

**ECFLOW CLIENT COMMANDS**

**For BETA version of ecFlow**

**March 2012**

This document is produced as a supplement to the online training course.

© Copyright 2012

European Centre for Medium Range Weather Forecasts  
Shinfield Park, Reading, RG2 9AX, United Kingdom

Literary and scientific copyrights belong to ECMWF and are reserved in all countries.

The information within this publication is given in good faith and considered to be true, but ECMWF accepts no liability for error, omission and for loss or damage arising from its use.

ECFLOW CLIENT Commands  
ecFlow\_client --help

|  |  |
| --- | --- |
| **Client command option** | **Description** |
| --abort arg | Mark task as aborted. For use in the '.ecf' script file \*only\*. Hence the context is supplied via environment variables  arg1 = (optional) string(reason)  Optionally provide a reason why the abort was raised  Usage:  ecflow\_client --abort=reason |
| --alter arg | Alter the node according to the options. To add/delete/change server variables use '/' for the path.  arg1 = [ delete | change | add | set\_flag | clear\_flag] # one option must be specified  arg2 = (dependant on arg1   * For delete: [ variable | time | today | date | day | cron | event | meter | label | trigger | complete | repeat | limit | inlimit | limit\_path | zombie ] * For change: [ variable | clock\_type | clock\_gain | clock\_date | event | meter | label | trigger | complete | repeat | limit\_max | limit\_value | defstatus ] * For add: [ variable, time | today | date | day ] * For set\_flag and clear\_flag: [ force\_aborted | user\_edit | task\_aborted | edit\_failed | ecfcmd\_failed | no\_script | killed | migrated | late | message | complete | queue\_limit | task\_waiting | locked | zombie ]   arg3 = name/value # when changing, attributes like variable, meter, event, label, limits we expect arguments to be quoted  arg4 = new\_value # specifies the new value only used for 'change' values with spaces must be quoted  arg5 = paths: # At least one path required. The paths must start with a leading '/' character |
| --begin arg | Begin playing the definition in the server. Expects zero or a single quoted string  arg1 = suite-name | Nothing | force  Play the chosen suite, if no arg specified, play all suites, in the definition force means reset the begin status on the suites and bypass checks. This is only required if suite-name is provide as the first argument. Using force can cause the creation of zombies  Usage:  --begin # will begin all suites  --begin="--force" # reset and then begin all suites, bypassing any checks. Note string must be quoted  --begin="mySuite" # begin playing suite of name 'mySuite'  --begin="mySuite --force" # reset and begin playing suite 'mySuite', bypass check |
| --ch\_add arg | Add a set of suites, to an existing handle  arg1 = handle(integer) # The handle must be an integer that is > 0  arg2 = names # should be a list of suite names, names not in the definition are ignored  Usage:  --ch\_add=10 s2 s3 s4 # add suites s2 s3,s4 to handle 10  An error is returned if the handle had not previously been registered  The handle is created with --ch\_register command. To list all suites and handles use –suites |
| --ch\_auto\_add arg | Change an existing handle so that new suites can be added automatically  arg1 = handle(integer) # The handle must be an integer that is > 0  arg2 = true | false # true means add new suites to my list, when they are created  Usage:  --ch\_auto\_add=10 true # modify handle 10 so that new suites, get added automatically to it  --ch\_auto\_add=10 false # modify handle 10 so that no new suites are added  The handle is created with --ch\_register command. To list all suites and handles use –suites. |
| --ch\_drop arg | Drop/de-register the client handles. Client must ensure un-used handle are dropped otherwise they will stay, in the server.  arg1 = handle(integer) # The handle must be an integer that is > 0  Usage:  --ch\_drop=10 # drop the client handle 10  An error is returned if the handle had not previously been registered. The handle stored on the local client is set to zero  To list all suites and handles use --suites  --ch\_drop\_user arg # Drop/de-register all handles associated with the given user. If no user provided will drop for current user. Client must ensure un-used handle are dropped otherwise they will stay, in the server.  arg1 = user # The user to be drooped, if left empty drop current user  Usage:  --ch\_drop\_user=ma0 # drop all handles associated with user ma0  --ch\_drop\_user # drop all handles associated with current user  An error is returned if no registered handles To list all suites and handles use –suites |
| --ch\_register arg | Register interest in a set of suites. If a definition has lots of suites, but the client is only interested in a small subset. Then using this command can reduce network bandwidth and synchronisation will be quicker. This command will create a client handle, which must be used for any other changes. The newly created handle can be shown with the --suites command  This option affects news() and sync() commands  arg1 = true | false # true means add new suites to my list, when they are created  arg2 = names # should be a list of suite names, names not in the definition are ignored  Usage:  --ch\_register=true s1 s2 s3 # register interest in suites s1,s2,s3 and any new suites  --ch\_register=false s1 s2 s3 # register interest in suites s1,s2,s3 only  --ch\_register=false # register handle, suites will be added later on |
| --ch\_rem arg | Remove a set of suites, from an existing handle  arg1 = handle(integer) # The handle must be an integer that is > 0  arg2 = names # should be a list of suite names, names not in the definition are ignored  Usage:  --ch\_rem=10 s2 s3 s4 # remove suites s2 s3,s4 from handle 10  The handle is created with --ch\_register command. To list all suites and handles use –suites |
| --check arg | Checks the expression and limits. Will also check trigger references. Trigger references where suite is loaded, but paths don't resolve are reported as errors  arg = [ \_all\_ | list of node paths ]  Usage:  --check=\_all\_ # Checks the whole suite  --check /s1 /s2/f1/t1 # Check suite /s1 and task t1 |
| --check\_pt arg | Forces the definition file in the server to be written to disk \*or\* allow mode and interval to be changed.  arg1 = (optional) mode [ never | on\_time | on\_time:<integer> | always | <integer>]  mode:   * never : Never check point the definition in the server * on\_time : Turn on automatic check pointing at interval stored on server * on\_time<integer> : Turn on automatic check point, with the specified interval in seconds * always : Check point at any change in node tree, \*NOT\* recommended for large definitions * <integer> : This specifies the interval in seconds when server should automatically check pt. This will only take effect of mode is on\_time/CHECK\_ON\_TIME Should ideally be a value greater than 60 seconds, default is 120 seconds   Usage:  --check\_pt  Immediately check point the definition held in the server  --check\_pt never  Switch off check pointing  --check\_pt on\_time  Start automatic check pointing at the interval stored in the server  --check\_pt 180  Change the check pt interval to 180 seconds  --check\_pt on\_time:90  Change mode and interval, to automatic check pointing every 90 seconds |
| --checkJobGenOnly arg | Test hierarchical Job generation only, for chosen Node. The jobs are generated independent of the dependencies.  This will generate the jobs \*only\*, i.e. no job submission. Used for checking job generation only.  arg = node path | arg = NULL  If no node path specified generates for all Tasks in the definition. For Test only |
| --complete | Mark task as complete. For use in the '.ecf' script file \*only\*  Hence the context is supplied via environment variables  Usage:  ecflow\_client --complete |
| --debug, -d | Dump out client environment settings for debug. Set environment variable ECF\_DEBUG\_CLIENT for additional debug info. |
| --debug\_server\_off | Disables debug output from the server |
| --debug\_server\_on | Enables debug output from the server |
| --delete arg | Deletes the specified node(s) or \_ALL\_ existing definitions( i.e. delete all suites) in the server  arg1 = [ force | yes ](optional) # Use this parameter to bypass checks, i.e. for active or submitted tasks  arg2 = yes(optional) # Use 'yes' to bypass the confirmation prompt  arg3 = node paths | \_all\_ # node paths must start with a leading '/'  Usage:  --delete=\_all\_ # Delete all suites in server. Use with care.  --delete=/suite/f1/t1 # Delete node at /suite/f1/t1. This will prompt  --delete=force /suite/f1/t1 # Delete node at /suite/f1/t1 even if active or submitted  --delete=force yes /s1 /s2 # Delete suites s1,s2 even if active or submitted, bypassing prompt |
| --edit\_script arg | Allows user to edit the script and submit it.  arg1 = path to task # The path to the task/alias  arg2 = [ edit | pre\_process | submit | pre\_process\_file | submit\_file ]   * edit : will return the script file to standard out. The script will include used variables enclosed between %comment/%end at the start of the file * pre\_process: Will return the script file to standard out. The script will include used variables enclosed between %comment/%end at the start of the file and with all %include expanded * submit : Will extract the used variables from the supplied file, i.e. between the %comment/%end and use these them to generate the job using the ecf file accessible from the server * pre\_process\_file : Will pre process the user supplied file * submit\_file : Like submit, but the supplied file, is submitted by the server   The last 2 options allow complete freedom to debug the script file  arg3 = [ path\_to\_script\_file ] # needed for option [ pre\_process\_file | submit\_file ]  arg4 = create\_alias (optional) # default value is false, for use with 'submit\_file' option  arg5 = no\_run (optional) # default value is false, i.e. immediately run the alias is no\_run is specified the alias in only created  Usage:  --edit\_script /path/to/task edit > script\_file  # server returns script with the used variables to standard out. The user can choose to edit this file.  --edit\_script /path/to/task pre\_process > pre\_processed\_script\_file  # Server will pre process the ecf file accessible from the server (i.e. expand all %includes) and return the file to standard out.  --edit\_script /path/to/task submit script\_file  # Will extract the used variables in the 'script\_file' and will uses these variables during variable substitution of the ecf file accessible by the server. This is then submitted as a job  --edit\_script /path/to/task pre\_process\_file file\_to\_pre\_process  # The server will pre-process the user supplied file and return the contents to standard out  --edit\_script /path/to/task submit\_file file\_to\_submit  # Will extract the used variables in the 'file\_to\_submit' and will uses these variables during variable substitution, the file is then submitted for job generation by the server  --edit\_script /path/to/task submit\_file file\_to\_submit create\_alias  # Like the previous example but will create and run as an alias |
| --event arg | Change event. For use in the '.ecf' script file \*only\*. Hence the context is supplied via environment variables  arg1(string | int) = event-name  Usage:  ecflow\_client --event=ev |
| --file arg | File command. Used to return the chosen file. By default will return the script.  arg1 = path to node  arg2 = (optional) [ script<default> | job | jobout | manual | kill | stat ]   * kill will attempt to return output of ECF\_KILL\_CMD, i.e. the file %ECF\_JOB%.kill * stat will attempt to return output of ECF\_STATUS\_CMD, i.e. the file %ECF\_JOB%.stat   arg3 = (optional) max\_lines = 10000 <default> |
| --force arg | Force a node to a given state, Or set its event.  arg1 = [ unknown | complete | queued | submitted | active | aborted | clear | set ]  arg2 = (optional) recursive # Applies state to node and recursively to all its children  arg3 = (optional) full # Set repeat variables to last value, only works in conjunction with recursive option  arg4 = path\_to\_node or path\_to\_node:<event>: paths must begin with '/'  Usage:  --force=complete /suite/t1 /suite/t2 # Set task t1 & t2 to complete  --force=clear /suite/task:ev # Clear the event 'ev' on task /suite/task  --force=complete recursive /suite/f1 # Recursively set complete all children of /suite/f1 |
| --force-dep-eval | Force dependency evaluation. Used for DEBUG only. |
| --free-dep arg | Free dependencies for a node. Defaults to triggers  arg1 = (optional) trigger  arg2 = (optional) all # Free trigger, date and all time dependencies  arg3 = (optional) date # Free date dependencies  arg4 = (optional) time # Free all time dependencies i.e. time, day, today, cron  arg5 = List of paths. # At least one required. Must start with a leading '/'  Usage:  --free-dep=/s1/t1 /s2/t2 # free trigger dependencies for task's t1,t2  --free-dep=all /s1/f1/t1 # free all dependencies of /s1/f1/t1  --free-dep=date /s1/f1 # free holding date dependencies of /s1/f1 |
| --get arg | Get all suite node trees from the server and write to standard out. The output is parse-able and can be used to re-load the definition  arg = NULL | arg = node path  Usage:  --get # gets the definition from the server, and writes to standard out  --get /s1 # gets the suite from the server, and writes to standard out |
| --get\_state arg | Get all suite node trees from the server and write to standard out. The output will include state information and hence is not parse-able  arg = NULL | arg = node path  Usage:  --get\_state # gets the definition from the server, and writes to standard out  --get\_state /s1 # gets the suite from the server, and writes to standard out |
| --halt arg | Stops server communication with jobs and new job scheduling. Also stops automatic check pointing  arg1 = yes(optional) # use to bypass confirmation prompt |
| --help, -h | Produce help message |
| --host arg | If specified will override the environment variable ECF\_NODE and default host, localhost  Usage:  --host class01 # set the host to class01 |
| --init arg | Mark task as started (active). For use in the '.ecf' script file \*only\*. Hence the context is supplied via environment variables  arg = process\_or\_remote\_id. The process id of the job or remote\_id. Using remote id allows the jobs to be killed  Usage:  ecflow\_client --init=$$ |
| --kill arg | Kills the job associated with the node. If a family or suite is selected, will kill hierarchically. Kill uses the ECF\_KILL\_CMD variable. After variable substitution it is invoked as a command. The command should be written in such a way that the output is written to %ECF\_JOB%.kill  i.e. /home/ma/emos/bin/ecfkill %USER% %HOST% %ECF\_RID% %ECF\_JOB% > %ECF\_JOB%.kill 2>&1::  Usage:  --kill /s1/f1/t1 /s1/f2/t2 # kill the jobs for tasks t1 and t2  --file /s1/f1/t1 kill # write to standard out the '.kill' file for task /s1/f1/t1 |
| --label arg | Change Label. For use in the '.ecf' script file \*only\*. Hence the context is supplied via environment variables  arg1 = label-name  arg2 = The new label value  The labels values can be single or multi-line(space separated quoted strings  Usage:  ecflow\_client --label=progressed to task merlin |
| --load arg | Loads definition file, into server. The loaded definition will be checked for valid trigger and complete expressions, additionally in limits will be validated. If the server already has the 'suites' of the same name, then an error message is issued.  The suite can be overridden however if the force option. To just check the definition and not send to server, use 'check\_only'  arg1 = path to the definition file  arg2 = [ force | check\_only ] # default = false for both  Usage:  --load=/my/home/exotic.def # will error if suites of same name exists  --load=/my/home/exotic.def force # overwrite suite's of same name  --load=/my/home/exotic.def check\_only # Just check, don't send to server |
| --log arg | Log Cmd. Used to create a new log file. The user must ensure that a valid path is specified. However for \*test\* we also provide functionality to get the log file. This could be a very large file, and should not generally be used; optionally the number of lines can be specified.  arg1 = [ get | clear | flush | new | path ]   * get - Outputs the log file to standard out. Not for general usage. The second argument can specify how many lines to return * clear - Clear the log file of its contents. * flush - Flush and close the log file. (Only temporary) next time server writes to log, it will be opened again. Hence it best to halt the server first * new - Flush and close the existing log file, and start using the new path, for the log file location. Will return an error if the directory part does not exist * path - Returns the path name to the existing log file   arg2 = [ new\_path | optional last n lines ]  If get specified can specify lines to get. Value must be convertible to an integer. Otherwise if arg1 is 'new' then the second argument must be a path  Usage:  --log=new /path/to/new/log/file # Close and flush log file, and create a new log file  --log=get 200 # Write the last 200 lines of the log file to standard out  --log=clear # Clear the log file. The log is now empty  --log=flush # Flush and close log file. |
| --meter arg | Change meter. For use in the '.ecf' script file \*only\*. Hence the context is supplied via environment variables  arg1(string) = meter-name  arg2(int) = the new meter value  Usage:  ecflow\_client --meter=my\_meter 20 |
| --msg arg | Writes the input string to the log file  arg1 = string  Usage:  --msg="place me in the log file" |
| --order arg | Re-orders the nodes held by the server  arg1 = node path  arg2 = [ top | bottom | alpha | order | up | down ]  It should be noted that in the absence of triggers and time/date dependencies, the tasks are submitted in order.  This changes the order and hence affects the submission order::   * top raises the node within its parent, so that it is first * bottom lowers the node within its parent, so that it is last * alpha Arranges for all the peers of selected note to be sorted alphabetically * order Arranges for all the peers of selected note to be sorted in reverse alphabet * up Moves the selected node up one place amongst its peers * down Moves the selected node down one place amongst its peers   This command can fail because:   * The node path does not exist in the server * The order\_type is not does not match one of arg2   Usage:  --order=/suite/f1 top # move node f1 to the top |
| --ping | Check if server is running on given host/port. Result reported to standard output.  Usage:  --ping --host=oetzi --port=3144 # Check if server alive on host oetzi & port 3144  --ping --host=fred # Check if server alive on host fred and port ECF\_PORT,  # otherwise default port of 3141  --ping # Check if server alive by using environment variables  # ECF\_NODE and ECF\_PORT  If ECF\_NODE not defined uses 'localhost', if ECF\_PORT not defined assumes 3141 |
| --plug arg | Plug command is used to move nodes. The destination node can be on another server In which case the destination path should be of the form '<host>:<port>/suite/family/task  arg1 = path to source node  arg2 = path to the destination node  This command can fail because:   * Source node is in a 'active' or 'submitted' state * Another user already has an lock * source/destination paths do not exist on the corresponding servers * If the destination node path is empty, i.e. only host:port is specified, then the source node must correspond to a suite. * If the source node is added as a child, then its name must be unique amongst its peers   Usage:  --plug=/suite macX:3141 # move the suite to ecFlow server on host(macX) and port(3141) |
| --port arg | If specified will override the environment variable ECF\_PORT and default port number of 3141  Usage:  --port 3142 # set the port number to 3142 |
| --reloadwsfile | Reload the white list file.  Raises an error if file does not exist or fails to parse |
| --replace arg | Replaces a node in the server, with the given path. Can also be used to add nodes in the server  arg1 = path to node must exist in the client defs (arg2). This is also the node we want to replace in the server  arg2 = path # path to client definition file provides the definition of the new node  arg3 = (optional) [ parent | false ] (default = parent) #create parent families or suite as needed, when arg1 does not exist in the server  arg4 = (optional) force (default = false) # Force the replacement even if it causes zombies to be created  Replace can fail if:   * The node path(arg1) does not exist in the provided client definition(arg2) * The client definition(arg2) must be free of errors * If the third argument is not provided, then node path(arg1) must exist in the server * Nodes to be replaced are in active/submitted state, in which case arg4(force) can be used   Usage:  --replace=/suite/f1/t1 /tmp/client.def parent # Add/replace node tree /suite/f1/t1  --replace=/suite/f1/t1 /tmp/client.def false force # replace t1 even if its active or submitted |
| --requeue arg | Re queues the specified node(s)  arg1 = (optional) [ abort | force ]  abort = re-queue only aborted tasks below node  force = Force the re-queuing even if there are nodes that are active or submitted  arg2 = list of node paths. The node paths must begin with a leading '/' character  Usage:  --requeue=abort /suite/f1 # re-queue all aborted children of /suite/f1 |
| --restart | Start job scheduling, communication with jobs, and respond to all requests. The following table shows server behaviour in the different states  ------------------------------------------------------------------------------------  | Server State | User Request | Task Request |Job Scheduling | Auto-Check-pointing |  |--------------|--------------|--------------|---------------|---------------------|  | RUNNING | yes | yes | yes | yes |  | SHUTDOWN | yes | yes | no | yes |  | HALTED | yes | no | no | no |  ------------------------------------------------------------------------------------ |
| --restore\_from\_checkpt | Ask the server to load the definition from a check point file. The server must be halted and the definition in the server must be deleted first, otherwise an error is returned |
| --resume arg | Resume the given node  Usage  --resume /suite/f1/t1 |
| --rid arg | If specified will override the environment variable ECF\_RID, Can only be used for child commands |
| --run arg | Ignore triggers, limits, time or date dependencies, just run the Task  arg1 = (optional) force # Forcibly run, even if there are nodes that are active or submitted. This can result in zombie creation.  arg2 = node path(s) #The paths must begin with a leading '/' character. If the path is /suite/family will recursively run all tasks. When providing multiple paths avoid running the same task twice |
| --server\_load arg | Shows the server load graphically. Will parse the log file and create a graphical view of the server load. If no log file is provided, then the log file is obtained from the server. If this file is accessible from the client, then a graphical view of the server load is shown. This command assumes that gnuplot is available on $PATH. This command produces two files in the current working directory ‘gnuplot.dat’ and 'gnuplot.script'. This generated script can be manually changed, to see different rendering effects. i.e. just run 'gnuplot gnuplot.script'  arg1 = <optional> path to log file  If the path to log file is known, it is \*preferable\* to use this, rather than requesting it from the server  Usage:  --server\_load=/path/to\_log\_file # parses log and shows a gnuplot window  --server\_load # Use server log to produce gnuplot window avoid if log file path is known |
| --show arg | Used to show state of the definition returned from the server to standard output. This command can \*only\* be used in a group command, and will only work if it is preceded with a get command.  arg1 = [ state ]  The output of show has 2 options: i.e.   * no arguments: With no arguments the output is re-parse-able   i.e. --group="get ; show"   * state:   With the argument 'state' the output will show node state and the trigger abstract syntax tree, this output is not parse-able  Usage:  --group="get ; show state"  --group="get /s1; show"  --group="get /s1; show state"  --group arg Group commands. Allows a series of ';' separated commands to be executed in the server Some commands like halt, shutdown and terminate will prompt the user. To bypass the prompt provide 'yes' as an additional parameter. See example below.  arg = string  Usage:  --group="halt yes; reloadwsfile; restart" # halt server, bypass the confirmation prompt, reload white list file, restart server  --group="get; show" # get server defs, and write to standard output  --group="get /s1; show state" # get suite 's1', and write state to standard output |
| --shutdown arg | Stops server from scheduling new jobs  arg1 = yes(optional) # use to bypass confirmation prompt |
| --stats | Returns the server statistics |
| --status arg | Shows the status of a job associated with a task. If a family or suite is selected, will invoke status command hierarchically.  Status uses the ECF\_STATUS\_CMD variable. After variable substitution it is invoked as a command. The command should be written in such a way that the output is written to %ECF\_JOB%.stat  i.e. /home/ma/emos/bin/ecfstatus %USER% %HOST% %ECF\_RID% %ECF\_JOB% > %ECF\_JOB%.stat 2>&1::  Usage::  --status /s1/f1/t1 /s1/f2/t2  --file /s1/f1/t1 stat # write to standard out the '.stat' file |
| --suites | Returns the list of suites, in the order defined in the server. Additionally returns all the client handle, and the suites they reference |
| --suspend arg | Suspend the given node. This prevents job generation for the given node, or any child node  Usage  --suspend /suite/f1/t1 |
| --terminate arg | Terminate the server  arg1 = yes(optional) # use to bypass confirmation prompt |
| --version, -v | Show ecFlow version number |
| --wait arg | Evaluates an expression and block while the expression is false. For use in the '.ecf' file \*only\*, hence the context is supplied via environment variables  arg1 = string(expression)  Usage:  ecflow\_client --wait="/suite/taskx == complete" |
| --why arg | Show the reason why a node is not running. Can only be used with the group command. The group command must include a 'get' command (i.e. returns the server defs). The why command take a optional string argument representing a node path  Will return reason why the node is holding and for all its children. If no arguments supplied will report on all nodes.  arg = node path | arg = NULL  Usage:  --group="get; why" # returns why for all holding nodes  --group="get; why /suite/family" # returns why for a specific node |
| --zombie\_adopt arg | Locates the task in the zombie list, and sets to adopt. Next time the child commands (init, event, meter, label, abort, complete) communicate with the server, the password on the zombie is adopted by the task. The zombie is then deleted.  arg = path to task |
| --zombie\_block arg | Locates the task in the zombie list, and blocks it. This is default behaviour of the child commands (init, event, meter, label, abort, complete) when the server cannot match the passwords. The each child commands will continue attempting to connect to the server for 24 hours, and will then return an error. The connection timeout can be configured with environment ECF\_TIMEOUT  arg = path to task |
| --zombie\_fail arg | Locates the task in the zombie list, and sets to fail. Next time the child commands (init, event, meter, label, abort, complete) communicate with the server, they will be set to fail. Depending on the job setup this may force an abort, the abort will also fail. Hence job structure should use 'set -e' in the error trapping functions to prevent infinite recursion. The server zombie is automatically deleted after 24 hrs  arg = path to task |
| --zombie\_fob arg | Locates the task in the zombie list, and sets to fob. Next time the child commands (init, event, meter, label, abort, complete) communicate with the server, they will complete successfully (but without updating the node tree) allowing the job to finish. The server zombie is automatically deleted after 24 hrs  arg = path to task |
| --zombie\_get | Returns the list of zombies from the server. |
| --zombie\_remove arg | Locates the task in the zombie list, and removes it. Since a job typically has many child commands(i.e. init, complete, event, meter, label) the zombie may reappear  arg = path to task |