * JOB_RUNLIMIT_RATIO of the remote cluster.

PEND_RMT_OVER_QUEUE_LIMIT

Job's required RUNTIME exceeds the hard runtime limit in the remote queue.

PEND_GENERAL_LIMIT_USER

Resource limit defined on user or user group has been reached.

PEND_GENERAL_LIMIT_QUEUE

Resource (%s) limit defined on queue has been reached.

PEND_GENERAL_LIMIT_PROJECT

Resource limit defined on project has been reached.

PEND_GENERAL_LIMIT_CLUSTER

Resource (%s) limit defined cluster wide has been reached.

PEND_GENERAL_LIMIT_HOST

Resource (%s) limit defined on host and/or host group has been reached.

PEND_GENERAL_LIMIT_JOBS_USER

JOBS limit defined for the user or user group has been reached.

PEND_GENERAL_LIMIT_JOBS_QUEUE

JOBS limit defined for the queue has been reached.

PEND_GENERAL_LIMIT_JOBS_PROJECT

JOBS limit defined for the project has been reached.

PEND_GENERAL_LIMIT_JOBS_CLUSTER

JOBS limit defined cluster-wide has been reached.

PEND_GENERAL_LIMIT_JOBS_HOST

JOBS limit defined on host or host group has been reached.

PEND_RMS_PLUGIN_INTERNAL

RMS scheduler plugin internal error.

PEND_RMS_PLUGIN_RLA_COMM

RLA communication failure.

PEND_RMS_NOT_AVAILABLE

RMS is not available.

PEND_RMS_FAIL_TOPOLOGY

Cannot satisfy the topology requirement.

PEND_RMS_FAIL_ALLOC

Cannot allocate an RMS resource.

PEND_RMS_SPECIAL_NO_PREEMPT_BACKFILL

RMS job with special topology requirements cannot be preemptive or backfill job.

PEND_RMS_SPECIAL_NO_RESERVE

RMS job with special topology requirements cannot reserve slots.

PEND_RMS_RLA_INTERNAL

RLA internal error.

PEND_RMS_NO_SLOTS_SPECIAL

Not enough slots for job. Job with RMS topology requirements cannot reserve slots, be preemptive, or be a backfill job.

PEND_RMS_RLA_NO_SUCH_USER

User account does not exist on the execution host.

PEND_RMS_RLA_NO_SUCH_HOST

Unknown host and/or partition unavailable.

PEND_RMS_CHUNKJOB

Cannot schedule chunk jobs to RMS hosts.

PEND_RLA_PROTOMISMATCH

RLA protocol mismatch.

PEND_RMS_BAD_TOPOLOGY

Contradictory topology requirements specified.

PEND_RMS_RESREQ_MCONT

Not enough slots to satisfy manditory contiguous requirement.

PEND_RMS_RESREQ_PTILE

Not enough slots to satisfy RMS ptile requirement.

PEND_RMS_RESREQ_NODES

Not enough slots to satisfy RMS nodes requirement.

PEND_RMS_RESREQ_NODES

Not enough slots to satisfy RMS nodes requirement.

PEND_RMS_RESREQ_BASE

Cannot satisfy RMS base node requirement.

PEND_RMS_RESREQ_RAILS

Cannot satisfy RMS rails requirement.

PEND_RMS_RESREQ_RAILMASK

Cannot satisfy RMS railmask requirement.

PEND_MAUI_UNREACH

Unable to communicate with external Maui scheduler.

PEND_MAUI_FORWARD

Job is pending at external Maui scheduler.

PEND_MAUI_REASON

External Maui scheduler sets detail reason.

PEND_CPUSET_ATTACH

CPUSET attach failed. Job requeued.

PEND_CPUSET_NOT_CPUSETHOST

Not a cpuset host

PEND_CPUSET_TOPD_INIT

Topd initialization failed

PEND_CPUSET_TOPD_TIME_OUT

Topd communication timeout

PEND_CPUSET_TOPD_FAIL_ALLOC

Cannot satisfy the cpuset allocation requirement

PEND_CPUSET_TOPD_BAD_REQUEST

Bad cpuset allocation request

PEND_CPUSET_TOPD_INTERNAL

Topd internal error

PEND_CPUSET_TOPD_SYSAPI_ERR

Cpuset system API failure

PEND_CPUSET_TOPD_NOSUCH_NAME

Specified static cpuset does not exist on the host

PEND_CPUSET_TOPD_JOB_EXIST

Cpuset is already allocated for this job

PEND_CPUSET_TOPD_NO_MEMORY

Topd malloc failure

PEND_CPUSET_TOPD_INVALID_USER

User account does not exist on the cpuset host

PEND_CPUSET_TOPD_PERM_DENY

User does not have permission to run job within cpuset

PEND_CPUSET_TOPD_UNREACH

Topd is not available

PEND_CPUSET_TOPD_COMM_ERR

Topd communication failure

PEND_CPUSET_PLUGIN_INTERNAL

CPUSET Scheduler Plugin internal error.

PEND_CPUSET_CHUNKJOB

Cannot schedule chunk jobs to cpuset hosts

PEND_CPUSET_CPULIST

Cannot satisfy CPUSET CPU_LIST requirement

PEND_CPUSET_MAXRADIUS

Cannot satisfy CPUSET MAX_RADIUS requirement

PEND_NODE_ALLOC_FAIL

Node allocation failed

PEND_RMSRID_UNAVAIL

RMS resource is not available.

PEND_NO_FREE_CPUS

Not enough free cpus to satisfy job requirements

PEND_TOPOLOGY_UNKNOWN

Topology unknown or recently changed

PEND_BAD_TOPOLOGY

Contradictory topology requirement specified

PEND_RLA_COMM

RLA communications failure

PEND_RLA_NO_SUCH_USER

User account does not exist on execution host

PEND_RLA_INTERNAL

RLA internal error

PEND_RLA_NO_SUCH_HOST

Unknown host and/or partition unavailable

PEND_RESREQ_TOOFEWSLOTS

Too few slots for specified topology requirement

PEND_PSET_PLUGIN_INTERNAL

PSET scheduler plugin internal error

PEND_PSET_RESREQ_PTILE

Cannot satisfy PSET ptile requirement

PEND_PSET_RESREQ_CELLS

Cannot satisfy PSET cells requirement

PEND_PSET_CHUNKJOB

Cannot schedule chunk jobs to PSET hosts

PEND_PSET_NOTSUPPORT

Host does not support processor set functionality

PEND_PSET_BIND_FAIL

PSET bind failed. Job requeued

PEND_PSET_RESREQ_CELLLIST

Cannot satisfy PSET CELL_LIST requirement

PEND_SLURM_PLUGIN_INTERNAL

SLURM scheduler plugin internal error

PEND_SLURM_RESREQ_NODES

Not enough resource to satisfy SLURM nodes requirment

PEND_SLURM_RESREQ_NODE_ATTR

Not enough resource to satisfy SLURM node attributes requirment.

PEND_SLURM_RESREQ_EXCLUDE

Not enough resource to satisfy SLURM exclude requirment.

PEND_SLURM_RESREQ_NODELIST

Not enough resource to satisfy SLURM nodelist requirment.

PEND_SLURM_RESREQ_CONTIGUOUS

Not enough resource to satisfy SLURM contiguous requirment.

PEND_SLURM_ALLOC_UNAVAIL

SLURM allocation is not available. Job requeued.

PEND_SLURM_RESREQ_BAD_CONSTRAINT

Invalid grammar in SLURM constraints option, job will never run.

PEND_CRAYX1_SSP

Not enough SSPs for job.

PEND_CRAYX1_MSP

Not enough MSPs for job.

PEND_CRAYX1_PASS_LIMIT

Unable to pass job limit information to psched.

PEND_CRAYXT3_ASSIGN_FAIL

Cannot create or assign a partition by CPA.

PEND_BLUEGENE_PLUGIN_INTERNAL

BG/L: Scheduler plug-in internal error.

PEND_JOB_APP_NOEXIST

Specified application profile does not exist.

PEND_BLUEGENE_ALLOC_UNAVAIL

BG/L: Allocation is not available. Job requeued.

PEND_BLUEGENE_NOFREEMIDPLANES

BG/L: No free base partitions available for a full block allocation.

PEND_BLUEGENE_NOFREEQUARTERS

BG/L: No free quarters available for a small block allocation.

PEND_BLUEGENE_NOFREENODECARDS

BG/L: No free node cards available for a small block allocation.

PEND_PS_PLUGIN_INTERNAL

Host does not have enough slots for this SLA job.

PEND_PS_MBD_SYNC

EGO SLA: Failed to synchronize resource with MBD.

PEND_CUSTOMER_MIN

Customized pending reason number between min and max.

PEND_CUSTOMER_MAX

Customized pending reason number between min and max.

PEND_MAX_REASONS

The maximum number of reasons.

PEND_RESIZE_FIRSTHOSTUNAVAIL

First execution host unavailable.

PEND_RESIZE_MASTERSUSP

Master host is not in the RUN state.

PEND_RESIZE_MASTER_SAME

The host is not same as for master.

PEND_RESIZE_SPAN_PTILE

The host is already used by master.

PEND_RESIZE_SPAN_HOSTS

The job can only use first host.

PEND_RESIZE_LEASE_HOST

The job cannot get slots on remote hosts.

PEND_COMPOUND_RESREQ_OLD_LEASE_HOST

The job cannot get slots on pre-7Update5 remote hosts.

PEND_COMPOUND_RESREQ_TOPLIB_HOST

Hosts using LSF HPC system integrations do not support compound resource requirements.

SUSP_USER_REASON

Virtual code. Not a reason.

SUSP_USER_RESUME

bresumed a USUSP job.

SUSP_USER_STOP

User bstoped a started job.

SUSP_QUEUE_REASON

Virtual code. Not a reason.

SUSP_QUEUE_WINDOW

Queue closed by its time window.

SUSP_RESCHED_PREEMPT

Preempted for re-scheduling.

SUSP_HOST_LOCK

Host is locked now.

SUSP_LOAD_REASON

Load index - has subreasons.

SUSP_MBD_PREEMPT

By jobs in preemptive queue.

SUSP_SBD_PREEMPT

By jobs in preemptive queue.

SUSP_QUE_STOP_COND

Queue's STOP_COND is true.

SUSP_QUE_RESUME_COND

Queue's RESUME_COND is false.

SUSP_PG_IT

PG_SUSP_IT not satisfied.

SUSP_REASON_RESET

Just reset the last reason.

SUSP_LOAD_UNAVAIL

Load is unavailable.

SUSP_ADMIN_STOP

admin bstoped a started job.

SUSP_RES_RESERVE

Don't have enough resource to resume the job.

SUSP_MBD_LOCK

The job is lock by mbatchd.

SUSP_RES_LIMIT

The job is terminated due to resource limit.

SUB_REASON_RUNLIMIT

Sub reason of SUSP_RES_LIMIT: RUNLIMIT is reached.

SUB_REASON_DEADLINE

Sub reason of SUSP_RES_LIMIT: DEADLINE is reached.

SUB_REASON_PROCESSLIMIT

Sub reason of SUSP_RES_LIMIT: PROCESSLIMIT is reached.

SUB_REASON_CPULIMIT

Sub reason of SUSP_RES_LIMIT: CPULIMIT is reached.

SUB_REASON_MEMLIMIT

Sub reason of SUSP_RES_LIMIT: MEMLIMIT is reached.

SUB_REASON_THREADLIMIT

Sub reason of SUSP_RES_LIMIT: THREADLIMIT is reached.

SUB_REASON_SWAPLIMIT

Sub reason of SUSP_RES_LIMIT: SWAPLIMIT is reached.

SUSP_SBD_STARTUP

The suspending action is being taken on the job when the sbatchd is restarting up.

SUSP_HOST_LOCK_MASTER

Host is locked by master LIM.

SUSP_HOST_RSVACTIVE

An advance reservation using the host is active.

SUSP_DETAILED_SUBREASON

There is a detailed reason in the subreason field.

SUSP_GLB_LICENSE_PREEMPT

The job is preempted by glb.

SUSP_CRAYX1_POSTED

Job not placed by Cray X1.

SUB_REASON_CRAYX1_RESTART

Application is in the process of being restarted and it is under the control of CPR.

SUB_REASON_CRAYX1_DEPTH

Depth does not match those allowed by the gate.

SUB_REASON_CRAYX1_GID

GID does not match those allowed by the gate.

SUB_REASON_CRAYX1_GASID

No GASID is available.

SUB_REASON_CRAYX1_HARDLABEL

Hard label does not match those allowed by the gate.

SUB_REASON_CRAYX1_LIMIT

Limit exceeded in regions or domains.

SUB_REASON_CRAYX1_MEMORY

Memory size does not match those allowed by the gate.

SUB_REASON_CRAYX1_SOFTLABEL

Soft label does not match those allowed by the gate.

SUB_REASON_CRAYX1_SIZE

Size gate (width times depth larger than gate allows).

SUB_REASON_CRAYX1_TIME

Time limit does not match those allowed by the gate.

SUB_REASON_CRAYX1_UID

UID does not match those allowed by the gate.

SUB_REASON_CRAYX1_WIDTH

Width does not match those allowed by the gate.

SUSP_ADVRSV_EXPIRED

Job suspended when its advance reservation expired.

PEND_HOST_EXCLUSIVE_RESERVE

Exclusive job reserves slots on host.

PEND_RMT_BAD_RUNLIMIT

Job's required RUNLIMIT exceeds RUNTIME * JOB_RUNLIMIT_RATIO of the remote cluster

PEND_RMT_OVER_QUEUE_LIMIT

Job's required RUNTIME exceeds the hard runtime limit in the remote queue

PEND_JGRP_JLIMIT

The specified job group has reached its job limit

PEND_RMS_RESREQ_RAILS

Cannot satisfy RMS rails requirement

PEND_RMS_RESREQ_RAILMASK

Cannot satisfy RMS railmask requirement

PEND_RMS_RESREQ_BASE

Cannot satisfy RMS base node requirement

jInfoH `jInfoH` contains job information.

numJobs

The number of jobs in the connection

jobIds

An array of job identification numbers in the conection

numHosts

The number of hosts in the connection

hostNames

An array of host names in the connection

ld From `lsb_suspreason`, when reasons is SUSP_LOAD_REASON, ld is used to determine the name of any external load indices. ld uses the most recent load index log in the `lsb.events` file.

The `loadIndexLog` structure contains the following fields:

nIdx

Number of load indices

names

Names of load indices

clusterId MultiCluster cluster ID. If `clusterId` is greater than or equal to 0, the job is a pending remote job, and lsb_pendreason checks for *host_name@cluster_name*. If host name is needed, it should be found in `jInfoH->remoteHosts`. If the remote host name is not available, the constant string `remoteHost` is used.

RETURN VALUES

character:reasons	The function is successful. It returns a reason why the job is pending.
character:NULL	The function fails. The reason code is bad.

ERRORS

If no PEND reason is found, the function fails and lsberrno is set to indicate the error.

SEE ALSO

Related APIs:

lsb_geteventrec() - Returns an event record from a log files

Equivalent line command

```
bjobs -p
```

Files:

```
${LSF_ENVDIR-/etc}/lsf.conf
```

lsb_perror()

Prints a batch LSF error message on stderr.

DESCRIPTION

`lsb_perror()` prints a batch LSF error message on stderr. The `usrMsg` is printed out first, followed by a ":" and the batch error message corresponding to `lsberrno`.

lsb_perror - Print `LSBATCH` error message on `stderr`. In addition to the error message defined by `lsberrno`, user supplied message usrMsg1 is printed out first and a ':' is added to separate `* usrMsg1` and `LSBATCH` error message.

SYNOPSIS

```
#include <lsf/lsbatch.h>
void lsb_perror (char *usrMsg)
```

PARAMETERS

***usrMsg**	A user supplied error message.

RETURN VALUES

None	Prints out the user supplied error message.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

none

Equivalent line command

none

Files

none

lsb_postjobmsg()

Sends messages and data posted to a job.

DESCRIPTION

Use `lsb_postjobmsg()` to post a message and data to a job, open a TCP connection, and transfer attached message and data from the mbatchd. Use `lsb_readjobmsg()` to display messages and copy data files posted by `lsb_postjobmsg()`.

While you can post multiple messages and attached data files to a job, you must call `lsb_postjobmsg()` for each message and attached data file you want to post. By default, `lsb_postjobmsg()` posts a message to position 0 of the message index (msgId) (see PARAMETERS) of the specified job. To post additional messages to a job, call `lsb_postjobmsg()` and increment the message index .

`lsb_readjobmsg()` reads posted job messages by their position in the message index.

If a data file is attached to a message and the flag `EXT_ATTA_POST` is set, use the `JOB_ATTA_DIR` parameter in `lsb.params(5)` to specify the directory where attachment data fies are saved. The directory must have at least 1MB of free space. The mbatchd checks for available space in the job attachment directory before transferring the file.

Use the `MAX_JOB_ATTA_SIZE` parameter in `lsb.params(5)` to set a maximum size for job message attachments.

Users can only send messages and data from their own jobs. Root and LSF administrators can also send messages of jobs submtted by other users, but they cannot attach data files to jobs owned by other users.

You can post messages and data to a job until it is cleaned from the system. You cannot send messages and data to finished or exited jobs.

SYNOPSIS

```
#include <lsf/lsbatch.h>
#include <time.h>

int lsb_readjobmsg(struct jobExternalMsgReq *jobExternalMsg,
                   char *filename)

struct jobExternalMsgReq {
    int options;
    LS_LONG_INT jobId;
    int msgIdx;
    char *desc;
    int userId;
    long dataSize;
    time_t postTime;
    int dataStatus;
};
```

PARAMETERS

filename	Name of attached data file. If no file is attached, use NULL.

jobExternalMsg This structure contains the information required to define an external message of a job.

options

Specifies if the message has an attachment to be posted.

`<lsf/lsbatch.h>` defines the following flags constructed from bits. These flags correspond to the options.

EXT_MSG_POST

Post an external job message. Don't attach a data file.

EXT_ATTA_POST

Post an external job message. Attach a data file.

When the message is read by `lsb_readjobmsg()`, if a data file is not attached, the error message "The attached data of the message is not available" is displayed, and the external job message is displayed.

jobId

The system generated job Id of the job.

msgIdx

The message index. A job can have more than one message. Use `msgIdx` in an array to index messages. The default is position 0.

desc

The text of the message.

userId

The userId of the author of the message.

dataSize

The size of the data file. If no data file is attached, the size is 0.

postTime

The time the author posted the message.

dataStatus

The status of the attached data file.

RETURN VALUES

integer:value	The successful function returns a socket number.
integer:0	The `EXT_ATTA_POST` bit of options is not set or there is no attached data.
integer:-1	The function failed.

ERRORS

If the function fails, lserrno is set to indicate the error.

SEE ALSO

Related APIs

lsb_readjobmsg() - Reads messages and data posted to a job

Equivalent line command

`bpost`

Files

`lsb.params`

`JOB_ATTA_DIR`

`LSB_SHAREDIR/info/`

lsb_puteventrec()

Puts information of an `eventRec` structure pointed to by `logPtr` into a log file.

DESCRIPTION

`lsb_puteventrec()` puts information of an `eventRec` structure pointed to by `logPtr` into a log file. `log_fp` is a pointer pointing to the log file name that could be either event a log file or job log file.

See `lsb_geteventrec()` for detailed information about parameters.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_puteventrec(FILE *log_fp, struct eventRec *logPtr)

struct eventRec {
    char version[MAX_VERSION_LEN];
    int type;
    time_t eventTime;
    union eventLog eventLog;
};
union  eventLog {
    struct jobNewLog jobNewLog;
    struct jobStartLog jobStartLog;
    struct jobStatusLog jobStatusLog;
    struct sbdJobStatusLog sbdJobStatusLog;
    struct jobSwitchLog jobSwitchLog;
    struct jobMoveLog jobMoveLog;
    struct queueCtrlLog queueCtrlLog;
    struct newDebugLog  newDebugLog;
    struct hostCtrlLog hostCtrlLog;
    struct mbdStartLog mbdStartLog;
    struct mbdDieLog mbdDieLog;
    struct unfulfillLog unfulfillLog;
    struct jobFinishLog jobFinishLog;
    struct loadIndexLog loadIndexLog;
    struct migLog migLog;
    struct calendarLog calendarLog;
    struct jobForwardLog jobForwardLog;
    struct jobAcceptLog jobAcceptLog;
    struct statusAckLog statusAckLog;
    struct signalLog signalLog;
    struct jobExecuteLog jobExecuteLog;
    struct jobMsgLog jobMsgLog;
```

SYNOPSIS

```
    struct jobMsgAckLog jobMsgAckLog;
    struct jobRequeueLog jobRequeueLog;
    struct chkpntLog chkpntLog;
    struct sigactLog sigactLog;
    struct jobOccupyReqLog jobOccupyReqLog;
    struct jobVacatedLog jobVacatedLog;
    struct jobExceptionLog jobExceptionLog;
    struct jobCleanLog jobCleanLog;
    struct Log jgrpNewLog;
    struct jgrpCtrlLog jgrpCtrlLog;
    struct jobForceRequestLog jobForceRequestLog;
    struct logSwitchLog logSwitchLog;
    struct jobModLog jobModLog;
    struct jgrpStatusLog jgrpStatusLog;
    struct jobAttrSetLog jobAttrSetLog;
      struct jobExternalMsgLog jobExternalMsgLog;
    struct jobChunkLog jobChunkLog;
    struct sbdUnreportedStatusLog sbdUnreportedStatusLog;
    struct rsvFinishLog rsvFinishLog;
    struct cpuProfileLog cpuProfileLog;
    struct dataLoggingLog dataLoggingLog;
    struct jobRunRusageLog   jobRunRusageLog;
    struct eventEOSLog       eventEOSLog;
    struct slaLog            slaLog;
    struct perfmonLog        perfmonLog;
};
struct xFile {
    char *subFn;
    char *execFn;
    int options;
};
struct jobAttrSetLog {   /* Structure for log_jobattrset() and other calls */
    int       jobId;
    int       idx;
    int       uid;
    int       port;
    char      *hostname;
};
struct logSwitchLog { /* Records of logged events */
    int lastJobId;
};
struct dataLoggingLog {          /* Records of job cpu data logged event */
    time_t loggingTime;
```

```
};
struct jgrpNewLog {
    int    userId;
    time_t submitTime;
    char   userName[MAX_LSB_NAME_LEN];
    char   *depCond;
    char   *timeEvent;
    char   *groupSpec;
    char   *destSpec;
    int    delOptions;
    int    delOptions2;
    int    fromPlatform;
    char   *sla;
    int  maxJLimit;
};
struct jgrpCtrlLog {
    int    userId;
    char   userName[MAX_LSB_NAME_LEN];
    char   *groupSpec;
    int    options;
    int    ctrlOp;
};
struct jgrpStatusLog {
    char   *groupSpec;
    int    status;
    int    oldStatus;
};
struct jobNewLog {         /* logged in lsb.events when a job is submitted */
    int     jobId;
    int     userId;
    char    userName[MAX_LSB_NAME_LEN];
    int     options;
    int     options2;
    int     numProcessors;
    time_t  submitTime;
    time_t  beginTime;
    time_t  termTime;
    int     sigValue;
    int     chkpntPeriod;
    int     restartPid;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    hostSpec[MAXHOSTNAMELEN];
    float   hostFactor;
```

SYNOPSIS

```
int     umask;
char    queue[MAX_LSB_NAME_LEN];
char    *resReq;
char    fromHost[MAXHOSTNAMELEN];
char    *cwd;
char    *chkpntDir;
char    *inFile;
char    *outFile;
char    *errFile;
char    *inFileSpool;
char    *commandSpool;
char    *jobSpoolDir;
char    *subHomeDir;
char    *jobFile;
int     numAskedHosts;
char    **askedHosts;
char    *dependCond;
char    *timeEvent;
char    *jobName;
char    *command;
int     nxf;
struct xFile *xf;
char    *preExecCmd;
char    *mailUser;
char    *projectName;
int     niosPort;
int     maxNumProcessors;
char    *schedHostType;
char    *loginShell;
char    *userGroup;
char    *exceptList;
int     idx;
int     userPriority;
char    *rsvId;
char    *jobGroup;
char    *extsched;
int     warningTimePeriod;
char    *warningAction;
char    *sla;
int     SLArunLimit;
char    *licenseProject;
int     options3;
char    *app;
```

```
    char    *postExecCmd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
};
struct jobModLog {
    char    *jobIdStr;
    int     options;
    int     options2;
    int     delOptions;
    int     delOptions2;
    int     userId;
    char    *userName;
    int     submitTime;
    int     umask;
    int     numProcessors;
    int     beginTime;
    int     termTime;
         int     sigValue;
    int     restartPid;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    char    *dependCond;
    char    *timeEvent;
    char    *subHomeDir;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *inFileSpool;
    char    *commandSpool;
    int     chkpntPeriod;
    char    *chkpntDir;
    int     nxf;
    struct  xFile *xf;
    char    *jobFile;
    char    *fromHost;
    char    *cwd;
    char    *preExecCmd;
    char    *mailUser;
    char    *projectName;
    int     niosPort;
    int     maxNumProcessors;
    char    *loginShell;
    char    *schedHostType;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *jobGroup;
    char    *sla;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    char    *apsString;
    char   *postExecCmd;
    int      runtimeEstimation;
};
struct jobStartLog {
    LS_LONG_INT jobId;
    int jStatus;
    int jobPid;
    int jobPGid;
    float hostFactor;
    int numExHosts;
    char **execHosts;
     char   *queuePreCmd;
    char   *queuePostCmd;
          int    jFlags;
    char   *userGroup;
    int    idx;
    char   *additionalInfo;
    int    duration4PreemptBackfill;
};
struct jobStartAcceptLog {  /* logged in lsb.events when a job start request is accepted
*/
```

```
    int    jobId;
    int    jobPid;
    int    jobPGid;
    int    idx;
};
struct jobExecuteLog {   /* logged in lsb.events when a job is executed */
    int    jobId;
    int    execUid;
    char   *execHome;
    char   *execCwd;
    int    jobPGid;
    char   *execUsername;
    int    jobPid;
    int    idx;
    char   *additionalInfo;
    int    SLAscaledRunLimit;
    int    position;
    char   *execRusage;
   int    duration4PreemptBackfill;
};
struct jobStatusLog {
    int    jobId;
    int    jStatus;
    int    reasons;
    int    subreasons;
    float cpuTime;
    time_t endTime;
    int    ru;
    struct lsfRusage lsfRusage;
    int    jFlags;
    int    exitStatus;
    int    idx;
    int    exitInfo;
};
struct sbdJobStatusLog {     /* logged when a job's status is changed */
    int    jobId;
    int    jStatus;
    int    reasons;
    int    subreasons;
    int    actPid;
    int    actValue;
    time_t actPeriod;
    int    actFlags;
```

SYNOPSIS

```
    int    actStatus;
    int    actReasons;
    int    actSubReasons;
    int    idx;
    int    sigValue;
    int    exitInfo;
};
struct sbdUnreportedStatusLog {   /* job status that we could send to MBD */
    int    jobId;
    int    actPid;
    int    jobPid;
    int    jobPGid;
    int    newStatus;
    int    reason;
    int    subreasons;
    struct lsfRusage lsfRusage;
    int    execUid;
    int    exitStatus;
    char   *execCwd;
    char   *execHome;
    char   *execUsername;
    int    msgId;
    struct jRusage runRusage;
    int    sigValue;
    int    actStatus;
    int    seq;
    int    idx;
    int    exitInfo;
};
struct jobSwitchLog {     /* logged when a job is switched to another queue */
    int    userId;
    int    jobId;
    char   queue[MAX_LSB_NAME_LEN];
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
};
struct jobMoveLog {       /* logged when a job is moved to another position */
    int    userId;
    int    jobId;
    int    position;
    int    base;
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
```

```
};
struct chkpntLog {
    int jobId;
    time_t period;
    int pid;
    int ok;
    int flags;
    int    idx;
};
struct jobRequeueLog {
    int   jobId;
    int   idx;
};
struct jobCleanLog {
    int   jobId;
    int   idx;
};
struct jobExceptionLog {
    int jobId;
    int    exceptMask;
    int    actMask;
    time_t timeEvent;
    int    exceptInfo;
    int    idx;
};
struct sigactLog {
    int      jobId;
    time_t   period;
    int      pid;
    int      jStatus;
    int      reasons;
    int      flags;
    char     *signalSymbol;
    int      actStatus;
    int      idx;
};
struct migLog {
    int    jobId;
    int    numAskedHosts;
    char   **askedHosts;
    int    userId;
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
```

SYNOPSIS

```
};
struct signalLog {
    int    userId;
    int    jobId;
    char   *signalSymbol;
    int    runCount;
    int    idx;
    char   userName[MAX_LSB_NAME_LEN];
};
struct queueCtrlLog { /* logged when bqc command is invoked */
    int opCode;
    char queue[MAX_LSB_NAME_LEN];
    int  userId;
    char userName[MAX_LSB_NAME_LEN];
    char message[MAXLINELEN];
};
struct newDebugLog {
    int opCode;
    int level;
    int logclass;
    int turnOff;
    char logFileName[MAXLSFNAMELEN];
    int userId;
 };
struct hostCtrlLog { /* logged when dynamic hosts are added to group */
    int opCode;
    char host[MAXHOSTNAMELEN];
    int  userId;
    char userName[MAX_LSB_NAME_LEN];
    char message[MAXLINELEN];
};
struct hgCtrlLog {       /* logged when dynamic hosts are added to group */
    int    opCode;
    char   host[MAXHOSTNAMELEN];
    char   grpname[MAXHOSTNAMELEN];
    int    userId;
    char   userName[MAX_LSB_NAME_LEN];
    char   message[MAXLINELEN];
};
struct mbdStartLog {
    char master[MAXHOSTNAMELEN];
    char cluster[MAXLSFNAMELEN];
    int  numHosts;
    int  numQueues;
};
struct mbdDieLog {
    char master[MAXHOSTNAMELEN];
    int  numRemoveJobs;
    int  exitCode;
    char   message[MAXLINELEN];
};
struct unfulfillLog {           /* logged before mbatchd dies */
    int  jobId;
    int  notSwitched;
    int  sig;
    int  sig1;
    int  sig1Flags;
    time_t chkPeriod;
    int  notModified;
    int  idx;
    int  miscOpts4PendSig;
};
struct jobFinishLog {          /* logged in lsb.acct when a job finished */
    int    jobId;
    int    userId;
    char   userName[MAX_LSB_NAME_LEN];
    int    options;
    int    numProcessors;
    int    jStatus;
    time_t submitTime;
    time_t beginTime;
    time_t termTime;
    time_t startTime;
    time_t endTime;
    char   queue[MAX_LSB_NAME_LEN];
    char   *resReq;
    char   fromHost[MAXHOSTNAMELEN];
    char   *cwd;
    char   *inFile;
    char   *outFile;
    char   *errFile;
    char   *inFileSpool;
    char   *commandSpool;
    char   *jobFile;
    int    numAskedHosts;
    char   **askedHosts;
```

SYNOPSIS

```
    float  hostFactor;
    int    numExHosts;
    char   **execHosts;
    float  cpuTime;
    char   *jobName;
    char   *command;
    struct  lsfRusage lsfRusage;
    char   *dependCond;
    char   *timeEvent;
    char   *preExecCmd;
    char   *mailUser;
    char   *projectName;
    int    exitStatus;
    int    maxNumProcessors;
    char   *loginShell;
    int    idx;
    int    maxRMem;
    int    maxRSwap;
    char   *rsvId;
    char   *sla;
    int    exceptMask;
    char   *additionalInfo;
    int    exitInfo;
    int    warningTimePeriod;
    char   *warningAction;
    char   *chargedSAAP;
    char   *app;
    char   *postExecCmd;
    int    runtimeEstimation;
    char   *jgroup;
};
struct loadIndexLog {
    int  nIdx;
    char **name;
};
struct calendarLog {
    int    options;
    int    userId;
    char   *name;
    char   *desc;
    char   *calExpr;
};
struct jobForwardLog {
    int    jobId;
    char   *cluster;
    int    numReserHosts;
    char   **reserHosts;
    int    idx;
          int    jobRmtAttr;
};
struct jobAcceptLog {
    int         jobId;
    LS_LONG_INT remoteJid;
    char        *cluster;
    int         idx;
    int         jobRmtAttr;
};
struct statusAckLog {
    int jobId;
    int statusNum;
    int    idx;
};
struct jobMsgLog {
    int   usrId;
    int   jobId;
    int   msgId;
    int   type;
    char  *src;
    char  *dest;
    char  *msg;
    int   idx;
};
struct jobMsgAckLog {
    int usrId;
    int jobId;
    int msgId;
    int type;
    char *src;
    char *dest;
    char *msg;
    int    idx;
};
struct jobOccupyReqLog {
    int userId;
    int jobId;
    int numOccupyRequests;
```

SYNOPSIS

```
    void *occupyReqList;
    int    idx;
    char userName[MAX_LSB_NAME_LEN];
};
struct jobVacatedLog {
    int userId;
    int jobId;
    int    idx;
    char userName[MAX_LSB_NAME_LEN];
};
struct jobForceRequestLog {
    int     userId;
    int     numExecHosts;
    char**  execHosts;
    int     jobId;
    int     idx;
    int     options;
    char    userName[MAX_LSB_NAME_LEN];
    char   *queue;
};
struct jobChunkLog {
    long         membSize;
    LS_LONG_INT *membJobId;
    long         numExHosts;
    char       **execHosts;
};
struct jobExternalMsgLog {
    int      jobId;
          int      idx;
    int      msgIdx;
    char     *desc;
    int      userId;
    long     dataSize;
    time_t   postTime;
    int      dataStatus;
    char     *fileName;
    char     userName[MAX_LSB_NAME_LEN];
};
struct rsvRes {
    char     *resName;
    int      count;
    int      usedAmt;
```

```
};
struct rsvFinishLog {           /* for advanced reservation */
    time_t        rsvReqTime;
    int           options;
    int         uid;
    char          *rsvId;
    char          *name;
    int           numReses;
    struct rsvRes *alloc;
    char          *timeWindow;
    time_t        duration;
    char          *creator;
};
struct cpuProfileLog {
    char    servicePartition[MAX_LSB_NAME_LEN];
    int     slotsRequired;
    int     slotsAllocated;
    int     slotsBorrowed;
    int     slotsLent;
};
struct jobRunRusageLog {         /* log the running rusage of a job in the lsb.stream
file */
    int              jobid;
    int              idx;
    struct jRusage   jrusage;
};
struct slaLog {              /* SLA event */
    char   *name;
    char   *consumer;
    int    goaltype;
    int    state;
    int    optimum;
    int    counters[NUM_JGRP_COUNTERS];
};
struct perfmonLog {              /* performance metrics log in lsb.stream */
    int    samplePeriod;
    int    totalQueries;
    int    jobQuries;
    int    queueQuries;
    int    hostQuries;
    int    submissionRequest;
    int    jobSubmitted;
    int    dispatchedjobs;
```

PARAMETERS

```
    int     jobcompleted;
    int     jobMCSend;
    int     jobMCReceive;
    time_t startTime;
};
struct eventEOSLog {      /* Event end of stream. */
    int    eos;
      };
```

PARAMETERS

***logPtr** The `eventRec` structure pointed to by `logPtr` into a log file.

***log_fp** A pointer pointing to the log file name that could be either event a log file or job log file.

RETURN VALUES

integer:0 The function was successful, and the information of `eventRec` structure has been put into log file pointed by `log_fp`

integer:-1 Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_geteventrec()`

Equivalent line command

none

Files

`lsb.events`

`lsb.acct`

lsb_queuecontrol()

Changes the status of a queue.

DESCRIPTION

`lsb_queuecontrol()` changes the status of a queue.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

If a queue is inactivated by its dispatch window (see `lsb.queues`), then it cannot be re-activated by this call.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_queuecontrol (struct queueCtrlReq *req)
struct queueCtrlReq {
  char *queue
  int   opCode
  char *message
```

PARAMETERS

***queue**	The name of the queue to be controlled.
opCode	One of the following operation codes:

QUEUE_OPEN

Open the queue to accept jobs.

QUEUE_CLOSED

Close the queue so it will not accept jobs.

QUEUE_ACTIVATE

Activate the queue to dispatch jobs.

QUEUE_INACTIVATE

Inactivate the queue so it will not dispatch jobs.

message	The message attached by the administrator.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_queueinfo()` - Returns information about queues

Equivalent line command

none

Files

`lsf.conf`

lsb_queueinfo()

Returns information about batch queues.

DESCRIPTION

`lsb_queueinfo()` gets information about batch queues. See `lsb.queues` for more information about queue parameters.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct queueInfoEnt *lsb_queueinfo(char **queues,
                     int *numQueues, char *hosts, char *users,
                     int options)
struct queueInfoEnt {
    char   *queue;
    char   *description;
    int    priority;
    short  nice;
    char   *userList;
    char   *hostList;
    char   *hostStr;
    int    nIdx;
    float  *loadSched;
    float  *loadStop;
    int    userJobLimit;
    float  procJobLimit;
    char   *windows;
    int    rLimits[LSF_RLIM_NLIMITS];
    char   *hostSpec;
    int    qAttrib;
    int    qStatus;
    int    maxJobs;
    int    numJobs;
    int    numPEND;
    int    numRUN;
    int    numSSUSP;
    int    numUSUSP;
    int    mig;
    int    schedDelay;
    int    acceptIntvl;
    char   *windowsD;
    char   *nqsQueues;
    char   *userShares;
```

SYNOPSIS

```
char   *defaultHostSpec;
int    procLimit;
char   *admins;
char   *preCmd;
char   *postCmd;
char   *requeueEValues;
int    hostJobLimit;
char   *resReq;
int    numRESERVE;
int    slotHoldTime;
char   *sndJobsTo;
char   *rcvJobsFrom;
char   *resumeCond;
char   *stopCond;
char   *jobStarter;
char   *suspendActCmd;
char   *resumeActCmd;
char   *terminateActCmd;
int    sigMap[LSB_SIG_NUM];
char   *preemption;
int    maxRschedTime;
int    numOfSAccts;
struct shareAcctInfoEnt*  shareAccts;
char   *chkpntDir;
int    chkpntPeriod;
int    imptJobBklg;
int    defLimits[LSF_RLIM_NLIMITS];
int    chunkJobSize;
int    minProcLimit;
int    defProcLimit;
char   *fairshareQueues;
char   *defExtSched;
char   *mandExtSched;
int    slotShare;
char   *slotPool;
int    underRCond;
int    overRCond;
float  idleCond;
int    underRJobs;
int    overRJobs;
int    idleJobs;
int    warningTimePeriod;
char   *warningAction;
```

```
    char   *qCtrlMsg;
    char   *acResReq;
    int    symJobLimit;
    char   *cpuReq;
    int    proAttr;
    int    lendLimit;
    int    hostReallocInterval;
    int    numCPURequired;
    int    numCPUAllocated;
    int    numCPUBorrowed;
    int    numCPULent;
    int    schGranularity;
    int    symTaskGracePeriod;
    int    minOfSsm;
    int    maxOfSsm;
    int    numOfAllocSlots;
    char *servicePreemption;
    int    provisionStatus;
    int    minTimeSlice;
    char   *queueGroup;
    int    numApsFactors;
    struct apsFactorInfo *apsFactorInfoList;
    struct apsFactorMap  *apsFactorMaps;
    struct apsLongNameMap *apsLongNames;
    int    maxJobPreempt;
    int    maxPreExecRetry;
    int    localMaxPreExecRetry;
    int    maxJobRequeue;
    int    usePam;
    int    cu_type_exclusive;
    char   cu_str_exclusive;
};
struct shareAcctInfoEnt {
    char   *shareAcctPath;
    int    shares;
    float  priority;
    int    numStartJobs;
    float  histCpuTime;
    int    numReserveJobs;
    int    runTime;
    int    shareAdjustment;
};
```

PARAMETERS

****queues**	An array of names of queues of interest.
***numQueues**	The number of queue names. To get information on all queues, set `*numQueues` to 0; `*numQueues` will be updated to the actual number of queues when this call returns. If `*numQueues` is 1 and `queues` is `NULL`, information on the system default queue is returned.
***hosts**	The host or cluster names. If `hosts` is not `NULL`, then only the queues that are enabled for the `hosts` are of interest.
***user**	The name of user. If `user` is not `NULL`, then only the queues that are enabled for the `user` are of interest.
options	Reserved for future use; supply 0.

queueInfoEnt structure fields

The `queueInfoEnt` structure contains the following fields:

queue	The name of the queue.
description	Describes the typical use of the queue.
priority	Defines the priority of the queue. This determines the order in which the job queues are searched at job dispatch time: queues with higher priority values are searched first. (This is contrary to UNIX process priority ordering.)
nice	Defines the `nice` value at which jobs in this queue will be run.
userList	A blank-separated list of names of users allowed to submit jobs to this queue.
hostList	A blank-separated list of names of hosts to which jobs in this queue may be dispatched.
hostStr	Original HOSTS string in case "-" is used.
nIdx	The number of load indices in the `loadSched` and `loadStop` arrays.
loadSched & loadStop	The queue and host `loadSched` and `loadStop` arrays control batch job dispatch, suspension, and resumption. The values in the `loadSched` array specify thresholds for the corresponding load indices. Only if the current values of all specified load indices of a host are within (below or above, depending on the meaning of the load index) the corresponding thresholds of this queue, will jobs in this queue be dispatched to the host. The same conditions are used to resume jobs dispatched from this queue that have been suspended on the host. Similarly, the values in the `loadStop` array specify the thresholds for job suspension. If any of the current load index values of a host goes beyond a queue's threshold, jobs from the queue will be suspended. For an explanation of the fields in the `loadSched` and `loadStop` arrays, see `lsb_hostinfo()`.
userJobLimit	Per-user limit on the number of jobs that can be dispatched from this queue and executed concurrently.

procJobLimit Per-processor limit on the number of jobs that can be dispatched from this queue and executed concurrently.

windows A blank-separated list of time windows describing the run window of the queue. When a queue's run window is closed, no job from this queue will be dispatched. When the run window closes, any running jobs from this queue will be suspended until the run window reopens, when they will be resumed. The default is no restriction, or always open (i.e., 24 hours a day, seven days a week).

A time window has the format `begin_time–end_time`. Time is specified in the format `[day:]hour[:minute]`, where all fields are numbers in their respective legal ranges: 0(Sunday)-6 for `day`, 0-23 for `hour`, and 0-59 for `minute`. The default value for `minute` is 0 (on the hour); the default value for `day` is every day of the week. The `begin_time` and `end_time` of a window are separated by '–', with no white space (i.e., blank or `TAB`) in between. Both `begin_time` and `end_time` must be present for a time window.

Note that this run window only applies to batch jobs; interactive jobs scheduled by the LSF Load Information Manager (LIM) are controlled by another set of run windows.

rLimits[LSF_RLIM_NLIMITS]

The per-process UNIX hard resource limits for all jobs submitted to this queue (see `getrlimit()` and `lsb.queues`). The default values for the resource limits are unlimited, indicated by -1. The constants used to index the `rLimits` array and the corresponding resource limits are listed below.

- LSF_RLIMIT_CPU CPULIMIT
- LSF_RLIMIT_FSIZE FILELIMIT
- LSF_RLIMIT_DATA DATALIMIT
- LSF_RLIMIT_STACK STACKLIMIT
- LSF_RLIMIT_CORE CORELIMIT
- LSF_RLIMIT_RSS MEMLIMIT
- LSF_RLIMIT_RUN RUNLIMIT
- LSF_RLIMIT_PROCESS PROCESSLIMIT
- LSF_RLIMIT_SWAP SWAPLIMIT
- LSF_RLIMIT_THREAD
- LSF_RLIMIT_NOFILE
- LSF_RLIMIT_OPENMAX
- LSF_RLIMIT_VMEM

hostSpec A host name or host model name. If the queue `CPULIMIT` or `RUNLIMIT` gives a host specification, `hostSpec` will be that specification. Otherwise, if `defaultHostSpec` (see below) is not `NULL`, `hostSpec` will be `defaultHostSpec`. Otherwise, if `DEFAULT_HOST_SPEC` is defined in the `lsb.params` file, (see `lsb.params`), `hostSpec` will be this value. Otherwise, `hostSpec` will be the name of the host with the largest CPU factor in the cluster.

qAttrib The attributes of the queue. The bitwise inclusive OR of some of the following:

Q_ATTRIB_EXCLUSIVE

This queue accepts jobs which request exclusive execution.

Q_ATTRIB_DEFAULT

This queue is a default queue of LSF.

Q_ATTRIB_FAIRSHARE

This queue uses the FAIRSHARE scheduling policy. The user shares are given in `userShares`. (See below.)

Q_ATTRIB_PREEMPTIVE

This queue uses the PREEMPTIVE scheduling policy.

Q_ATTRIB_NQS

This is an NQS forward queue. The target NQS queues are given in `nqsQueues`. (See below.) For NQS forward queues, the `hostList`, `procJobLimit`, `windows`, `mig` and `windowsD` fields are meaningless.

Q_ATTRIB_RECEIVE

This queue can receive jobs from other clusters.

Q_ATTRIB_PREEMPTABLE

This queue uses a preemptable scheduling policy.

Q_ATTRIB_BACKFILL

This queue uses a backfilling policy.

Q_ATTRIB_HOST_PREFER

This queue uses a host preference policy.

Q_ATTRIB_NONPREEMPTIVE

This queue can't preempt any other another queue.

Q_ATTRIB_NONPREEMPTABLE

This queue can't be preempted from any queue.

Q_ATTRIB_NO_INTERACTIVE

This queue does not accept batch interactive jobs.

Q_ATTRIB_ONLY_INTERACTIVE

This queue only accepts batch interactive jobs.

Q_ATTRIB_NO_HOST_TYPE

No host type related resource name specified in resource requirement.

Q_ATTRIB_ IGNORE_DEADLINE

This queue disables deadline constrained resource scheduling.

Q_ATTRIB_ CHKPNT

Jobs may run as checkpointable.

Q_ATTRIB_ RERUNABLE

Jobs may run as rerunnable.

Q_ATTRIB_MC_FAST_SCHEDULE

Turn on multicluster fast scheduling policy.

Q_ATTRIB_ENQUE_INTERACTIVE_AHEAD

Push interactive jobs in front of other jobs in queue.

Q_MC_FLAG

Flags used by MultiCluster.

Q_ATTRIB_LEASE_LOCAL

Lease and local.

Q_ATTRIB_LEASE_ONLY

Lease only; no local.

Q_ATTRIB_RMT_BATCH_LOCAL

Remote batch and local.

Q_ATTRIB_RMT_BATCH_ONLY

Remote batch only.

Q_ATTRIB_RESOURCE_RESERVE

Memory reservation.

Q_ATTRIB_FS_DISPATCH_ORDER_QUEUE

Cross-queue fairshare.

Q_ATTRIB_BATCH

Batch queue/partition.

Q_ATTRIB_ONLINE

Online partition.

Q_ATTRIB_INTERRUPTIBLE_BACKFILL

Interruptible backfill.

Q_ATTRIB_APS

This queue sets an absolute priority scheduling (APS) value.

qStatus The status of the queue. It is the bitwise inclusive OR of some of the following values:

QUEUE_STAT_OPEN

The queue is open to accept newly submitted jobs.

QUEUE_STAT_ACTIVE

The queue is actively dispatching jobs. The queue can be inactivated and reactivated by the LSF administrator using `lsb_queuecontrol()`. The queue will also be inactivated when its run or dispatch window is closed. In this case it cannot be reactivated manually; it will be reactivated by the LSF system when its run and dispatch windows reopen.

QUEUE_STAT_RUN

The queue run and dispatch windows are open.

The initial state of a queue at LSF boot time is `open` and either `active` or `inactive`, depending on its run and dispatch windows.

	QUEUE_STAT_NOPERM Remote queue rejecting jobs.
	QUEUE_STAT_DISC Remote queue status is disconnected.
	QUEUE_STAT_RUNWIN_CLOSE Queue run windows are closed.
maxJobs	The maximum number of jobs dispatched by the queue and not yet finished.
numJobs	Number of jobs in the queue, including pending, running, and suspended jobs.
numPEND	Number of pending jobs in the queue.
numRUN	Number of running jobs in the queue.
numSSUSP	Number of system suspended jobs in the queue.
numUSUSP	Number of user suspended jobs in the queue.
mig	The queue migration threshold in minutes.
schedDelay	The number of seconds that a new job waits, before being scheduled. A value of zero (0) means the job is scheduled without any delay.
acceptIntvl	The number of seconds for a host to wait after dispatching a job to a host, before accepting a second job to dispatch to the same host.
windowsD	A blank-separated list of time windows describing the dispatch window of the queue. When a queue's dispatch window is closed, no job from this queue will be dispatched. The default is no restriction, or always open (i.e., 24 hours a day, seven days a week). For the time window format, see `window` above.
nqsQueues	A blank-separated list of queue specifiers. Each queue specifier is of the form `queue@host` where `host` is an NQS host name and `queue` is the name of a queue on that host.
userShares	A blank-separated list of user shares. Each share is of the form `[user, share]` where `user` is a user name, a user group name, the reserved word `default` or the reserved word `others`, and `share` is the number of shares the user gets.
defaultHostSpec	The value of `DEFAULT_HOST_SPEC` in the `Queue` section for this queue in the `lsb.queues` file.
procLimit	An LSF resource limit used to limit the number of job slots (processors) a (parallel) job in the queue will use. A job submitted to this queue must specify a number of processors not greater than this limit.
admins	A list of administrators of the queue. The users whose names are here are allowed to operate on the jobs in the queue and on the queue itself.
preCmd	Queue's pre-exec command. The command is executed before the real batch job is run on the execution host (or on the first host selected for a parallel batch job).
postCmd	Queue's post-exec command. The command is run when a job terminates.
requeueEValues	Jobs that exit with these values are automatically requeued.

hostJobLimit	The maximum number of job slots a host can process from this queue, including job slots of dispatched jobs which have not finished yet and reserved slots for some PEND jobs. This limit controls the number of jobs sent to each host, regardless of a uniprocessor host or multiprocessor host. Default value for this limit is infinity.
resReq	Resource requirement string used to determine eligible hosts for a job.
numRESERVE	Number of reserved job slots for pending jobs.
slotHoldTime	The time used to hold the reserved job slots for a PEND job in this queue.
sndJobsTo	Remote MultiCluster send-jobs queues to forward jobs to.
rcvJobsFrom	Remote MultiCluster receive-jobs queues that can forward to this queue.
resumeCond	Resume threshold conditions for a suspended job in this queue.
stopCond	Stop threshold conditions for a running job in this queue.
jobStarter	Job starter command for a running job in this queue.
suspendActCmd	Command configured for the SUSPEND action.
resumeActCmd	Command configured for the RESUME action.
terminateActCmd	Command configured for the TERMINATE action.
preemption	Preemptive scheduling and preemption policy specified for the queue.
maxRschedTime	Time period for a remote cluster to schedule a job. MultiCluster job forwarding model only. Determines how long a MultiCluster job stays pending in the execution cluster before returning to the submission cluster. The remote timeout limit in seconds is: `MAX_RSCHED_TIME * MBD_SLEEP_TIME=timeout`
numOfSAccts, shareAccts	(Only used for queues with fairshare policy) a share account vector capturing the fairshare information of the users using the queue. The storage for the array of `queueInfoEnt` structures will be reused by the next call.
chkpntDir	The directory where the checkpoint files are created.
chkpntPeriod	The checkpoint period in minutes.
imptJobBklg	MultiCluster job forwarding model only. Specifies the MultiCluster pending job limit for a receive-jobs queue. This represents the maximum number of MultiCluster import jobs that can be pending in the queue; once the limit has been reached, the queue stops accepting jobs from remote clusters.
defLimits[LSF_RLIM_NLIMITS]	The default (soft) resource limits for all jobs submitted to this queue (see `getrlimit()` and `lsb.queues`).
chunkJobSize	The maximum number of jobs allowed to be dispatched together in one job chunk. Must be a positive integer greater than 1.
minProcLimit	The minimum number of job slots (processors) that a job in the queue will use.
defProcLimit	The default (soft) limit on the number of job slots (processors) that a job in the queue will use.
fairshareQueues	The list of queues for cross-queue fairshare.

defExtSched	Default external scheduling options for the queue.
mandExtSched	Mandatory external scheduling options for the queue.
slotShare	Share of job slots for queue-based fairshare. Represents the percentage of running jobs (job slots) in use from the queue. SLOT_SHARE must be greater than zero (0) and less than or equal to 100. The sum of SLOT_SHARE for all queues in the pool does not need to be 100%. It can be more or less, depending on your needs.
slotPool	Name of the pool of job slots the queue belongs to for queue-based fairshare. A queue can only belong to one pool. All queues in the pool must share the same set of hosts. Specify any ASCII string up to 60 characters long. You can use letters, digits, underscores (_) or dashes (-). You cannot use blank spaces.
underRCond	Specifies a threshold for job underrun exception handling. If a job exits before the specified number of minutes, LSF invokes `LSF_SERVERDIR/eadmin` to trigger the action for a job underrun exception.
overRCond	Specifies a threshold for job overrun exception handling. If a job runs longer than the specified run time, LSF invokes `LSF_SERVERDIR/eadmin` to trigger the action for a job overrun exception.
idleCond	Specifies a threshold for idle job exception handling. The value should be a number between 0.0 and 1.0 representing CPU time/runtime. If the job idle factor is less than the specified threshold, LSF invokes `LSF_SERVERDIR/eadmin` to trigger the action for a job idle exception.
underRJobs	The number of underrun jobs in the queue.
overRJobs	The number of overrun jobs in the queue.
idleJobs	The number of idle jobs in the queue.
warningTimePeriod	Specifies the amount of time before a job control action occurs that a job warning action is to be taken. For example, 2 minutes before the job reaches run time limit or termination deadline, or the queue's run window is closed, an URG signal is sent to the job. Job action warning time is not normalized. A job action warning time must be specified with a job warning action in order for job warning to take effect.
warningAction	Specifies the job action to be taken before a job control action occurs. For example, 2 minutes before the job reaches run time limit or termination deadline, or the queue's run window is closed, an URG signal is sent to the job. A job warning action must be specified with a job action warning time in order for job warning to take effect. If specified, LSF sends the warning action to the job before the actual control action is taken. This allows the job time to save its result before being terminated by the job control action. You can specify actions similar to the JOB_CONTROLS queue level parameter: send a signal, invoke a command, or checkpoint the job.

AcResReq	
symJobLimit	Limit of running session scheduler jobs.
cpuReq	cpu_req for service partition of session scheduler.
proAttr	Indicate whether it would be willing to donate/borrow.
lendLimit	The maximum number of hosts to lend.
hostReallocInterval	The grace period to lend/return idle hosts.
numCPURequired	Number of CPUs required by CPU provision.
numCPUAllocated	Number of CPUs actually allocated.
numCPUBorrowed	Number of CPUs borrowed.
numCPULent	Number of CPUs lent.
schGranularity	The scheduling granularity. in milliseconds.
symTaskGracePeriod	The grace period for stopping session scheduler tasks.
minOfSsm	The minimum number of SSMs.
maxOfSsm	The maximum number of SSMs.
numOfAllocSlots	The number of allocated slots.
servicePreemption	The service preemption policy.
provisionStatus	Dynamic CPU provision status.
minTimeSlice	The minimum time for preemption and backfill, in seconds.
queueGroup	List of queues defined in a queue group for absolute priority scheduling (APS) across multiple queues.
numApsFactors	The number of calculation factors for absolute priority scheduling (APS).
apsFactorInfo	List of calculation factors for absolute priority scheduling (APS).
apsFactorMap	The mapping of factors to subfactors for absolute priority scheduling (APS).
apsLongNameMap	The mapping of factors to their long names for absolute priority scheduling (APS).
maxJobPreempt	Maximum number of job preempted times.
maxPreExecRetry	Maximum number of pre-exec retry times.
localMaxPreExecRetry	Maximum number of pre-exec retry times for local cluster.
maxJobRequeue	Maximum number of job re-queue times.
usePam	Use Linux-PAM.
cu_type_exclusive	Compute unit type.
cu_str_exclusive	A string specified in EXCLUSIVE+CU[<string>].

shareAcctInfoEnt structure fields

The `shareAcctInfoEnt` structure contains the following fields:

shareAcctPath	The user name or user group name. (See `lsb_userinfo()` and `lsb_usergrpinfo()`.)

PARAMETERS

shares	The number of shares assigned to the user or user group, as configured in the file `lsb.queues`.
numStartJobs	The number of job slots (belonging to the user or user group) that are running or suspended in the fairshare queue.
histCpuTime	The normalized CPU time accumulated in the fairshare queue by jobs belonging to the user or user group, over the time period configured in the file `lsb.params`. The default time period is 5 hours.
priority	The priority of the user or user group in the fairshare queue. Larger values represent higher priorities. Job belonging to the user or user group with the highest priority are considered first for dispatch in the fairshare queue. In general, a user or user group with more `shares`, fewer `numStartJobs` and less `histCpuTime` has higher priority.
numReserveJobs	The number of job slots that are reserved for the `PEND` jobs belonging to the user or user group in the host partition.
runTime	The time unfinished jobs spend in the RUN state.
shareAdjustment	The fairshare adjustment value from the fairshare plugin (`libfairshareadjust.*`). The adjustment is enabled and weighted by setting the value of `FAIRSHARE_ADJUSTMENT_FACTOR` in `lsb.params`.

RETURN VALUES

array: queueInfoEnt	An array of `queueInfoEnt` structures which store the queue information and sets `*numQueues` to the size of the array.
char:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error. If `lsberrno` is `LSBE_BAD_QUEUE`, `(*queues)[*numQueues]` is not a queue known to the LSF system. Otherwise, if `*numQueues` is less than its original value, `*numQueues` is the actual number of queues found.

SEE ALSO

Related APIs

`lsb_hostinfo()` - Get information about job server hosts

`lsb_userinfo()` - Get information about users

`lsb_usergrpinfo()` - Get information about user groups

Equivalent command

`bqueues`

Files

`$LSB_CONFDIR/`*cluster*`_name/lsb.queues`

lsb_readjobinfo()

Returns the next job information record in `mbatchd`.

DESCRIPTION

`lsb_readjobinfo()` reads the number of records defined by the `more` parameter. The `more` parameter receives its value from either `lsb_openjobinfo()` or `lsb_openjobinfo_a()`. Each time `lsb_readjobinfo()` is called, it returns one record from `mbatchd`. Use `lsb_readjobinfo()` in a loop and use `more` to determine how many times to repeat the loop to retrieve job information records.

SYNOPSIS

```
#include <lsf/lsbatch.h>
#include <time.h>
#include <lsf/lsf.h>
struct jobInfoEnt *lsb_readjobinfo(int *)
struct jobInfoEnt {
    LS_LONG_INT jobId;
    char    *user;
    int     status;
    int     *reasonTb;
    int     numReasons;
    int     reasons;
    int     subreasons;
    int     jobPid;
    time_t  submitTime;
    time_t  reserveTime;
    time_t  startTime;
    time_t  predictedStartTime;
    time_t  endTime;
    time_t  lastEvent;
    time_t  nextEvent;
    int     duration;
    float   cpuTime;
    int     umask;
    char    *cwd;
    char    *subHomeDir;
    char    *fromHost;
    char    **exHosts;
    int     numExHosts;
    float   cpuFactor;
    int     nIdx;
    float   *loadSched;
    float   *loadStop;
    struct  submit submit;
    int     exitStatus;
    int     execUid;
    char    *execHome;
    char    *execCwd;
    char    *execUsername;
    time_t  jRusageUpdateTime;
    struct  jRusage runRusage;
    int     jType;
    char    *parentGroup;
    char    *jName;
    int     counter[NUM_JGRP_COUNTERS];
    u_short port;
    int     jobPriority;
    int numExternalMsg;
    struct jobExternalMsgReply **externalMsg;
    int     clusterId;
    char   *detailReason;
    float   idleFactor;
    int     exceptMask;
    char   *additionalInfo;
    int     exitInfo;
    int    warningTimePeriod;
    char   *warningAction;
    char   *chargedSAAP;
    char   *execRusage;
    time_t rsvInActive;
    int    numLicense;
    char   **licenseNames;
    float  aps;
    float  adminAps;
    int runTime
    int reserveCnt
    struct reserveItem *items;
    float  adminFactorVal;
    int    resizeMin
    int    resizeMax
    time_t resizeReqTime
    int    jStartNumExHosts
    char   **jStartExHosts
    time_t lastResizeTime
```

```
};
struct submit {
    int     options;
    int     options2;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    int     numProcessors;
    char    *dependCond;
    char    *timeEvent;
    time_t  beginTime;
    time_t  termTime;
    int     sigValue;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *newCommand;
    time_t  chkpntPeriod;
    char    *chkpntDir;
    int     nxf;
    struct xFile *xf;
    char    *preExecCmd;
    char    *mailUser;
    int     delOptions;
    int     delOptions2;
    char    *projectName;
    int     maxNumProcessors;
    char    *loginShell;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *jobGroup;
    char    *sla;
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    int     jsdlFlag;
    char    *jsdlDoc;
    void    *correlator;
    char    *apsString;
    char    *postExecCmd;
    char    *cwd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
    char    *notifyCmd;
};
struct jRusage{
    int mem;
    int swap;
    int utime;
    int stime;
    int npids;
    struct pidInfo *pidInfo;
    int npgids;
    int *pgid;
    int nthreads;
};
struct pidInfo{
    int pid;
    int ppid;
    int pgid;
    int jobid;
};
struct reserveItem {
    char    *resName;
    int     nHost;
    float   *value;
    int     shared;
};
```

PARAMETERS

***more**	Number of job records in the master batch daemon.

RETURN VALUES

jobInfoEnt Function was successful.

The fields in the `jobInfoEnt` structure have the following meaning:

jobId

The job ID that the LSF system assigned to the job.

user

The name of the user who submitted the job.

status

The current status of the job. Possible values are:

JOB_STAT_PEND: The job is pending, i.e., it has not been dispatched yet.

JOB_STAT_PSUSP: The pending job was suspended by its owner or the LSF system administrator.

JOB_STAT_RUN: The job is running.

JOB_STAT_SSUSP: The running job was suspended by the system because an execution host was overloaded or the queue run window closed. (See lsb_queueinfo(), lsb_hostinfo(), and `lsb.queues`.)

JOB_STAT_USUSP: The running job was suspended by its owner or the LSF system administrator.

JOB_STAT_EXIT: The job has terminated with a non-zero status – it may have been aborted due to an error in its execution, or killed by its owner or by the LSF system administrator.

JOB_STAT_DONE: The job has terminated with status 0.

JOB_STAT_PDONE: Post job process done successfully.

JOB_STAT_PERR: TPost job process has error.

JOB_STAT_WAIT: Chunk job waiting its turn to execute.

JOB_STAT_UNKWN: The slave batch daemon (sbatchd) on the host on which the job is processed has lost contact with the master batch daemon (mbatchd).

reasonTb

Pending or suspending reasons of the job.

numReasons

Length of reasonTb vector.

reasons

The reason a job is pending or suspended.

subreasons

The reason a job is pending or suspended. If status is JOB_STAT_PEND, the values of reasons and subreasons are explained by lsb_pendreason(). If status is JOB_STAT_PSUSP, the values of reasons and subreasons are explained by lsb_suspreason().

When reasons is PEND_HOST_LOAD or SUSP_LOAD_REASON, subreasons indicates the load indices that are out of bounds. If reasons is PEND_HOST_LOAD, subreasons is the same as busySched in the hostInfoEnt structure; if reasons is SUSP_LOAD_REASON, subreasons is the same as busyStop in the hostInfoEnt structure. (See lsb_hostinfo().)

jobPid

The job process ID.

submitTime

The time the job was submitted, in seconds since 00:00:00 GMT, Jan. 1, 1970.

reserveTime

Time when job slots are reserved

startTime

The time that the job started running, if it has been dispatched.

PredictedStartTime

Job's predicted start time

endTime

The termination time of the job, if it has completed.

LastEvent

Last time event.

nextEvent

Next time event.

duration

Duration time (minutes).

cpuTime

The CPU time that the job has used.

umask

The file creation mask when the job was submitted.

cwd

The current working directory when the job was submitted.

subHomeDir

Home directory on submission host.

fromHost

The name of the host from which the job was submitted.

exHosts

The array of names of hosts on which the job executes.

numExHosts

The number of hosts on which the job executes.

cpuFactor

The CPU factor for normalizing CPU and wall clock time limits.

nIdx

The number of load indices in the loadSched and loadStop arrays.

loadSched & loadStop

The `loadSched` and `loadStop` arrays are assigned to the job according to those of the queue and hosts to control job suspension and resumption.

The values in the `loadSched` array specify the thresholds for the corresponding load indices. Only if the current values of all specified load indices of a host are within (below or above, depending on the meaning of the load index) their corresponding thresholds may the suspended job be resumed on this host.

Similarly, the values in the `loadStop` array specify the thresholds for job suspension; if any of the current load index values of the host crosses its threshold, the job will be suspended.

For an explanation of the entries in the `loadSched` and `loadStop` arrays, see `lsb_hostinfo()`.

submit

Structure for `lsb_submit()` call.

exitStatus

Job exit status.

execUid

Mapped UNIX user ID on the execution host.

execHome

Home directory for the job on the execution host.

execCwd

Current working directory for the job on the execution host.

execUsername

Mapped user name on the execution host.

jRusageUpdateTime

Time of the last job resource usage update.

jRusage

Contains resource usage information for the job.

jType

Job type.

parentGroup

The parent job group of a job or job group.

jName

if jType is JGRP_NODE_GROUP, then it is the job group name. Otherwise, it is the job name.

counter[NUM_JGRP_COUNTERS]

Index into the counter array. Only used for job arrays:

- JGRP_COUNT_NJOBS—total jobs in the array
- JGRP_COUNT_PEND—number of pending jobs in the array
- JGRP_COUNT_NPSUSP—number of held jobs in the array
- JGRP_COUNT_NRUN—number of running jobs in the array
- JGRP_COUNT_NSSUSP—number of jobs suspended by the system in the array
- JGRP_COUNT_NUSUSP—number of jobs suspended by the user in the array
- JGRP_COUNT_NEXIT—number of exited jobs in the array
- JGRP_COUNT_NDONE—number of successfully completed jobs
- JGRP_COUNT_NJOBS_SLOTS—total slots in the array
- JGRP_COUNT_PEND_SLOTS—number of pending slots in the array
- JGRP_COUNT_RUN_SLOTS—number of running slots in the array
- JGRP_COUNT_SSUSP_SLOTS—number of slots suspended by the system in the array
- JGRP_COUNT_USUSP_SLOTS— number of slots suspended by the user in the array
- JGRP_COUNT_RESV_SLOTS—number of reserverd slots in the array

port

Service port of the job.

jobPriority

Job dynamic priority.

numExternalMsg

The number of external messages in the job.

jobExternalMsgReply

This structure contains the information required to define an external message reply.

clusterId

MultiCluster cluster ID. If clusterId is greater than or equal to 0, the job is a pending remote job, and `lsb_readjobinfo` checks for `host_name@cluster_name`. If host name is needed, it should be found in `jInfoH->remoteHosts`. If the remote host name is not available, the constant string `remoteHost` is used.

detailReason

Detailed reason field.

idleFactor

Idle factor for job exception handling. If the job idle factor is less than the specified threshold, LSF invokes `LSF_SERVERDIR/eadmin` to trigger the action for a job idle exception.

exceptMask

Job exception handling mask.

additionalInfo

Placement information of LSF HPC jobs.

exitInfo

Job termination reason. See `lsbatch.h`.

warningTimePeriod

Job warning time period in seconds; -1 if unspecified.

warningAction

Job warning action, SIGNAL | CHKPNT | command; NULL if unspecified.

chargedSAAP

SAAP charged for job.

execRusage

The rusage satisfied at job runtime.

rsvInActive

The time when advance reservation expired or was deleted.

numLicense

The number of licenses reported from License Scheduler.

licenseNames

License Scheduler license names.

aps

Absolute priority scheduling (APS) priority value.

adminAps

Absolute priority scheduling (APS) string set by administrators to denote static system APS value

adminFactorVal

Absolute priority scheduling (APS) string set by administrators to denote ADMIN factor APS value.

runTime

The real runtime on the execution host.

reserveCnt

How many kinds of resource are reserved by this job

reserveItem

The `reserveItem` structure contains the following fields:

resname: Name of the resource to reserve.

items : Details reservation information for each kind of resource.

value: Amount of reservation is made on each host. Some hosts may reserve 0.

nhost: The number of hosts to reserve this resource.

shared: Flag for shared or host-base resource.

resizeMin: Pending resize min. 0, if no resize pending.

resizeMax: Pending resize max. 0, if no resize pending.

resizeReqTime: Time when pending request was issued.

jStartNumExHosts: Number of hosts when job starts.

jStartExHosts: Host list when job starts.

lastResizeTime: Last time when job allocation changed.

The fields in the submit structure:

submit `submit` uses the submit structure provided by the invoker of `lsb_submit()`.

See lsb_submit() on page 377 for descriptions of the `submit` structure fields.

The fields in the `runRusage` structure have the following meaning:

runRusage `runRusage` uses the `jRusage` structure to provide the total resident memory usage in KB, total virtual memory usage inKB, cumulative total CPU time in seconds and a list of currently active process group IDs and process IDs in a job.

The `jRusage` structure contains the following fields:

mem

Total resident memory usage in KB of all currently running processes in given process groups.

swap

Total virtual memory usage in KB of all currently running processes in given proces groups.

utime

Cumulative total user time in seconds.

stime

Cumulative total system time in seconds.

npids

Number of currently active processesin given process groups.

npgids

Number of currently active process groups

pgid

Array of currently active process group ids

nthreads

Number of currently active threads in given process groups.

The fields in the pidInfo structure have the following meaning:

pidInfo

Structure containing information about an active process.

pid

Process id.

ppid

Parent's process id.

pgid

Process group id.

jobid

Process Cray job ID (only on Cray).

ERRORS

If there are no more records, then `lsberrno` is set to LSBE_EOF.

SEE ALSO

Related API

`lsb_openjobinfo()` - Opens a job information connection to`mbatchd`

`lsb_openjobinfo_a()` - Provides the name and number of jobs and hosts in`mbatchd`

`lsb_closejobinfo()` - Closes job information connection with `mbatchd`

`lsb_hostinfo()` - Returns informaton about job server hosts

`lsb_pendreason()` - Explains why a job is pending

`lsb_queueinfo()` - Returns information about batch queues

`lsb_suspreason()` - Explains why a job was suspended

Equivalent line command

none

Files

`lsb.queues`

lsb_readjobinfo_cond()

Returns the next job information record for condensed host groups in `mbatchd`.

DESCRIPTION

`lsb_readjobinfo_cond()` reads the number of records defined by the `more` parameter. The `more` parameter receives its value from either `lsb_openjobinfo()` or `lsb_openjobinfo_a()`. Each time `lsb_readjobinfo_cond()` is called, it returns one record from `mbatchd`. Use `lsb_readjobinfo_cond()` in a loop and use more to determine how many times to repeat the loop to retrieve job information records.

`lsb_readjobinfo_cond()` differs from `lsb_readjobinfo()` in that if `jInfoHExt` is not `NULL`, `lsb_readjobinfo_cond()` substitutes hostGroup (if it is a condensed host group) for job execution hosts.

SYNOPSIS

```
#include <lsf/lsbatch.h>
#include <time.h>
#include <lsf/lsf.h>
struct jobInfoEnt *lsb_readjobinfo_cond(int *more,
                   struct jobInfoHeadExt *jInfoHExt);
struct_jobInfoEnt {
    LS_LONG_INT jobId;
    char    *user;
    int     status;
    int     *reasonTb;
    int     numReasons;
    int     reasons;
    int     subreasons;
    int     jobPid;
    time_t  submitTime;
    time_t  reserveTime;
    time_t  startTime;
    time_t  predictedStartTime;
    time_t  endTime;
    time_t  lastEvent;
    time_t  nextEvent;
    int     duration;
    float   cpuTime;
    int     umask;
    char    *cwd;
    char    *subHomeDir;
    char    *fromHost;
```

SYNOPSIS

```
char    **exHosts;
int     numExHosts;
float   cpuFactor;
int     nIdx;
float   *loadSched;
float   *loadStop;
struct  submit submit;
int     exitStatus;
int     execUid;
char    *execHome;
char    *execCwd;
char    *execUsername;
time_t  jRusageUpdateTime;
struct  jRusage runRusage;
int     jType;
char    *parentGroup;
char    *jName;
int     counter[NUM_JGRP_COUNTERS];
u_short port;
int     jobPriority;
int numExternalMsg;
struct jobExternalMsgReply **externalMsg;
int     clusterId;
char   *detailReason;
float   idleFactor;
int     exceptMask;
char   *additionalInfo;
int     exitInfo;
int    warningTimePeriod;
char   *warningAction;
char   *chargedSAAP;
char   *execRusage;
time_t rsvInActive;
int    numLicense;
char   **licenseNames;
float  aps;
int runTime
int reserveCnt
struct reserveItem *items;
float  adminFactorVal;
int    resizeMin
int    resizeMax
time_t resizeReqTime
```

```
    int    jStartNumExHosts
    char   **jStartExHosts
    time_t lastResizeTime
struct reserveItem *items;
};
struct submit {
    int     options;
    int     options2;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    int     numProcessors;
    char    *dependCond;
    char    *timeEvent;
    time_t  beginTime;
    time_t  termTime;
    int     sigValue;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *newCommand;
    time_t  chkpntPeriod;
    char    *chkpntDir;
    int     nxf;
    struct xFile *xf;
    char    *preExecCmd;
    char    *mailUser;
    int     delOptions;
    int     delOptions2;
    char    *projectName;
    int     maxNumProcessors;
    char    *loginShell;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *jobGroup;
    char    *sla;
```

SYNOPSIS

```
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    int     jsdlFlag;
    char    *jsdlDoc;
    void    *correlator;
    char    *apsString;
    char    *postExecCmd;
    char    *cwd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
    char    *notifyCmd;
};
struct jRusage{
    int mem;
    int swap;
    int utime;
    int stime;
    int npids;
    struct pidInfo;
    int npgids;
    int *pgid;
int nthreads;
};
struct pidInfo {
    int pid;
    int ppid;
    int pgid;
    int jobid;
};
struct reserveItem {
    char    *resName;
    int     nHost;
    float   *value;
    int     shared;
};
```

PARAMETERS

***more**	Number of job records in the master batch daemon.
***jInfoHExt**	Job information header info for the condensed host group.

RETURN VALUES

jobInfoEnt Function was successful.

The fields in the `jobInfoEnt` structure have the following meaning:

jobId

The job ID that the LSF system assigned to the job.

user

The name of the user who submitted the job.

status

The current status of the job. Possible values are:

JOB_STAT_PEND

The job is pending, i.e., it has not been dispatched yet.

JOB_STAT_PSUSP

The pending job was suspended by its owner or the LSF system administrator.

JOB_STAT_RUN

The job is running.

JOB_STAT_SSUSP

The running job was suspended by the system because an execution host was overloaded or the queue run window closed. (See lsb_queueinfo(), lsb_hostinfo(), and `lsb.queues`.)

JOB_STAT_USUSP: The running job was suspended by its owner or the LSF system administrator.

JOB_STAT_EXIT: The job has terminated with a non-zero status – it may have been aborted due to an error in its execution, or killed by its owner or by the LSF system administrator.

JOB_STAT_DONE: The job has terminated with status 0.

JOB_STAT_UNKWN: The slave batch daemon (`sbatchd`) on the host on which the job is processed has lost contact with the master batch daemon (`mbatchd`).

reasonTb

Pending or suspending reasons of the job.

numReasons

Length of reasonTb vector.

reasons

The reason a job is pending or suspended.

If status is JOB_STAT_PEND, the values of reasons and subreasons are explained by `lsb_pendreason()`. If status is JOB_STAT_PSUSP, the values of reasons and subreasons are explained by `lsb_suspreason()`.

When reasons is PEND_HOST_LOAD or SUSP_LOAD_REASON, subreasons indicates the load indices that are out of bounds. If reasons is PEND_HOST_LOAD, subreasons is the same as busySched in the hostInfoEnt structure; if reasons is SUSP_LOAD_REASON, subreasons is the same as busyStop in the hostInfoEnt structure. (See `lsb_hostinfo()`.)

submitTime

The time the job was submitted, in seconds since 00:00:00 GMT, Jan. 1, 1970.

reserveTime

Time when job slots are reserved.

startTime

The time that the job started running, if it has been dispatched.

PredictedStartTime

Job's predicted start time.

endTime

The termination time of the job, if it has completed.

LastEvent

Last time event.

nextEvent

Next time event.

duration

Duration time (in minutes).

cpuTime

The CPU time that the job has used.

umask

The file creation mask when the job was submitted.

cwd

The current working directory when the job was submitted.

subHomeDir

Home directory on submission host.

fromHost

The name of the host from which the job was submitted.

exHosts

The array of names of hosts on which the job executes.

numExHosts

The number of hosts on which the job executes.

cpuFactor

The CPU factor for normalizing CPU and wall clock time limits.

nIdx

The number of load indices in the `loadSched` and `loadStop` arrays.

loadSched & loadStop

The `loadSched` and `loadStop` arrays are assigned to the job according to those of the queue and hosts to control job suspension and resumption.

The values in the `loadSched` array specify the thresholds for the corresponding load indices. Only if the current values of all specified load indices of a host are within (below or above, depending on the meaning of the load index) their corresponding thresholds may the suspended job be resumed on this host.

Similarly, the values in the `loadStop` array specify the thresholds for job suspension; if any of the current load index values of the host crosses its threshold, the job will be suspended.

For an explanation of the entries in the `loadSched` and `loadStop` arrays, see `lsb_hostinfo()`.

submit

Structure for `lsb_submit()` call.

exitStatus

Job exit status.

execUid

Mapped UNIX user ID on the execution host.

execHome

Home directory for the job on the execution host.

execCwd

Current working directory for the job on the execution host.

execUsername

Mapped user name on the execution host.

jRusageUpdateTime

Time of the last job resource usage update.

jRusage

Contains resource usage information for the job.

jType

Job type.

parentGroup

The parent job group of a job or job group.

jName

if jType is JGRP_NODE_GROUP, then it is the job group name. Otherwise, it is the job name.

counter[NUM_JGRP_COUNTERS]

Index into the counter array. Only used for job arrays:

- JGRP_COUNT_NJOBS—total jobs in the array
- JGRP_COUNT_PEND—number of pending jobs in the array
- JGRP_COUNT_NPSUSP—number of held jobs in the array
- JGRP_COUNT_NRUN—number of running jobs in the array
- JGRP_COUNT_NSSUSP—number of jobs suspended by the system in the array
- JGRP_COUNT_NUSUSP—number of jobs suspended by the user in the array
- JGRP_COUNT_NEXIT—number of exited jobs in the array
- JGRP_COUNT_NDONE—number of successfully completed jobs
- JGRP_COUNT_NJOBS_SLOTS—total slots in the array
- JGRP_COUNT_PEND_SLOTS—number of pending slots in the array
- JGRP_COUNT_RUN_SLOTS—number of running slots in the array
- JGRP_COUNT_SSUSP_SLOTS—number of slots suspended by the system in the array
- JGRP_COUNT_USUSP_SLOTS— number of slots suspended by the user in the array
- JGRP_COUNT_RESV_SLOTS—number of reserverd slots in the array

port

Service port of the job.

jobPriority

Job dynamic priority.

numExternalMsg

The number of external messages in the job.

jobExternalMsgReply

This structure contains the information required to define an external message reply.

clusterId

MultiCluster cluster ID. If clusterId is greater than or equal to 0, the job is a pending remote job, and `lsb_readjobinfo` checks for `host_name@cluster_name`. If host name is needed, it should be found in `jInfoH->remoteHosts`. If the remote host name is not available, the constant string `remoteHost` is used.

detailReason

Detailed reason field.

idleFactor

Idle factor for job exception handling. If the job idle factor is less than the specified threshold, LSF invokes `LSF_SERVERDIR/eadmin` to trigger the action for a job idle exception.

exceptMask

Job exception handling mask.

additionalInfo

Placement information of LSF HPC jobs.

exitInfo

Job termination reason. See `lsbatch.h`.

warningTimePeriod

Job warning time period in seconds; -1 if unspecified.

warningAction

Job warning action, SIGNAL | CHKPNT | command; NULL if unspecified.

chargedSAAP

SAAP charged for job.

execRusage

The rusage satisfied at job runtime.

rsvInActive

The time when advance reservation expired or was deleted.

numLicense

The number of licenses reported from License Scheduler.

licenseNames

License Scheduler license names.

aps

Absolute priority scheduling (APS) priority value.

adminAps

Absolute priority scheduling (APS) string set by administrators to denote static system APS value

adminFactorVal

Absolute priority scheduling (APS) string set by administrators to denote ADMIN factor APS value.

runTime

The real runtime on the execution host.

runTime

The real runtime on the execution host.

reserveCnt

How many kinds of resource are reserved by this job

reserveItem

The `reserveItem` structure contains the following fields:

resname: Name of the resource to reserve.

items : Details reservation information for each kind of resource.

value: Amount of reservation is made on each host. Some hosts may reserve 0.

nhost: The number of hosts to reserve this resource.

shared: Flag for shared or host-base resource

resizeMin: Pending resize min. 0, if no resize pending.

resizeMax: Pending resize max. 0, if no resize pending.

resizeReqTime: Time when pending request was issued.

jStartNumExHosts: Number of hosts when job starts.

jStartExHosts: Host list when job starts.

lastResizeTime: Last time when job allocation changed.

The fields in the submit structure:

submit `submit` uses the submit structure provided by the invoker of `lsb_submit()`.

See lsb_submit() on page 377 for descriptions of the `submit` structure fields.

The fields in the `runRusage` structure have the following meaning:

runRusage `runRusage` uses the `jRusage` structure to provide the total resident memory usage in KB, total virtual memory usage in KB, cumulative total CPU time in seconds and a list of currently active process group IDs and process IDs in a job.

The `jRusage` structure contains the following fields:

mem

Total resident memory usage in KB of all currently running processes in given process groups.

swap

Total virtual memory usage in KB of all currently running processes in given proces groups.

utime

Cumulative total user time in seconds.

stime

Cumulative total system time in seconds.

npids

Number of currently active processesin given process groups.

npgids

Number of currently active process groups.

pgid

Array of currently active process group ids.

nthreads

Number of currently active threads in given process groups.

The fields in the pidInfo structure have the following meaning:

pidInfo

Structure containing information about an active process.

pid

Process id.

ppid

Parent's process id.

pgid

Process group id.

jobid

Process Cray job id (only on Cray).

ERRORS

If there are no more records, then `lsberrno` is set to LSBE_EOF.

SEE ALSO

Related API

`lsb_openjobinfo()` - Opens a job information connection to `mbatchd`

`lsb_openjobinfo_a()` - Provides the name and number of jobs and hosts in `mbatchd`

`lsb_closejobinfo()` - Closes job information connection with `mbatchd`

`lsb_hostinfo()` - Returns informaton about job server hosts

`lsb_pendreason()` - Explains why a job is pending

`lsb_queueinfo()` - Returns information about batch queues

`lsb_readjobinfo()` - Returns the next job information record in `mbatchd`

`lsb_suspreason()` - Explains why a job was suspended

Equivalent line command

none

Files

`lsb.queues`

lsb_readjobmsg()

Reads messages and data posted to a job.

DESCRIPTION

Use `lsb_readjobmsg()` to open a TCP connection, receive attached messages and data from the mbatchd, and display the messages posted by `lsb_postjobmsg()`.

By default, `lsb_readjobmsg()` displays the message “no description” or the message at index position 0 of the specified job. To read other messages, choose another index position. The index is populated by `lsb_postjobmsg()`.

If a data file is attached to a message and the flag `EXT_ATTA_READ` is set, `lsb_readjobmsg()` gets the message and copies its data file to the default directory `JOB_ATTA_DIR`, overwriting the specified file if it already exists. If there is no file attached, the system reports an error.

Users can only read messages and data from their own jobs. Root and LSF administrators can also read messages of jobs submtted by other users, but they cannot read data files attached to jobs owned by other users.

You can read messages and data from a job until it is cleaned from the system. You cannot read messages and data from done or exited jobs.

SYNOPSIS

```
#include <lsf/lsbatch.h>
#include <time.h>
int lsb_readjobmsg(struct jobExternalMsgReq *jobExternalMsg,
    struct jobExternalMsgReply *jobExternalMsgReply)
struct jobExternalMsgReq {
    int options;
    LS_LONG_INT jobId;
    char *jobName;
    int msgIdx;
    char *desc;
    int userId;
    long dataSize;
    time_t postTime;
    char *userName;
};
struct jobExternalMsgReply {
    LS_LONG_INT jobId;
    int msgIdx;
    char *desc;
    int userId;
    long dataSize;
    time_t postTime;
    int dataStatus;
    char *jobName;
};
```

PARAMETERS

jobExternalMsg	This structure contains the information required to define an external message of a job.
options	Specifies if the message has an attachment to be read. `<lsf/lsbatch.h>` defines the following flags constructed from bits. These flags correspond to options. **EXT_MSG_READ** Read the external job message. There is no attached data file. **EXT_ATTA_READ** Read the external job message and data file posted to the job. If there is no data file attached, the error message "The attached data of the message is not available" is displayed, and the external job message is displayed.
jobId	The system generated job Id of the job.
msgIdx	The message index. A job can have more than one message. Use `msgIdx` in an array to index messages.
desc	Text description of the message
userId	The userId of the author of the message.
dataSize	The size of the data file. If no data file is attached, the size is 0.
postTime	The time the author posted the message.
userName	The author of the message.
jobExternalMsgReply	This structure contains the information required to define an external message reply.
jobId	The system generated job Id of the job associated with the message.
msgIdx	The message index. A job can have more than one message. Use `msgIdx` in an array to index messages.
desc	The message you want to read.
userId	The user Id of the author of the message.
dataSize	The size of the data file attached. If no data file is attached, the size is 0.
postTime	The time the message was posted.
dataStatus	The status of the attached data file. The status of the data file can be one of the following: **EXT_DATA_UNKNOWN** Transferring the message's data file. **EXT_DATA_NOEXIST** The message does not have an attached data file. **EXT_DATA_AVAIL** The message's data file is available.

EXT_DATA_UNAVAIL

The message's data file is corrupt.

RETURN VALUES

integer:value	The successful function returns a socket number.
integer:0	The `EXT_ATTA_READ` bit of options is not set or there is no attached data.
integer:-1	The function failed.

ERRORS

If the function fails, `lserrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_postjobmsg()` - Sends messages and attaches data files to a jobs

Equivalent line command

`bread`

Files

`lsb.params`

`JOB_ATTA_DIR`

`LSB_SHAREDIR/info/`

lsb_readframejob()

Returns all frame jobs information which matchs the specified parameters and fills related information into the frame job information table.

DESCRIPTION

`lsb_readframejob()` gets all frame jobs information that matches the specified parameters and fills related information into the frame job information table. lsb_readframejob is a wrapper of `lsb_openjobinfo()`, `lsb_readjobinfo()`, and `lsb_closejobinfo()`. Memory allocated in `frameJobInfoTbl` will be freed by user.

The fields in the `frameJobInfo` structure have the following meaning:

jobGid The job ID that the LSF system assigned to the frame job array.

maxJob The max job number in one frame job array.

userName The user submitted the frame job array.

jobName The full job name of the frame job array. `frameElementPtr` The pointer to frame job array table.

The fields in the `frameElementInfo` structure have the following meaning:

jobindex The job index in the frame job array.

jobState The job status.

start The start frame of this frame job.

end The end frame of this frame job.

step The step of this frame job.

chunk The chunk size of this frame job.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_readframejob(LS_LONG_INT jobId, char *frameName,
    char *user, char *queue, char *host, int options,
    struct frameJobInfo **frameJobInfoTbl)

struct frameJobInfo {
/* jobid of the job array */
    int jobGid;
/* job number in a job array */
    int maxJob;
/* user name */
    char userName[MAX_LSB_NAME_LEN];
/* full job name */
    char jobName[MAXLINELEN];
/* pointer to job array table */
    struct frameElementInfo *frameElementPtr;
};

struct frameElementInfo {
/* job index in a job array */
```

```
    int jobindex;
/* job status */
    int jobState;
/* start frame */
    int start;
/* end frame */
    int end;
/* step size */
    int step;
/* chunk size */
    int chunk;
};
```

PARAMETERS

jobId	Get information about the frame jobs with the given job ID. If jobID is 0, get information about frame jobs which satisfy the other specifications. If a job in a job array is to be modified, use the array form `jobID[i]` where `jobID` is the job array name, and `i` is the index value.
***frameName**	Get information about frame jobs with the given frame name.
***user**	Get information about frame jobs submitted by the named user or user group, or by all users if `user` is `all`. If `user` is `NULL`, the user invoking this routine is assumed.
***queue**	Get information about frame jobs belonging to the named queue. If `queue` is `NULL`, jobs in all queues of the batch system will be considered.
***host**	Get information about frame jobs on the named host, host group or cluster name. If `host` is `NULL`, jobs on all hosts of the batch system will be considered.
options	`<lsf/lsbatch.h>` defines the following flags constructed from bits. Use the bitwise OR to set more than one flag.
****frameJobInfoTbl**	The result of all frame jobs information.

RETURN VALUES

integer:Length of frame job information table

Function was successful.

integer:-1

Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`lsb_openjobinfo()`

`lsb_readjobinfo()`

`lsb_closejobinfo()`

Equivalent line command

none

Files

none

lsb_readstream()

Reads a current version eventRec structure from the lsb_stream file.

DESCRIPTION

lsb_readstream() reads an eventrRec from the open streamFile

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct EventRec lsb_readstream(int *nline)
```

See lsb_geteventrec() for details on the eventRec structure.

PARAMETERS

*** nline** Line number in the stream file to be read.

See lsb_geteventrec() for details on the eventRec structure.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

lsb_closestream(): Close the stream file.

lsb_geteventrec(): Get eventRec structure from an event log file.

lsb_openstream(): Open the stream file.

lsb_puteventrec(): Puts information of an eventRec structure into a log file.

lsb_readstreamline(): Read a line from the stream file.

lsb_streamversion(): Version of the current event type supported by mbatchd.

lsb_writestream(): Write an event to the stream file.

Equivalent line command

None

Files

`lsb.params`

lsb_readstreamline()

Reads a current version eventRec structure from the lsb_stream file.

DESCRIPTION

lsb_readstreamline() reads an eventrRec from the open streamFile

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct EventRec lsb_readstreamline(const char *line)
```

See lsb_geteventrec() and lsb_puteventrec() for details on the eventRec structure.

PARAMETERS

*** line** Line number in the stream file to be read.

See lsb_puteventrec() and lsb_geteventrec() for details on the eventRec structure. Additionally, there are three additional event types supported.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

lsb_closestream(): Close the stream file.

lsb_geteventrec(): Get eventRec structure from an event log file.

lsb_openstream(): Open the stream file.

lsb_puteventrec(): Puts information of an eventRec structure into a log file.

lsb_readstream(): Read from the stream file.

lsb_streamversion(): Version of the current event type supported by mbatchd.

lsb_writestream(): Write an event to the stream file.

Equivalent line command

None

Files

`lsb.params`

lsb_reconfig()

Dynamically reconfigures an LSF batch system.

DESCRIPTION

`lsb_reconfig()` dynamically reconfigures an LSF batch system to pick up new configuration parameters and changes to the job queue setup since system startup or the last reconfiguration (see `lsb.queues`).

To restart a slave batch daemon, use `lsb_hostcontrol()`. This call is successfully invoked only by root or by the LSF administrator.

Any program using this API must be setuid to root if LSF_AUTH is not defined in the lsf.conf file.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_reconfig(void)
```

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`lsb_openjobinfo()`

Equivalent line command

`badmin reconfig`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_removereservation()

Removes a reservation.

DESCRIPTION

Use `lsb_removereservation()` to remove a reservation. `mbatchd` removes the reservation with the specified reservation ID.

SYNOPSIS

```
#include <lsf/lsf.h>
int lsb_removereservation(char *rsvId)
struct rmRsvRequest {
    char    *rsvId;
};
```

PARAMETERS

***rsvId**	Reservation ID of the reservation that you wish to remove.

RETURN VALUES

integer:0	The reservation was removed successfully.
integer:-1	The reservation removal failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_addreservation()` - Makes a reservation

`lsb_modreservation()` - Modifies a reservation

`lsb_reservationinfo()` - Retrieves reservation information

Equivalent line command

`brsvdel`

Files:

none

lsb_requeuejob()

Requeues job arrays, jobs in job arrays, and individual jobs.

DESCRIPTION

Use `lsb_requeuejob()`to requeue job arrays, jobs in job arrays, and individual jobs that are running, pending, done, or exited. In a job array, you can requeue all the jobs or requeue individual jobs of the array.

`lsb_requeuejob()`requeues jobs as if the jobs were in an array. A job not in an array is considered to be a job array composed of one job.

Jobs in a job array can be requeued independently of each other regardless of any job's status (running, pending, exited, done). A requeued job is requeued to the same queue it was originally submitted from or switched to. The job submission time does not change so a requeued job is placed at the top of the queue. Use `lsb_movejob()` to place a job at the bottom or any other position in a queue.

If a clean period is reached before `lsb_requeuejob()` is called, the cleaned jobs cannot be requeued. Set the variable CLEAN_PERIOD in your lsb.params file to determine the amount of time that job records are kept in MBD core memory after jobs have finished or terminated.

To requeue a job assign values to the data members of the jobrequeue data structure, process command line options in case the user has specified a different job, and call `lsb_requeue()` to requeue the job array.

Assign values to the jobID, status, and options data members of the jobrequeue data structure. Assign the job identification number to jobID. Assign JOB_STAT_PEND or JOB_STAT_PSUSP to status. Assign REQUEUE_DONE, REQUEUE_EXIT, and or REQUEUE_RUN to requeue running jobs.

SYNOPSIS

```
#include <lsf/lsbatch.h>
```

```
int lsb_requeuejob(struct jobrequeue *)
```

```
struct jobrequeue {
    LS_LONG_INT jobId;
    int status;
    int options;
};
```

PARAMETERS

jobrequeue	This structure contains the information required to requeue a job.
jobId	Specifies the jobid of a single job or an array of jobs.
status	Specifies the lsbatch status of the requeued job after it has been requeued. The job status can be JOB_STAT_PEND or JOB_STATE_PSUSP. The default status is JOB_STAT_PEND.
options	Specifies the array elements to be requeued.

`<lsf/lsbatch.h>` defines the following flags constructed from bits. These flags correspond to the following options:

REQUEUE_DONE

Requeues jobs that have finished running. Jobs that have exited are not re-run. Equivalent to `brequeue -d` command line option.

REQUEUE_EXIT

Requeues jobs that have exited. Finished jobs are not re-run. Equivalent to `brequeue -e` command line option.

REQUEUE_RUN

Requeues running jobs and puts them in PEND state. Equivalent to `brequeue -r` command line option.

RETURN VALUES

integer:0	The function is successful.
integer:-1	The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_movejob()` - Changes the position of a pending job in a queue

`lsb_pendreason()` - Explains why a job is pending

Equivalent line command

```
brequeue -d
brequeue -e
brequeue -a
brequeue -r
brequeue -H
```

Files

`lsb.params`

`LSB_SHAREDIR`

lsb_reservationinfo()

Retrieve reservation information to display active advance reservations.

DESCRIPTION

Use `lsb_reservationinfo()` to retrieve reservation information from `mbatchd`. This function allocates memory that the caller should free. If the `lsb_reservationinfo()` function succeeds, it returns the reservation records pertaining to a particular reservation ID (`rsvId`) as an array of `rsvInfoEnt` structs.

If `rsvId` is NULL, all reservation information will be returned. If a particular `rsvId` is specified:

- If found, the reservation record pertaining to a particular `rsvId` is returned
- If not found, the number of reservation records is set to zero and the lsberrno is set appropiately

SYNOPSIS

```
#include <lsf/lsf.h>
struct rsvInfoEnt *lsb_reservationinfo(char *rsvId, int *numEnts,
                   int options)

struct rsvInfoEnt {
    int options;
    char *rsvId;
    char *name;
    int numRsvHosts;
    struct hostRsvInfoEnt *rsvHosts;
    char *timeWindow;
    int numRsvJobs;
    LS_LONG_INT *jobIds;
    int *jobStatus;
    char *desc;
    char **disabledDurations;
    int state;
    char *nextInstance;
    char *creator
};

struct hostRsvInfoEnt {
    char *host;
    int numCPUs;
    int numSlots;
    int numRsvProcs;
    int numUsedProcs;
};
```

PARAMETERS

***rsvId**	Reservation ID of the requested reservation.
***numEnts**	Number of reservation entries that `mbatchd` returns.
options	The parameter `options` is currently ignored.

RsvInfoEnt structure

options	Reservation options.
***rsvId**	Reservation ID returned from mbatchd. If the reservation fails, this is NULL. The memory for `rsvid` is allocated by the caller.
name	LSF user group name for the reservation. See the `-g` option of `brsvadd`.
numRsvHosts	Number of hosts reserved.
timeWindow	Active time window for a recurring reservation. See the `-t` option of `brsvadd`.
numRsvJobs	Number of jobs running in the reservation.
***jobIds**	Job IDs of jobs running in the reservation.
***jobStatus**	Status of jobs running in the reservation.
desc	description for the reservation to be created. The description must be provided as a double quoted text string. The maximum length is 512 characters. Equivalent to the value of `brsvadd -d`.
****disabledDurations**	Null-terminated list of disabled durations.
state	The current state of the reservation - active or inactive.
***nextInstance**	The time of the next instance of a recurring reservation.
***creator**	Creator of the reservation.

hostRsvInfoEnt structure

host	Host name.
numCPUs	Number of CPUs reserved on the host.
numSlots	Number of job slots reserved on the host.
numRsvProcs	Number of processors reserved on the host.
numUsedProcs	Number of processors in use on the host.

RETURN VALUES

array:rsvInfoEnt	The information retrieval is successful.
struct:NULL	The information retrieval failed.

ERRORS

On failure, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_addreservation()` - Makes a reservation

`lsb_modreservation()` - Modifies a reservation

`lsb_removereservation()` - Removes a reservation

Equivalent line command

`brsvs`

Files:

none

lsb_resize_cancel()

Cancels a pending job resize allocation request.

DESCRIPTION

Use `lsb_resize_cancel()` to cancel a pending allocation request for a resizable job. A running job can only have one pending request at any particular time. If one request is still pending, additional requests are rejected with a proper error code.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_resize_cancel(LS_LONG_INT jobId);
```

jobId	LSF job ID

RETURN VALUES

On success, returns zero.

On failure, returns -1.

ERRORS

`lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_resize_release()` - Releases part of the allocation of a running resizable job

Equivalent line command

`bresize cancel` *job_ID*

Files

none

lsb_resize_release()

Releases part of the allocation of a running resizable job.

DESCRIPTION

Use `lsb_resize_release()` to release part of a running job allocation.

A running job can only have one pending request at any particular time. If one request is still pending, additional requests are rejected with a proper error code.

If a notification command is defined through job submission, application profile, or the `lsb_resize_release()` API, the notification command is invoked on the first execution host of the job allocation once allocation resize requests have been satisfied.

SYNOPSIS

```
#include <lsf/lsbatch.h>
#define LSB_RESIZE_REL_NONE         0x0
#define LSB_RESIZE_REL_ALL          0x01
#define LSB_RESIZE_REL_CANCEL       0x02
#define LSB_RESIZE_REL_NO_NOTIFY    0x04
lsb_resize_release(struct job_resize_release *req);
struct job_resize_release {
     LS_LONG_INT jobId;
     int         options;
     int         nHosts;
     char        **hosts;
     int         *slots;
     char        *notifyCmd;
};
```

PARAMETERS

The `job_resize_release` struct contains the following fields:

options `options` is constructed from the bitwise inclusive OR of zero or more of the following flags, as defined in `lsbatch.h`:

- LSB_RESIZE_REL_ALL means release all slots—In this case, `nHosts`, `hosts` and `slots` indicate the slots that are not released
- LSB_RESIZE_REL_CANCEL means cancel any pending resize request
- LSB_RESIZE_REL_NO_NOTIFY means execute no resize notification command
- LSB_RESIZE_REL_NONE means release no slots

nHosts number of hosts in the hosts list, if no hosts are to be specified this should be zero

hosts specified hosts list, `nHosts` number of elements

slots	slots list, each element specifies the number of slots per corresponding host (0 implies all), `nHosts` number of elements
notifyCmd	name and location of notification command
jobId	LSF job ID

RETURN VALUES

On success, returns zero.

On failure, returns -1.

ERRORS

`lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_resize_cancel()` - Cancels a pending job resize allocation request

Equivalent line command

`release [-c] [-rnc` *resize_notification_cmd* | `-rncn]` *released_host_specification job_ID*

Files

none

lsb_runjob()

Starts a batch job immediately on a set of specified `host()`.

DESCRIPTION

`lsb_runjob()` starts a batch job immediately on a set of specified `host()`. The job must have been submitted and is in `PEND` or `FINISHED` status. Only the LSF administrator or the owner of the job can start the job. If the options is set to `RUNJOB_OPT_NOSTOP`, then the job will not be suspended by the queue's `RUNWINDOW`, `loadStop` and `STOP_COND` and the hosts' `RUNWINDOW` and `loadStop` conditions. By default, these conditions apply to the job as do to other normal jobs.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the lsf.conf file.

SYNOPSIS

```
truct runJobRequest {
    LS_LONG_INT jobId;    /* jobid of the requested job */
    int     numHosts;     /* The number of hosts */
    char    **hostname;   /* Vector of hostnames */
#define RUNJOB_OPT_NORMAL     0x01
#define RUNJOB_OPT_NOSTOP     0x02
#define RUNJOB_OPT_PENDONLY   0x04 /* pending jobs only, no finished jobs */
#define RUNJOB_OPT_FROM_BEGIN 0x08 /* chkpnt job only, from beginning */
#define RUNJOB_OPT_FREE       0x10 /* brun to use free CPUs only */
#define RUNJOB_OPT_IGNORE_RUSAGE  0x20 /* brun ignoring rusage */
    int     options;      /* Run job request options */
 int     *slots;       /* Vector of number of slots per host */
};
```

PARAMETERS

***runReq**	The job-starting request.

RETURN VALUES

integer:0	Function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

none

SEE ALSO

Equivalent line command

brun

Files

`lsf.conf`

lsb_sharedresourceinfo()

Returns the requested shared resource information in dynamic values.

DESCRIPTION

`lsb_sharedresourceinfo()` returns the requested shared resource information in dynamic values. The result of this call is a chained data structure as defined in `<lsf/lsbatch.h>`, which contains requested information.

SYNOPSIS

```
#include <lsf/lsbatch.h>
LSB_SHARED_RESOURCE_INFO_T *lsb_sharedresourceinfo(
                            char **resources,
                            int *numResources,
                            char *hostName, int options)

typedef struct lsbSharedResourceInfo {
    char *resourceName;
    int nInstances;
    LSB_SHARED_RESOURCE_INST_T *instances;
} LSB_SHARED_RESOURCE_INFO_T;

typedef struct lsbSharedResourceInstance {
    char *totalValue;
    char *rsvValue;
    int nHosts;
    char **hostList;
} LSB_SHARED_RESOURCE_INST_T;
```

PARAMETERS

****resources**	`resources` is an `NULL` terminated string array storing requesting resource names. Setting `resources` to point to `NULL` returns all shared resources.
***numResources**	`numResources` is an input/output parameter. On input it indicates how many resources are requested. Value 0 means requesting all shared resources. On return it contains qualified resource number.
***hostName**	`hostName` is a string containing a host name. Only shared resource available on the specified host will be returned. If `hostName` is a `NULL`, shared resource available on all hosts will be returned.
options	`options` is reserved for future use. Currently, it should be set to 0.

RETURN VALUES

pointer:	On success, `lsb_sharedresourceinfo()` returns a pointer to an `LSB_SHARED_RESOURCE_INFO_T` structure, which contains complete shared resource information.
char:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`ls_sharedresourceinfo()`

Equivalent line command

none

Files

`$LSF_CONFDIR/lsf.shared`

`$LSF_CONFDIR/lsf.cluster.`*cluster_name*

lsb_signaljob()

Sends a signal to a job.

DESCRIPTION

Use `lsb_signaljob()` when migrating a job from one host to another. Use `lsb_signaljob()` to stop or kill a job on a host before using `lsb_mig()` to migrate the job. Next, use `lsb_signaljob()` to continue the stopped job at the specified host.

Generally, use `lsb_signaljob()` to apply any UNIX signal to a job or process.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_signaljob (LS_LONG_INT jobId, int sigValue)
```

PARAMETERS

jobId The job to be signaled. If a job in a job array is to be signaled, use the array form jobID[i] where jobID is the job array name, and `i` is the index value.

sigValue `SIGSTOP`, `SIGCONT`, `SIGKILL` or some other UNIX signal.

RETURN VALUES

integer:0 The function was successful.

integer:-1 The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

`lsb_chkpntjob()` - checkpoints a job

`lsb_forcekilljob()` - kills a job

`lsb_mig()` - migrates a job to another host

Equivalent line command

`bkill` - sends a signal to kill, suspend, or resume unfinished jobs

`bstop` - suspends unfinished jobs

`bresume` - resumes a suspended job

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_streamversion()

Version of the current event type supported by mbatchd.

DESCRIPTION

lsb_streamversion() returns the event version number of mbatchd, which is the version of the events to be written to the stream file.

This API function is inside `liblsbstream.so`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
```

```
char * lsb_streamversion(void)
```

PARAMETERS

none

RETURN VALUES

char* Pointer to a string of the current event version in mbatchd, also known as THIS_VERSION.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

lsb_closestream(): Close the stream file.

lsb_geteventrec(): Get eventRec structure from an event log file.

lsb_openstream(): Open the stream file.

lsb_puteventrec(): Puts information of an eventRec structure into a log file.

lsb_readstreamline(): Read a line from the stream file.

lsb_writestream(): Write an event to the stream file.

lsb_readstream(): Read from the stream file.

Equivalent line command

None

Files

`lsb.params`

lsb_submit()

Submits or restarts a job in the batch system.

DESCRIPTION

`lsb_submit()` submits or restarts a job in the batch system according to the jobSubReq specification.

SYNOPSIS

```
#include <lsf/lsbatch.h>
LS_LONG_INT lsb_submit (struct submit *jobSubReq,
                         struct submitReply *jobSubReply)
struct submit {
    int     options;
    int     options2;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    int     numProcessors;
    char    *dependCond;
    char    *timeEvent;
    time_t  beginTime;
    time_t  termTime;
    int     sigValue;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *newCommand;
    time_t  chkpntPeriod;
    char    *chkpntDir;
    int     nxf;
    struct xFile *xf;
    char    *preExecCmd;
    char    *mailUser;
    int     delOptions;
    int     delOptions2;
    char    *projectName;
    int     maxNumProcessors;
    char    *loginShell;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *jobGroup;
    char    *sla;
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    int     jsdlFlag;
    char    *jsdlDoc;
    void    *correlator;
    char    *apsString;
    char    *postExecCmd;
    char    *cwd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
    char    *notifyCmd;
};
struct submitReply {
    char *queue;
    LS_LONG_INT badJobId;
    char *badJobName;
    int badReqIndx;
};
```

PARAMETERS

jobSubReq	Describes the requirements for job submission to the batch system. A job that does not meet these requirements is not submitted to the batch system and an error is returned.
jobSubReply	Describes the results of the job submission to the batch system.

structure jobSubReq

The `submit` structure contains the following fields:

options `<lsf/lsbatch.h>` defines the following flags constructed from bits. These flags correspond to some of the options of the `bsub` command line. Use the bitwise OR to set more than one flag.

SUB_JOB_NAME

Flag to indicate `jobName` parameter has data. Equivalent to `bsub -J` command line option existence.

SUB_QUEUE

Flag to indicate `queue` parameter has data. Equivalent to `bsub -q` command line option existence.

SUB_HOST

Flat to indicate `numAskedHosts` parameter has data. Equivalent to `bsub -m` command line option existence.

SUB_IN_FILE

Flag to indicate `inFile` parameter has data. Equivalent to `bsub -i` command line option existence.

SUB_OUT_FILE

Flag to indicate `outFile` parameter has data. Equivalent to `bsub -o` command line option existence.

SUB_ERR_FILE

Flag to indicate `errFile` parameter has data. Equivalent to `bsub -e` command line option existence.

SUB_EXCLUSIVE

Flag to indicate execution of a job on a host by itself requested. Equivalent to `bsub -x` command line option existence.

SUB_NOTIFY_END

Flag to indicate whether to send mail to the user when the job finishes. Equivalent to `bsub -N` command line option existence.

SUB_NOTIFY_BEGIN

Flag to indicate whether to send mail to the user when the job is dispatched. Equivalent to bsub -B command line option existence.

SUB_USER_GROUP

Flag to indicate userGroup name parameter has data. Equivalent to `bsub -G` command line option existence.

SUB_CHKPNT_PERIOD

Flag to indicatechkpntPeriod parameter has data . Equivalent to `bsub -k` command line option existence.

SUB_CHKPNT_DIR

Flag to indicate chkpntDir parameter has data. Equivalent to `bsub -k` command qwreqwqwline option existence.

SUB_CHKPNTABLE

Indicates the job is checkpointable. Equivalent to `bsub -k` command line option.

SUB_RESTART_FORCE

Flag to indicate whether to forces the job to restart even if non-restartable conditions exist. These conditions are operating system specific. Equivalent to `brestart() -f` command line option existence.

SUB_RESTART

Flag to indicate restart of a checkpointed job. Only jobs that have been successfully checkpointed can be restarted. Jobs are re-submitted and assigned a new job ID. By default, jobs are restarted with the same output file, file transfer specifications, job name, window signal value, checkpoint directory and period, and rerun options as the original job. To restart a job on another host, both hosts must be binary compatible, run the same OS version, have access to the executable, have access to all open files (LSF must locate them with an absolute path name), and have access to the checkpoint directory. Equivalent to bsub -k command line option existence.

SUB_RERUNNABLE

Indicates the job is re-runnable. If the execution host of the job is considered down, the batch system will re-queue this job in the same job queue, and re-run it from the beginning when a suitable host is found. Everything will be as if it were submitted as a new job, and a new job ID will be assigned. The user who submitted the failed job will receive a mail notice of the job failure, requeueing of the job, and the new job ID.

For a job that was checkpointed before the execution host went down, the job will be restarted from the last checkpoint. Equivalent to `bsub -r` command line option existence.

SUB_WINDOW_SIG

Flag to indicate sigValue parameter has data. Sends a signal as the queue window closes.

SUB_HOST_SPEC

Flag to indicate hostSpec parameter has data.

SUB_DEPEND_COND

Flag to indicate dependCond parameter has data. Equivalent to `bsub -w` command line option existence.

SUB_RES_REQ

Flag to indicate resReq parameter has data. Equivalent to `bsub -R` command line option existence.

SUB_OTHER_FILES

Flag to indicate nxf parameter and structure xf have data.

SUB_PRE_EXEC

Flag to indicate preExecCmd parameter has data. Equivalent to `bsub -E` command line option existence.

SUB_LOGIN_SHELL

Flag to indicate loginShell parameter has data.

Equivalent to `bsub -L` command line option existence.

SUB_MAIL_USER

Flag to indicate mailUser parameter has data.

SUB_MODIFY

Flag to indicate newCommand parameter has data. Equivalent to `bmod bsub_options` existence.

SUB_MODIFY_ONCE

Flag to indicate modify option once.

SUB_PROJECT_NAME

Flag to indicate `ProjectName` parameter has data . Equivalent to `bsub -P` command line option existence.

SUB_INTERACTIVE

Indicates that the job is submitted as a batch interactive job. When this flag is given, `lsb_submit()` does not return unless an error occurs during the submission process. When the job is started, the user can interact with the job's standard input and output via the terminal. See the `-I` option in `bsub` for the description of a batch interactive job. Unless the SUB_PTY flag is specified, the job will run without a pseudo-terminal. Equivalent to `bsub -I` command line option.

SUB3_INTERACTIVE_SSH

Protects the sessions of interactive jobs with SSH encryption. Equivalent to `bsub -IS|-ISp|-ISs`.

SUB3_XJOB_SSH

Protect the sessions of interactive x-window job with SSH encryption. Equivalent to `bsub -IX`.

SUB_PTY

Requests pseudo-terminal support for a job submitted with the SUB_INTERACTIVE flag. This flag is ignored if SUB_INTERACTIVE is not specified. A pseudo-terminal is required to run some applications (e.g., `vi`). Equivalent to `bsub -Ip` command line option.

SUB_PTY_SHELL

Requests pseudo-terminal shell mode support for a job submitted with the SUB_INTERACTIVE and SUB_PTY flags. This flag is ignored if SUB_INTERACTIVE and SUB_PTY are not specified. This flag should be specified for submitting interactive shells, or applications which redefine the ctrl-C and ctrl-Z keys (e.g., `jove`). Equivalent to `bsub -Is` command line option.

SUB_EXCEPT

Exception handler for job.

SUB_TIME_EVENT

Specifies time_event.

options2 Extended bitwise inclusive OR of some of the following flags:

SUB2_HOLD

Hold the job after it is submitted. The job will be in PSUSP status. Equivalent to `bsub -H` command line option.

SUB2_MODIFY_CMD

New cmd for `bmod`.

SUB2_BSUB_BLOCK

Submit a job in a synchronous mode so that submission does not return until the job terminates. Note once this flag is set, the `lsb_submit()` will never return if the job is accepted by LSF. Programs that wishes to know the status of the submission needs to fork, with the child process invoking the API call in the blocking mode and the parent process wait on the child process (see `wait()` for details.

SUB2_HOST_NT

Submit from NT.

SUB2_HOST_UX

Submit fom UNIX.

SUB2_QUEUE_CHKPNT

Submit to a chkpntable queue.

SUB2_QUEUE_RERUNNABLE

Submit to a rerunnble queue.

SUB2_IN_FILE_SPOOL

Spool job command.

SUB2_JOB_CMD_SPOOL

Inputs the specified file with spooling.

SUB2_JOB_PRIORTY

Submits job with priority.

SUB2_USE_DEF_PROCLIMIT

Job submitted wihtout `-n`, use queue's default proclimit.

SUB2_MODIFY_RUN_JOB

`bmod -c/-M/-W/-o/-e`

SUB2_MODIFY_PEND_JOB

`bmod` options only to pending jobs.

SUB2_WARNING_TIME_PERIOD

Job action warning time. Equivalent to `bsub` or `bmod -wt`.

SUB2_WARNING_ACTION

Job action to be taken before a job control action occurs. Equivalent to `bsub` or `bmod -wa`.

SUB2_USE_RSV

Use an advance reservation created with the `brsvadd` command. Equivalent to `bsub -U`.

SUB2_TSJOB

Windows Terminal Services job

SUB2_LSF2TP Deprecated

Parameter is deprecated

SUB2_JOB_GROUP

Submit into a job group

SUB2_SLA

Submit into a service class

SUB2_EXTSCHED

Submit with `-extsched` options

SUB2_LICENSE_PROJECT

License Scheduler project

SUB2_OVERWRITE_OUT_FILE

Overwrite the standard output of the job. Equivalent to `bsub -oo`.

SUB2_OVERWRITE_ERR_FILE

Overwrites the standard error output of the job. Equivalent to `bsub -eo`.

SUB3_RUNTIME_ESTIMATION

Use in conjunction with SUB3_RUNTIME_ESTIMATION_ACC and SUB3_RUNTIME_ESTIMATION_PERC.

SUB3_RUNTIME_ESTIMATION_ACC

Runtime estimate that is the accumulated run time plus the runtime estimate. Equivalent to `bmod -We+`. Use in conjunction with SUB3_RUNTIME_ESTIMATION.

SUB3_RUNTIME_ESTIMATION_PERC

Runtime estimate in percentage of completion. Equivalent to `bmod -Wep`. Two digits after the decimal point are suported. The highest eight bits of runtimeEstimation in the submit structure are used for the integer; the remaining bits are used for the fraction. Use in conjunction with SUB3_RUNTIME_ESTIMATION.

jobName	The job name. If `jobName` is `NULL`, `command` is used as the job name.
queue	Submit the job to this queue. If `queue` is `NULL`, submit the job to a system default queue.
numAskedHosts	The number of invoker specified candidate hosts for running the job. If `numAskedHosts` is 0, all qualified hosts will be considered.
askedHosts	The array of names of invoker specified candidate hosts. The number of hosts is given by `numAskedHosts`.
resReq	The resource requirements of the job. If `resReq` is `NULL`, the batch system will try to obtain resource requirements for `command` from the remote task lists (see `ls_task()`). If the task does not appear in the remote task lists, then the default resource requirement is to run on `host()` of the same type.

rLimits[LSF_RLIM_NLIMITS]

Limits on the consumption of system resources by all processes belonging to this job. See `getrlimit()` for details. If an element of the array is `-1`, there is no limit for that resource. For the constants used to index the array, see `lsb_queueinfo()`.

hostSpec	Specify the host model to use for scaling `rLimits[LSF_RLIMIT_CPU]` and `rLimits[LSF_RLIMIT_RUN]`. (See `lsb_queueinfo()`). If `hostSpec` is `NULL`, the local host is assumed.
numProcessors	The initial number of processors needed by a (parallel) job. The default is 1.
timeEvent	Time event string.
dependCond	The job dependency condition.
beginTime	Dispatch the job on or after `beginTime`, where `beginTime` is the number of seconds since 00:00:00 GMT, Jan. 1, 1970 (See `time()`, `ctime()`). If `beginTime` is 0, start the job as soon as possible.
termTime	The job termination deadline. If the job is still running at `termTime`, it will be sent a USR2 signal. If the job does not terminate within 10 minutes after being sent this signal, it will be ed. `termTime` has the same representation as `beginTime`. If `termTime` is 0, allow the job to run until it reaches a resource limit.
sigValue	Applies to jobs submitted to a queue that has a run window (See `lsb_queueinfo()`). Send signal `sigValue` to the job 10 minutes before the run window is going to close. This allows the job to clean up or checkpoint itself, if desired. If the job does not terminate 10 minutes after being sent this signal, it will be suspended.
inFile	The path name of the job's standard input file. If `inFile` is `NULL`, use `/dev/null` as the default.
outFile	The path name of the job's standard output file. If `outFile` is `NULL`, the job's output will be mailed to the submitter.
errFile	The path name of the job's standard error output file. If `errFile` is `NULL`, the standard error output will be merged with the standard output of the job.
command	The command line of the job.
newCommand	New command line for `bmod`.
chkpntPeriod	The job is checkpointable with a period of `chkpntPeriod` seconds. The value 0 disables periodic checkpointing.
chkpntDir	The directory where the `chk` directory for this job checkpoint files will be created. When a job is checkpointed, its checkpoint files are placed in `chkpntDir/chk`. `chkpntDir` can be a relative or absolute path name.
nxf	The number of files to transfer.
xf	The array of file transfer specifications. (The `xFile` structure is defined in `<lsf/lsbatch.h>`.)
preExecCmd	The job pre-execution command.
mailUser	The user that results are mailed to.
delOptions	Delete options in `options` field .
delOptions2	Extended delete options in `options2` field .

projectName	The name of the project the job will be charged to.
maxNumProcessors	Maximum number of processors the required to run the job.
loginShell	Specified login shell used to initialize the execution environment for the job (see the `-L` option of `bsub`).
userGroup	The name of the LSF user group (see `lsb.users`) to which the job will belong. (see the `-G` option of `bsub`)
exceptList	Passes the exception handlers to mbatchd during a job. (see the `-X` option of `bsub`).

Specifies execption handlers that tell the system how to respond to an exceptional condition for a job. An action is performed when any one of the following exceptions is detected:

- missched - A job has not been scheduled within the time event specified in the -T option.
- overrun - A job did not finish in its maximum time (maxtime).
- underrun - A job finished before it reaches its minimum running time (mintime).
- abend - A job terminated abnormally. Test an exit code that is one value, two or more comma separated values, or a range of values (two values separated by a '-' to indivate a range). If the job exits with one of the tested values, the abend condition is detected.
- startfail - A job did not start due to insufficient system resources.
- cantrun - A job did not start because a dependency condition (see the `-w` option of `bsub`) is invalid, or a startfail exception occurs 20 times in a row and the job is suspended. For jobs submitted with a time event (see the -T option of bsub), the cantrun exception condition can be detected once in each time event.
- hostfail - The host running a job becomes unavailable.

When one or more of the above exceptions is detected, you can specify one of the following actions to be taken:

- alarm - Triggers an alarm incident (see balarms(1)). The alarm can be viewed, acknowledged and resolved.
- setexcept - Causes the exception event event_name to be set. Other jobs waiting on the exception event event_name specified through the -w option can be triggered. event_name is an arbitrary string.
- rerun - Causes the job to be rescheduled for execution. Any dependencies associated with the job must be satisfied before re-execution takes place. The rerun action can only be specified for the abend and hostfail exception conditions. The startfail exception condition automatically triggers the rerun action.
- kill - Causes the current execution of the job to be terminated. This action can only be specified for the overrun exception condition.

userPriority	User priority for fairshare scheduling.
rsvId	Reservation ID for advance reservation.
jobGroup	Job group under which the job runs.

PARAMETERS

sla	SLA under which the job runs.
extsched	External scheduler options.
warningTimePeriod	warning time period in seconds. -1 if unspecified.
warningAction	warning action: SIGNAL, or CHKPNT, or command. NULL if unspecified.
licenseProject	License Scheduler project name.
options3	Extended bitwise inclusive OR of options flags:

SUB3_APP

Application profile name. Equivalent to `bsub -app`.

SUB3_APP_RERUNNABLE

Job rerunable because of application profile

SUB3_ABSOLUTE_PRIORITY

Job modified with absolute priority. . Equivalent to `bmod -aps`.

SUB3_DEFAULT_JOBGROUP

Submit into a default job group. Equivalent to `bsub -g`.

SUB3_POST_EXEC

Run the specified post-execution command on the execution host after the job finishes. Equivalent to `bsub -Ep`.

SUB3_USER_SHELL_LIMITS

Pass user shell limits to execution host. Equivalent to `bsub -ul`.

SUB3_CWD

Current working directory specified on on the command line with `bsub -cwd`

SUB3_RUNTIME_ESTIMATION

Runtime estimate. Equivalent to `bsub -We`.

SUB3_NOT_RERUNNABLE

Job is not rerunnable. Equivalent to `bsub -rn`.

SUB3_JOB_REQUEUE

Job level requeue exit values.

SUB3_INIT_CHKPNT_PERIOD

Initial checkpoint period. Equivalent to `bsub -k` *initial_checkpoint_period.*

SUB3_MIG_THRESHOLD

Job migration threshold. Equivalent to `bsub -mig` *migration_threshold*.

SUB3_APP_CHKPNT_DIR

Checkpoint dir was set by application profile.

SUB3_BSUB_CHK_RESREQ

Bsub only checks the reqreq syntax.

SUB3_BSUB_CHK_RESREQ

Value of BSUB_CHK_RESREQ environment variable, used for select section resource requirement string syntax checking with `bsub -R`.

SUB3_AUTO_RESIZE

SUB3_RESIZE_NOTIFY_CMD

delOptions3	Extended delete options in `options3` field.
app	Application profile under which the job runs.
jsdlFlag	-1 if no `-jsdl` and `-jsdl_strict` options: ◆ 0 -jsdl_strict option ◆ 1 -jsdl option
jsdlDoc	JSDL file name.
apsString	Absolute priority scheduling string set by administrators to denote static system APS value or ADMIN factor APS value. This field is ignored by `lsb_submit()`.
postExecCmd	Post-execution commands specified by `-Ep` option of `bsub` and `bmod`.
cwd	Current working directory specified by `-cwd` option of `bsub` and `bmod`.
runtimeEstimation	Runtime estimate specified by `-We` option of `bsub` and `bmod`.
requeueEValues	Job-level requeue exit values specified by `-Q` option of `bsub` and `bmod`.
initChkpntPeriod	Initial checkpoint period specified by `-k` option of `bsub` and `bmod`.
migThreshold	Job migration threshold specified by `-mig` option of `bsub` and `bmod`.
notifyCmd	Job resize notification command to be invoked on the first execution host when a resize request has been satisfied.

submitReply structure

The `submitReply` structure contains the following fields:

queue

The queue the job was submitted to.

badJobId

`dependCond` contained `badJobId` but `badJobId` does not exist in the system.

badJobName

`dependCond` contained `badJobName` but `badJobName` does not exist in the system.

If the environment variable BSUB_CHK_RESREQ is set, the value of `lsberrno` is either LSBE_RESREQ_OK or LSBE_RESREQ_ERR, depending on the result of resource requirement string checking. The `badJobName` field contains the detailed error message.

badReqIndx

If `lsberrno` is LSBE_BAD_HOST, `(**askedHosts)[badReqIndx]` is not a host known to the system. If `lsberrno` is `LSBE_QUEUE_HOST`, `(**askedHosts)[badReqIndx]` is not a host used by the specified queue. If `lsberrno` is LSBE_OVER_LIMIT, `(*rLimits)[badReqIndx]` exceeds the queue's limit for the resource.

RETURN VALUES

character:job ID The function was successful, and sets the `queue` field of `jobSubReply` to the name of the queue that the job was submitted to.

If the environment variable BSUB_CHK_RESREQ is set, `lsb_submit()` returns a jobid less than zero (0).

integer:-1 Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

If the environment variable BSUB_CHK_RESREQ is set, the value of `lsberrno` is either LSBE_RESREQ_OK or LSBE_RESREQ_ERR, depending on the result of resource requirement string checking. The `badJobName` field in the `submitReply` structure contains the detailed error message.

SEE ALSO

Related API

`lsb_modify()` - Modifies a submitted job's parameters

`ls_info()` - Returns a pointer to an `lsInfo` structure

`lsb_queueinfo()` - Returns information about batch queues

Equivalent line command

`bsub`

`brestart`

Files

`${LSF_ENVDIR/etc}/lsf.conf`

lsb_submitframe()

Submits a frame job to the batch system.

DESCRIPTION

`lsb_submitframe()` submits a frame job to the batch system according to the *jobSubReq* specification and *frameExp*.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_submitframe (struct submit *jobSubReq, char *frameExp,
    struct submitReply *jobSubReply)

struct submit {
    int     options;
    int     options2;
    char    *jobName;
    char    *queue;
    int     numAskedHosts;
    char    **askedHosts;
    char    *resReq;
    int     rLimits[LSF_RLIM_NLIMITS];
    char    *hostSpec;
    int     numProcessors;
    char    *dependCond;
    char    *timeEvent;
    time_t  beginTime;
    time_t  termTime;
    int     sigValue;
    char    *inFile;
    char    *outFile;
    char    *errFile;
    char    *command;
    char    *newCommand;
    time_t  chkpntPeriod;
    char    *chkpntDir;
    int     nxf;
    struct xFile *xf;
    char    *preExecCmd;
    char    *mailUser;
    int     delOptions;
    int     delOptions2;
    char    *projectName;
    int     maxNumProcessors;
    char    *loginShell;
    char    *userGroup;
    char    *exceptList;
    int     userPriority;
    char    *rsvId;
    char    *jobGroup;
    char    *sla;
    char    *extsched;
    int     warningTimePeriod;
    char    *warningAction;
    char    *licenseProject;
    int     options3;
    int     delOptions3;
    char    *app;
    int     jsdlFlag;
    char    *jsdlDoc;
    void    *correlator;
    char    *apsString;
    char    *postExecCmd;
    char    *cwd;
    int     runtimeEstimation;
    char    *requeueEValues;
    int     initChkpntPeriod;
    int     migThreshold;
};
struct submitReply {
    char *queue;
    LS_LONG_INT badJobId;
    char *badJobName;
    int badReqIndx;
};
```

PARAMETERS

***jobSubReq** Describes the requirements for job submission to the batch system. A job that does not meet these requirements is not submitted to the batch system and an error is returned.

See lsb_submit() on page 377 for descriptions of the `submit` structure fields.

***frameExp** The syntax of `frameExp` is:

frame_name[indexlist]

`frame_name` is any name consisting of alphanumerics, periods, forward slashes, dashes or underscores. `indexlist` is a list of one or more frame indexes, separated by commas. These indexes can each be either a single integer or a range, specified in the following format:

start-end[xstep[:chunk]]

`start`, `end`, `step`, and `chunk` are integers, but `chunk` must be positive. If step and chunk are ommitted, the default value is 1.

An example of a valid expression for `frameExp` is:

`Frame_job_1[5,10-15,20-30x2:3]`

***jobSubReply**	Describes the results of the job submission to the batch system. See lsb_submit() on page 377 for descriptions of the `submitReply` structure fields.

RETURN VALUES

char:Job ID	The function was successful, and sets the `queue` field of `jobSubReply` to the name of the queue that the job was submitted to.
int:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error and `jobSubReply` gives additional information about the error.

SEE ALSO

Related API

none

Equivalent line command

none

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_suspreason()

Explains why a job was suspended.

DESCRIPTION

Using the SBD, `lsb_suspreason()` explains why system-suspended and user-suspended jobs were suspended.

SYNOPSIS

```
#include <lsf/lsbatch.h>
```

```
char *lsb_suspreason(int reasons, int subreasons,
                     struct loadIndexLog *ld)
```

```
struct loadIndexLog {
    int nIdx;
    char **name;
};
```

PARAMETERS

reasons Reasons a job suspends:

SUSP_HOST_LOCK

The LSF administrator has locked the execution host.

SUSP_LOAD_REASON

A load index exceeds its threshold. The `subreasons` field indicates which indices.

SUSP_MBD_PREEMPT

The job was preempted by mbatchd because of a higher priorty job.

SUSP_QUEUE_WINDOW

The run window of the queue is closed.

SUSP_RESCHED_PREEMPT

Suspended after preemption. The system needs to re-allocate CPU utilization by job priority.

SUSP_SBD_PREEMPT

Preempted by sbatchd. The job limit of the host/user has been reached.

SUSP_USER_RESUME

The job is waiting to be re-scheduled after being resumed by the user.

SUSP_USER_STOP

The user suspended the job.

SUSP_ADMIN_STOP

The job was suspened by root or the LSF administrator.

SUSP_SBD_STARTUP

The job is suspended while the sbatchd is restarting.

SUSP_HOST_LOCK_MASTER

The execution host is locked by the master LIM.

SUSP_QUE_STOP_CONDITION

The suspend conditions of the queue, as specified by the `STOP_COND` parameter in `lsb.queues`, are true.

SUSP_QUE_RESUME_CONDITION

The resume conditions of the queue, as specified by the RESUME_COND parameter in `lsb.queues`, are false.

SUSP_RES_RESERVE

The job is terminated due to resource limit.

SUSP_RES_LIMIT

The job's requirements for resource reservation are not satisfied.

SUSP_PG_IT

The job was suspended due to the paging rate and the host is not idle yet.

SUSP_REASON_RESET

Resets the previous reason.

SUSP_MBD_LOCK

The job is locked by the mbatchd.

SUSP_LOAD_UNAVAIL

Load information on the execution hosts is unavailable.

subreasons If `reasons` is SUSP_LOAD_REASON, `subreasons` indicates the load indices that are out of bounds. The integer values for the load indices are found in `lsf.h`.

If `reasons` is `SUSP_RES_LIMIT`, `subreasons` indicates the job's requirements for resource reservation are not satisfied. The integer values for the job's requirements for resource reservation are found in `lsbatch.h`.

`Subreasons` a job suspends if `reasons` is `SUSP_LOAD_REASON`:

R15S

15 second CPU run queue length

R1M

1 minute CPU run queue length

R15M

15 minute CPU run queue length

UT

1 minute CPU utilization

PG

Paging rate

IO

Disk IO rate

LS

Number of log in sessions

IT

Idle time

TMP

Available temporary space

SWP

Available swap space

MEM

Available memory

USR1

USR1 is used to describe unavailable or out of bounds user defined load information of an external dynamic load indice on execution hosts.

USR2

USR2 is used to describe unavailable or out of bounds user defined load information of an external dynamic load indice on execution hosts.

Subreasons a job suspends if `reasons` is SUSP_RES_LIMIT:

SUB_REASON_RUNLIMIT

The run limit was reached.

SUB_REASON_DEADLINE

The deadline was reached.

SUB_REASON_PROCESSLIMIT

The process limit was reached.

SUB_REASON_CPULIMIT

The CPU limit was reached.

SUB_REASON_MEMLIMIT

The memory limit was reached.

ld When `reasons` is SUSP_LOAD_REASON, ld is used to determine the name of any external load indices. ld uses the most recent load index log in the `lsb.events` file.

The `loadIndexLog` structure contains the following fields:

nIdx

Number of load indices.

names

Names of load indices.

RETURN VALUES

char: reasons	Returns the suspending reason string.
char:NULL	The function failed. The reason code is bad.

ERRORS

No error handling

SEE ALSO

Related API

`lsb_pendreason()` - Explains why a job is pending

Equivalent line command

`bjobs -s`

Environment Variable

`LSB_SUSP_REASONS`

Files

`lsb.queues`

`lsb.events`

lsb_switchjob()

Switches an unfinished job to another queue.

DESCRIPTION

`lsb_switchjob()` switches an unfinished job to another queue. Effectively, the job is removed from its current queue and re-queued in the new queue.

The switch operation can be performed only when the job is acceptable to the new queue. If the switch operation is unsuccessful, the job will stay where it is.

A user can only switch his/her own unfinished jobs, but root and the LSF administrator can switch any unfinished job.

Any program using this API must be setuid to root if `LSF_AUTH` is not defined in the `lsf.conf` file.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_switchjob (LS_LONG_INT jobId, char *queue)
```

PARAMETERS

jobId	The job to be switched. If an element of a job array is to be switched, use the array form `jobID[i]` where `jobID` is the job array name, and `i` is the index value.
***queue**	The new queue for the job.

RETURN VALUES

integer:0	The function was successful.
integer:-1	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

none

Equivalent line command

`bswitch`

Files

`${LSF_ENVDIR-/etc}/lsf.conf`

lsb_sysmsg()

Returns a pointer to static data.

DESCRIPTION

`lsb_sysmsg()` returns a pointer to static data which stores the batch error message corresponding to `lsberrno`. The global variable `lsberrno` maintained by LSBLIB holds the error number from the most recent LSBLIB call that caused an error. If `lsberrno == LSBE_SYS_CALL`, then the system error message defined by `errno` is also returned. If `lsberrno == LSBE_LSLIB`, then the error message returned by `ls_sysmsg()` is returned.

SYNOPSIS

```
#include <lsf/lsbatch.h>
extern int lsberrno;

char *lsb_sysmsg (void)
```

RETURN VALUES

char: errno/ls_sysmsg()

If `lsberrno == LSBE_SYS_CALL`, then the system error message defined by `errno` is also returned. If `lsberrno == LSBE_LSLIB`, then the error message returned by `ls_sysmsg()` is returned.

char:NULL

Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related API

`ls_perror()`

`ls_sysmsg()`

Equivalent line command

none

Files

none

lsb_usergrpinfo()

Returns LSF user group membership.

DESCRIPTION

`lsb_usergrpinfo()` gets LSF user group membership.

LSF user group is defined in the configuration file `lsb.users`.

The storage for the array of `groupInfoEnt` structures will be reused by the next call.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct groupInfoEnt *lsb_usergrpinfo (char **groups,
                     int *numGroups, int options)

struct groupInfoEnt {
    char *group;
    char *memberList;
    char *adminMemberList;
    int  *numUserShares;
    struct userShares;
    int  options;
    char *pattern;
    char *neg_pattern;
    int  cu_type;
};
```

PARAMETERS

****groups** An array of group names.

***numGroups** The number of group names. `*numGroups` will be updated to the actual number of groups when this call returns.

options The bitwise inclusive OR of some of the following flags:

GRP_RECURSIVE

Expand the group membership recursively. That is, if a member of a group is itself a group, give the names of its members recursively, rather than its name, which is the default.

GRP_ALL

Get membership of all groups.

groupInfoEnt structure fields

group Group name.

memberList ASCII list of member names.

adminMemberList	ASCII list of admin member names.
numUserShares	The number of users with shares.
userShares	The user shares representation.
options	The bitwise inclusive OR of some of the following: **GRP_NO_CONDENSE_OUTPUT** 0x01 Group output is in regular (uncondensed) format. **GRP_CONDENSE_OUTPUT** 0x02 Group output is in condensed format. **GRP_HAVE_REG_EXP** 0x04 **GRP_SERVICE_CLASS** 0x08 Group is a service class. **GRP_IS_CU** 0x10 Group is a compute unit.
pattern	Host membership pattern.
neg_pattern	Negation membership pattern.
cu_type	Compute unit type.

RETURN VALUES

array:groupInfoEnt	On success, returns an array of `groupInfoEnt` structures which hold the group name and the list of names of its members. If a member of a group is itself a group (i.e., a subgroup), then a '/' is appended to the name to indicate this. `*numGroups` is the number of `groupInfoEnt` structures returned.
char:NULL	Function failed.

ERRORS

On failure, returns `NULL` and sets `lsberrno` to indicate the error. If there are invalid groups specified, the function returns the groups up to the invalid ones. It then set `lsberrno` to `LSBE_BAD_GROUP`, that is the specified `(*groups)[*numGroups]` is not a group known to the LSF system. If the first group is invalid, the function returns `NULL`.

SEE ALSO

Related APIs

`lsb_hostgrpinfo()`

Equivalent line command

none

Files

`$LSB_CONFDIR/`*cluster_name*`/lsb.hosts`

SEE ALSO

`$LSB_CONFDIR/`***cluster_name***`/lsb.users`

lsb_userinfo()

Returns the maximum number of job slots that a user can use simultaneously on any host and in the whole local LSF cluster.

DESCRIPTION

`lsb_userinfo()` gets the maximum number of job slots that a user can use simultaneously on any host and in the whole local LSF cluster, as well as the current number of job slots used by running and suspended jobs or reserved for pending jobs. The maximum numbers of job slots are defined in the LSF configuration file `lsb.users` (see `lsb.users`). The reserved user name `default`, defined in the `lsb.users` configuration file, matches users not listed in the `lsb.users` file who have no jobs started in the system.

The returned array will be overwritten by the next call.

SYNOPSIS

```
#include <lsf/lsbatch.h>
struct userInfoEnt *lsb_userinfo(char **users, int *numUsers)

struct userInfoEnt {
    char *user;    int procJobLimit;
    int maxJobs;
    int numStartJobs;
    int numJobs;
    int numPEND;
    int numRUN;
    int numSSUSP;
    int numUSUSP;
    int numRESERVE;
    int maxPendJobs;
};
```

PARAMETERS

****users** An array of user names.

***numUsers** The number of user names.

To get information about all users, set `*numUsers = 0`; `*numUsers` will be updated to the actual number of users when this call returns. To get information on the invoker, set `users = NULL, *numUsers = 1`.

The `userInfoEnt` structures contain the following fields:

user The name of the user or user group.

procJobLimit The maximum number of job slots the user or user group can use on each processor. The job slots can be used by started jobs or reserved for PEND jobs.

maxJobs The maximum number of job slots that the user or user group can use simultaneously in the local LSF cluster. The job slots can be used by started jobs or reserved for PEND jobs.

numStartJobs	The current number of job slots used by running and suspended jobs belonging to the user or user group.
numJobs	The total number of job slots in the LSF cluster for the jobs submitted by the user or user group.
numPEND	The number of job slots the user or user group has for pending jobs.
numRUN	The number of job slots the user or user group has for running jobs.
numSSUSP	The number of job slots for the jobs belonging to the user or user group that have been suspended by the system.
numUSUSP	The number of job slots for the jobs belonging to the user or user group that have been suspended by the user or the LSF system administrator.
numRESERVE	The number of job slots reserved for the pending jobs belonging to the user or user group.
maxPendJobs	The maximum number of pending jobs allowed.

RETURN VALUES

array:userInfoEnt	The function was successful, and `*numUsers` is set to the number of `userInfoEnt` structures returned.
character:NULL	Function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error. If `lsberrno` is `LSBE_BAD_USER, (*users)[*numUsers]` is not a user known to the LSF system. Otherwise, if `*numUsers` is less than its original value, `*numUsers` is the actual number of users found.

SEE ALSO

Related API

`lsb_hostinfo()` - Get information about job server hosts

`lsb_queueinfo()` - Get information about job queues

Equivalent line command

`busers`

Files

`$LSB_CONFDIR/`*cluster_name*`/lsb.users`

lsb_writestream()

Writes a current version eventRec structure into the lsb_stream file.

DESCRIPTION

lsb_writestream() writes an eventrRec to the open streamFile.

This API function is inside `liblsbstream.so`.

SYNOPSIS

```
#include <lsf/lsbatch.h>
int lsb_writestream(struct eventRec *)
```

See lsb_geteventrec() for details on the eventRec structure.

PARAMETERS

***eventRec** Pointer to the eventRec structure.

See lsb_geteventrec() for details on the eventRec structure.

RETURN VALUES

integer:0	The function was successful.
integer:-1	The function failed.

ERRORS

If the function fails, `lsberrno` is set to indicate the error.

SEE ALSO

Related APIs

lsb_closestream(): Close the stream file.

lsb_geteventrec(): Get eventRec structure from an event log file.

lsb_openstream(): Open the stream file.

lsb_puteventrec(): Puts information of an eventRec structure into a log file.

lsb_readstreamline(): Read a line from the stream file.

lsb_streamversion(): Version of the current event type supported by mbatchd.

lsb_readstream(): Read from the stream file.

Equivalent line command

None

Files

`lsb.params`