

Customizing the configuration file

```
LoadL_master          9616/tcp  # Master port number for stream port
LoadL_negotiator      9614/tcp  # Negotiator port number
LoadL_negotiator_collector 9612/tcp  # Second negotiator stream port
LoadL_schedd          9605/tcp  # Schedd port number for stream port
LoadL_schedd_status   9606/tcp  # Schedd stream port for job status data
LoadL_startd          9611/tcp  # Startd port number for stream port
LoadL_master          9617/udp  # Master port number for dgram port
LoadL_startd          9615/udp  # Startd port number for dgram port
```

Step 14: Enable checkpointing

This section tells you how to set up checkpointing for jobs. Checkpointing is a method of periodically saving the state of a job step so that if the step does not complete it can be restarted from the saved state. When checkpointing is enabled, checkpoints can be initiated from within the application at major milestones, or by the user, administrator or LoadLeveler external to the application. Both serial and parallel job steps can be checkpointed.

Once a job step has been successfully checkpointed, if that step terminates before completion, the checkpoint file can be used to resume the job step from its saved state rather than from the beginning. When a job step terminates and is removed from the LoadLeveler job queue, it can be restarted from the checkpoint file by submitting a new job and setting the **restart_from_ckpt = yes** job command file keyword. When a job is terminated and remains on the LoadLeveler job queue, such as when a job step is vacated, the job step will automatically be restarted from the latest valid checkpoint file. A job can be vacated as a result of flushing a node, issuing checkpoint and hold, stopping or recycling LoadLeveler or as the result of a node crash.

Checkpoint keywords

The following is a summary of keywords associated with the checkpoint and restart function.

Administration file keyword summary:

Table 21. Administration file keyword summary

Keyword	Stanza	Default Value	Description
ckpt_dir	Class	Initial working directory	The location to be used for checkpoint files
ckpt_time_limit	Class	Unlimited	Amount of time a job step can take to checkpoint

Note: For more information on these keywords see "Administration file keywords" on page 121.

Configuration file keyword summary:

Table 22. Configuration file keyword summary

Keyword	Default Value	Description
CKPT_CLEANUP_INTERVAL	-1	How frequently, in seconds, the CKPT_CLEANUP_PROGRAM should be run
CKPT_CLEANUP_PROGRAM	No default	Identify an administrator provided program to be run at the interval specified by CKPT_CLEANUP_INTERVAL

Table 22. Configuration file keyword summary (continued)

Keyword	Default Value	Description
MAX_CKPT_INTERVAL	7200 (2 hours)	Maximum interval, in seconds, LoadLeveler will use for checkpointing running job steps
MIN_CKPT_INTERVAL	900 (15 minutes)	Initial (and minimum) interval, in seconds, LoadLeveler will use for checkpointing running job steps
Note: For more information on these keywords see "Configuration file keywords and LoadLeveler variables" on page 126.		

Job command file keyword summary:

Table 23. Job command file keyword summary

Keyword	Default Value	Description
checkpoint	No	Indicates if a job step should be enabled for checkpoint
ckpt_dir	The value of the ckpt_dir keyword in the class stanza of the administration file	The location to be used for checkpoint files
ckpt_file	[jobname.]job_step_id.ckpt	The base name to be used for checkpoint file
ckpt_time_limit	The value of the ckpt_time_limit keyword in the class stanza of the administration file	Amount of time a job step can take to checkpoint
restart_from_ckpt	No	Indicates if a job step is to be restarted from an existing checkpoint file
Note: For more information on these keywords see Chapter 11, "Job command file keywords," on page 93.		

Naming checkpoint files and directories

At checkpoint time, a checkpoint file and potentially an error file will be created. For jobs which are enabled for checkpoint, a control file may be generated at the time of job submission. The directory which will contain these files must pre-exist and have sufficient space and permissions for these files to be written. The name and location of these files will be controlled through keywords in the job command file or the LoadLeveler configuration. The file name specified is used as a base name from which the actual checkpoint file name is constructed. To prevent another job step from writing over your checkpoint file, make certain that your checkpoint file name is unique. For serial jobs and the master task (POE) of parallel jobs, the checkpoint file name will be *<basename>.Tag*. For a parallel job, a checkpoint file is created for each task. The checkpoint file name will be *<basename>.Taskid.Tag*.

The tag is used to differentiate between a current and previous checkpoint file. A control file may be created in the checkpoint directory. This control file contains information LoadLeveler uses for restarting certain jobs. An error file may also be created in the checkpoint directory. The data in this file is in a machine readable format. The information contained in the error file is available in mail, LoadLeveler

Customizing the configuration file

logs or is output of the checkpoint command. Both of these files are named with the same base name as the checkpoint file with the extensions *.cntl* and *.err*, respectively.

See “How to checkpoint a job” on page 383 for more information.

Naming checkpoint files for serial and batch parallel jobs: The following describes the order in which keywords are checked to construct the full path name for a serial or batch checkpoint file:

- Base name for the checkpoint file name
 1. The **ckpt_file** keyword in the job command file
 2. The default file name [*< jobname.>*]*<job_step_id>*.ckpt

Where:

jobname

The *job_name* specified in the Job Command File. If *job_name* is not specified, it is omitted from the default file name

job_step_id

Identifies the job step that is being checkpointed

- Checkpoint Directory Name
 1. The **ckpt_file** keyword in the job command file, if it contains a “/” as the first character
 2. The **ckpt_dir** keyword in the job command file
 3. The **ckpt_dir** keyword specified in the class stanza of the LoadLeveler admin file
 4. The default directory is the initial working directory

Note that two or more job steps running at the same time cannot both write to the same checkpoint file, since the file will be corrupted.

Naming checkpointing files for interactive parallel jobs: The following describes the order in which keywords and variables are checked to construct the full path name for the checkpoint file for an interactive parallel job.

- Checkpoint File Name
 1. The value of the MP_CKPTFILE environment variable within the POE process
 2. The default file name, *poe.ckpt.<pid>*
- Checkpoint Directory Name
 1. The value of the MP_CKPTFILE environment variable within the POE process, if it contains a full path name.
 2. The value of the MP_CKPTDIR environment variable within the POE process.
 3. The initial working directory.

Note: The keywords **ckpt_dir** and **ckpt_file** are not allowed in the command file for an interactive session. If they are present, they will be ignored and the job will be submitted.

Planning considerations for checkpointing jobs

Note: Before you consider using the Checkpoint/Restart function refer to the LoadL.README file in */usr/lpp/LoadL/READMEs* for information on availability and support of this function.

Review the following guidelines before you submit a checkpointing job:

Plan for jobs that you will restart on different nodes

If you plan to migrate jobs (restart jobs on a different node or set of nodes), you should understand the difference between writing checkpoint files to a local file system versus a global file system (such as AFS or GPFS). The `ckpt_file` and `ckpt_dir` keywords in the job command and configuration files allows you to write to either type of file system. If you are using a local file system, before restarting the job from checkpoint, make certain that the checkpoint files are accessible from the machine on which the job will be restarted.

Reserve adequate disk space

A checkpoint file requires a significant amount of disk space. The checkpoint will fail if the directory where the checkpoint file is written does not have adequate space. For serial jobs, one checkpoint file will be created. For parallel jobs, one checkpoint file will be created for each task. Since the old set of checkpoint files are not deleted until the new set of files are successfully created, the checkpoint directory should be large enough to contain two sets of checkpoint files. You can make an accurate size estimate only after you have run your job and noticed the size of the checkpoint file that is created.

Set your checkpoint file size to the maximum

To make sure that your job can write a large checkpoint file, assign your job to a job class that has its file size limit set to the maximum (unlimited). In the administration file, set up a class stanza for checkpointing jobs with the following entry:

```
file_limit = unlimited,unlimited
```

This statement specifies that there is no limit on the maximum size of a file that your program can create.

Choose a unique checkpoint file name

To prevent another job step from writing over your checkpoint file with another checkpoint file, make certain that your checkpoint file name is unique. The `ckpt_dir` and `ckpt_file` keywords give you control over the location and name of these files.

For more information, see “Naming checkpoint files and directories” on page 379.

Checkpoint and restart limitations

- The following items cannot be checkpointed:
 - Programs that are being run under:
 - The dynamic probe class library (DPCL).
 - Any debugger.
 - MPI programs that are *not* compiled with `mpicc_r`, `mpCC_r`, `mpxlf_r`, `mpxlf90_r`, or `mpxlf95_r`.
 - Processes that use:
 - Extended `shmat` support
 - Pinned shared memory segments.
 - Sets of processes in which any process is running a `setuid` program when a checkpoint occurs.
 - Sets of processes if any process is running a `setgid` program when a checkpoint occurs.
 - Interactive parallel jobs for which POE input or output is a pipe.
 - Interactive parallel jobs for which POE input or output is redirected, unless the job is submitted from a shell that had the CHECKPOINT environment variable set to `yes` before the shell was started. If POE is run from inside a shell script and is run in the background, the script must be started from a shell started in the same manner for the job to be checkpointable.

Customizing the configuration file

- Interactive POE jobs for which the **su** command was used prior to checkpointing or restarting the job.
- The node on which a process is restarted must have:
 - The same operating system level (including PTFs). In addition, a restarted process may not load a module that requires a system call from a kernel extension that was not present at checkpoint time.
 - The same switch type (SP Switch or SP Switch2) as the node where the checkpoint occurred.

If any threads in a process were bound to a specific processor ID at checkpoint time, that processor ID must exist on the node where that process is restarted.

- If the LoadLeveler cluster contains nodes running a mix of 32-bit and 64-bit kernels then applications must be checkpointed and restarted on the same set of nodes. See “lckpt - Checkpoint a running job step” on page 152 and “restart_on_same_nodes” on page 115 for more information.
- For a parallel job, the number of tasks and the task geometry (the tasks that are common within a node) must be the same on a restart as it was when the job was checkpointed.
- Any regular file open in a process when it is checkpointed must be present on the node where that process is restarted, including the executable and any dynamically loaded libraries or objects.
- If any process uses sockets or pipes, user callbacks should be registered to save data that may be “in flight” when a checkpoint occurs, and to restore the data when the process is resumed after a checkpoint or restart. Similarly, any user shared memory in a parallel task should be saved and restored.
- A checkpoint operation will not begin on a process until each user thread in that process has released all pthread locks, if held. This can potentially cause a significant delay from the time a checkpoint is issued until the checkpoint actually occurs. Also, any thread of a process that is being checkpointed that does not hold any pthread locks and tries to acquire one will be stopped immediately. There are no similar actions performed for atomic locks (`_check_lock` and `_clear_lock`, for example).
- Atomic locks must be used in such a way that they do not prevent the releasing of pthread locks during a checkpoint. For example, if a checkpoint occurs and thread 1 holds a pthread lock and is waiting for an atomic lock, and thread 2 tries to acquire a different pthread lock (and does not hold any other pthread locks) before releasing the atomic lock that is being waited for in thread 1, the checkpoint will hang.
- A process must not hold a pthread lock when creating a new process (either implicitly using **popen**, for example, or explicitly using **fork**) if releasing the lock is contingent on some action of the new process. Otherwise, a checkpoint could occur which would cause the child process to be stopped before the parent could release the pthread lock causing the checkpoint operation to hang.
- The checkpoint operation will hang if any user pthread locks are held across:
 - Any collective communication calls in MPI or LAPI
 - Calls to `mpc_init_ckpt` or `mp_init_ckpt`
- Processes cannot be profiled at the time a checkpoint is taken.
- There can be no devices other than TTYs or `/dev/null` open at the time a checkpoint is taken.
- Open files must either have an absolute pathname that is less than or equal to `PATHMAX` in length, or must have a relative pathname that is less than or equal

to PATHMAX in length from the current directory at the time they were opened. The current directory must have an absolute pathname that is less than or equal to PATHMAX in length.

- Semaphores or message queues that are used within the set of processes being checkpointed must only be used by processes within the set of processes being checkpointed. This condition is not verified when a set of processes is checkpointed. The checkpoint and restart operations will succeed, but inconsistent results can occur after the restart.
- The processes that create shared memory must be checkpointed with the processes using the shared memory if the shared memory is ever detached from all processes being checkpointed. Otherwise, the shared memory may not be available after a restart operation.
- The ability to checkpoint and restart a process is not supported for B1 and C2 security configurations.
- SP Switch Communications Adapter Type 6–9 (Microchannel TB3 adapters) are not supported.
- A process can only checkpoint another process if it can send a signal to the process. In other words, the privilege checking for checkpointing processes is identical to the privilege checking for sending a signal to the process. A privileged process (the effective user ID is 0) can checkpoint any process. A set of processes can only be checkpointed if each process in the set can be checkpointed.
- A process can only restart another process if it can change its entire privilege state (real, saved, and effective versions of user ID, group ID, and group list) to match that of the restarted process. A set of processes can only be restarted if each process in the set can be restarted.
- The only DCE function supported is DCE credential forwarding by LoadLeveler using the DCE_AUTHENTICATION_PAIR configuration keyword. DCE credential forwarding is for the sole purpose of DFS™ access by the application.

How to checkpoint a job

Checkpoints can be taken either under the control of the user application or external to the application.

The LoadLeveler API **ll_init_ckpt** is used to initiate a serial checkpoint from the user application. For initiating checkpoints from within a parallel application, the API **mpc_init_ckpt** should be used. These APIs allow the writer of the application to determine at what point(s) in the application it would be appropriate save the state of the job. To enable parallel applications to initiate checkpointing, you must use the APIs provided with the Parallel Environment (PE) program. For information on parallel checkpointing, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

It is also possible to checkpoint a program running under LoadLeveler outside the control of the application. There are several ways to do this:

- Use the **llckpt** command to initiate checkpoint for a specific job step
For more information see “llckpt - Checkpoint a running job step” on page 152
- Checkpoint from a program which invokes the **ll_ckpt** API to initiate checkpoint of a specific job step
For more information see “ll_ckpt” on page 231.
- Have LoadLeveler automatically checkpoint all running jobs which have been enabled for checkpoint
- As the result of an **llctl flush** command

Customizing the configuration file

Note: For interactive parallel jobs, the environment variable CHECKPOINT must be set to "yes" in the environment prior to starting the parallel application or the job will not be enabled for checkpoint. For more information see *IBM Parallel Environment for AIX: MPI Programming Guide*.

Table 24. Checkpoint configurations

To specify that:	Do this:
Your job is checkpointable	<ul style="list-style-type: none">• Add either one of the following two options to your job command file:<ol style="list-style-type: none">1. checkpoint = yes<p>This enables your job to checkpoint in any of the following ways:</p><ul style="list-style-type: none">- The application can initiate the checkpoint- Checkpoint from a program which invokes the ll_ckpt API- Checkpoint using the llckpt command- As the result of a flush commandOR2. checkpoint = interval<p>This enables your job to checkpoint in any of the following ways:</p><ul style="list-style-type: none">- The application can initiate the checkpoint- Checkpoint from a program which invokes the ll_ckpt API- Checkpoint using the llckpt command- Checkpoint automatically taken by LoadLeveler- As the result of a flush command• If you would like your job to checkpoint itself, use the API ll_init_ckpt in your serial application, or mpc_init_ckpt for parallel jobs to cause the checkpoint to occur.

Table 24. Checkpoint configurations (continued)

To specify that:	Do this:
<p>LoadLeveler automatically checkpoints your job at preset intervals</p>	<ol style="list-style-type: none"> <li data-bbox="581 262 1448 514"> <p>Add the following option to your job command file:</p> <p>checkpoint = interval</p> <p>This enables your job to checkpoint in any of the following ways:</p> <ul style="list-style-type: none"> • Checkpoint automatically at preset intervals • Checkpoint initiated from user application • Checkpoint from a program which invokes the <code>ll_ckpt</code> API • Checkpoint using the <code>llckpt</code> command • As the result of a flush command <li data-bbox="581 514 1448 1360"> <p>The system administrators must set the following two keywords in the configuration file to specify how often LoadLeveler should take a checkpoint of the job. These two keywords are:</p> <p>MIN_CKPT_INTERVAL = number Where <i>number</i> specifies the initial period, in seconds, between checkpoints taken for running jobs.</p> <p>MAX_CKPT_INTERVAL = number Where <i>number</i> specifies the maximum period, in seconds, between checkpoints taken for running jobs.</p> <p>The time between checkpoints will be increased after each checkpoint within these limits as follows:</p> <ul style="list-style-type: none"> • The first checkpoint is taken after a period of time equal to the MIN_CKPT_INTERVAL has passed. • The second checkpoint is taken after LoadLeveler waits <i>twice as long</i> (MIN_CKPT_INTERVAL X 2) • The third checkpoint is taken after LoadLeveler waits twice as long again (MIN_CKPT_INTERVAL X 4) before taking the third checkpoint. <p>LoadLeveler continues to double this period until the value of MAX_CKPT_INTERVAL has been reached, where it stays for the remainder of the job.</p> <p>A minimum value of 900 (15 minutes) and a maximum value of 7200 (2 hours) are the defaults.</p> <p>You can set these keyword values globally in the global configuration file so that all machines in the cluster have the same value, or you can specify a different value for each machine by modifying the local configuration files.</p>
<p>Your job will not be checkpointed</p>	<p>Add the following option to your job command file:</p> <ul style="list-style-type: none"> • checkpoint = no <p>This will disable checkpoint.</p>

Customizing the configuration file

Table 24. Checkpoint configurations (continued)

To specify that:	Do this:
Your job has successfully checkpointed and terminated. The job has left the LoadLeveler job queue and you would like LoadLeveler to restart your executable from an existing checkpoint file.	<ol style="list-style-type: none">1. Add the following option to your job command file:<ul style="list-style-type: none">• restart_from_ckpt = yes2. Specify the name of the checkpoint file by setting the following job command file keywords to specify the directory and file name of the checkpoint file to be used:<ul style="list-style-type: none">• ckpt_dir• ckpt_file <p>When the job command file is submitted, a new job will be started which uses the specified checkpoint file to restart the previously checkpointed job.</p> <p>The job command file which was used to submit the original job should be used to restart from checkpoint. The only modifications to this file should be the addition of restart_from_ckpt = yes and ensuring ckpt_dir and ckpt_file point to the appropriate checkpoint file.</p>
Your job has successfully checkpointed. The job has been vacated and remains on the LoadLeveler job queue.	<p>When the job restarts, if a checkpoint file is available, the job will be restarted from that file.</p> <p>If a checkpoint file is not available upon restart, the job will be started from the beginning.</p>

Remove old checkpoint files

To keep your system free of checkpoint files that are no longer necessary, LoadLeveler provides two keywords to help automate the process of removing these files.

ckpt_cleanup_program = *name of program to be run*

Identifies an administrator provided program which is to be run at the interval specified by the **ckpt_cleanup_interval** keyword. The intent of this program is to delete old checkpoint files created by jobs running under LoadLeveler during the checkpoint process. The name of the program to be run should be fully qualified and must be accessible and executable by LoadLeveler.

A sample program to remove checkpoint files is provided in the `/usr/lpp/LoadL/full/samples/lckpt/rmckptfiles.c` file.

ckpt_cleanup_interval = *number*

Specifies the interval, in seconds, at which the **schedd** daemon will run the **ckpt_cleanup_program**. This number must be a positive integer.

Note: Both **ckpt_cleanup_program** and **ckpt_cleanup_interval** must contain valid values to automate this process.

Step 15: Specify process tracking

When a job terminates, its orphaned processes may continue to consume or hold resources, thereby degrading system performance, or causing jobs to hang or fail. Process tracking allows LoadLeveler to cancel any processes (throughout the entire cluster), left behind when a job terminates. Using process tracking is optional. There are two keywords used in specifying process tracking:

PROCESS_TRACKING

To activate process tracking, set **PROCESS_TRACKING=TRUE** in the LoadLeveler global configuration file. By default, **PROCESS_TRACKING** is set to **FALSE**.