Glossary

aborted

Is a *node status* .

When the ECF_JOB_CMD fails or the *job file* sends a *ecflow_client* –abort *child command* , then the task is placed into an aborted state.

active

Is a *node status* .

If *job creation* was successful, and *job file* has started, then the *ecflow_client* –init *child command* is received by the *ecflow_server* and the *task* is placed into a active state

autocancel

autocancel is a way to automatically delete a *node* which has completed.

The delete may be delayed by an amount of time in hours and minutes or expressed in days. Any node may have a single autocancel attribute. If the auto cancelled node is referenced in the *trigger* expression of other nodes it may leave the node waiting. This can be solved by making sure the *trigger* expression also checks for the *unknown* state. i.e...:

```
trigger node_to_cancel == complete or node_to_cancel == unknown
```

This guards against the ‘node_to_cancel’ being undefined or deleted

For python see ecflow.Autocancel and ecflow.Node.add_autocancel . For text BNF see autocancel

check point

The check point file is like the *suite definition* , but includes all the state information. It is periodically saved by the *ecflow_server* (this period can be changed, see ecflow_client --help check_pt)

It can be used to recover the state of the node tree should server die, or machine crash.

By default when a *ecflow_server* is started it will look to load the check point file.

The default check point file name is <host>.<port>.ecf.check. This can be overridden by the ECF_CHECK environment variable.

The check point file format is the same as the defs file format.(from release 4.7.0 onwards). However, the indentation has been removed to preserve space. To view with indentation use :

```
ecflow_client --load=<check_point_file> print check_only
```

child command

Child command’s(or task requests) are called from within the *ecf script* files.

The table also includes the default action(from version 4.0.4) if the child command is part of a zombie.

'block' means the job will be held by ecflow_client command. Until time out, or manual/automatic intervention.

Child Command	**Description**	**Zombie (default action)**
ecflow_client –init	Sets the *task* to the *active status*	block
ecflow_client –event	Set an event	fob
ecflow_client –meter	Change a meter	fob
ecflow_client –label	Change a label	fob
ecflow_client –wait	wait for an expression to evaluate.	block
ecflow_client –queue	Update queue step in server	block
ecflow_client –abort	Sets the *task* to the *aborted status*	block
ecflow_client –complete	Sets the *task* to the *complete status*	block

The following environment variables must be set for the child commands. ECF_HOST, ECF_NAME ,ECF_PASS and ECF_RID. See *ecflow_client* .

clock	A clock is an attribute of a *suite* . A gain can be specified to offset from the given date. The hybrid and real clock’s always runs in phase with the system clock (UTC in UNIX) but can have any offset from the system clock. The clock can be : • hybrid clock • real clock *(default if not explicitly specified)* *time* , *day* and *date* and *cron dependencies* work a little differently under the clocks. If the *ecflow_server* is *shutdown* or *halted* the job *scheduling* is suspended. If this suspension is left for period of time, then it can affect task submission under **hybrid** and **real** clocks. In particular it will affect *task* s with *time* , *today* or *cron dependencies* . • dependencies *with time series, can result in missed time slots:* `time 10:00 20:00 00:15    # If server is suspended > 15 minutes, time slots can be missed` `time +00:05 20:00 00:15   # start 5 minutes after the start of the suite, then every 15m until 20:00` • *When the server is placed back into* running *state any time* dependencies *with an expired time slot are submitted straight away. i.e... if* ecflow_server *is* halted *at 10:59 and then placed back into* running *state at 11:20* *time 11:00* *Then any* task *with an expired single time slot dependency will be submitted straight away.* For python see ecflow.Clock and ecflow.Suite.add_clock . For text BNF see clock
complete	Is a *node status* . The node can be set to complete: • By the *complete expression* • At job end when the *task* receives the *ecflow_client* –complete *child command* • Manually via the command line or GUI. When this happens any time attributes are expired in order.
complete expression	Force a node to be complete **if** the expression evaluates, without running any of the nodes. This allows you to have tasks in the suite which a run only if others fail. In practice the node would need to have a *trigger* also. For python see ecflow.Expression and ecflow.Node.add_complete

cron

Like *time* , cron defines time dependency for a *node* , but it will be repeated **indefinitely**

```
cron -w <weekdays> -d <days> -m <months> <start_time> <end_time> <increment>
# weekdays:   range [0...6], Sunday=0, Monday=1, etc    e.g. -w, 0,3,6
# days:       range [1..31]                             e.g. -d 1,2,20,30    if the month does not have a
day, i.e. February 21st it is ignored
# months:     range [1..12]                             e.g. -m 5,6,7,8
# start_time: The starting time. format hh:mm           e.g. 15:21
# end_time:   The end time, if multiple times used
# increment:  The increment in time if multiple times are given

-w day of the week   valid values are , 0  6 where 0 is Sunday , 1 is Monday etc AND
                     0L6L, where 0L means last Sunday of the month, and 1L means the last Monday of the
month, etc
                     It is an error to overlay, i.e. cron -w 0,1,2,1L,2L,3L   23:00  will throw an exception
-d day of the month   valid values are in range 0-31,L   Extended so that we now use 'L' to mean the last
day of the month
-m month              valid values are in range 0-12

cron 11:00                           # single time
cron 10:00 22:00 00:30               # <start> <finish> <increment>
cron +00:20 23:59 00:30              # relative to suite start time, or when re-queued  as part of a repeat
loop. Note: maximum relative time is 24 hours
cron -w 0,1 10:00 11:00 01:00        # run every Sunday & Monday at 10 and 11 am
cron -d 15,16 -m 1 10:00 11:00 01:00 # run 15,16 January at 10 and 11 am
cron -w 5L 23:00                     # run on *last* Friday(5L) of each month at 23pm,
                                     # Python: cron = Cron("23:00",last_week_days_of_the_month=[5])
cron -w 0,1L 23:00                   # run every Sunday(0) and *last* Monday(1L) of the month at 23pm
                                     # Python: cron = Cron("23:00",days_of_week=[0],
last_week_days_of_the_month=[1])
cron -w 0L,1L,2L,3L,4L,5L,6L 10:00   # run on the last Monday,Tuesday..Saturday,Sunday of the month at 10 am
                                     # Python: cron = Cron("10:00",last_week_days_of_the_month=
[0,1,2,3,4,5,6])
cron -d 1,L  23:00                   # Run on the first and last of the month at 23pm
                                     # Python: cron = Cron("23:00",days_of_week=[1],
last_day_of_the_month=True)
```

When the node becomes complete it will be *queued* **immediately**. This means that the suite will **never** complete, and the output is not directly accessible through *ecflow_ui*

If tasks abort, the *ecflow_server* will not schedule it again.

If the time the job takes to complete is longer than the interval a time "slot" is missed, e.g.

    cron 10:00 20:00 01:00

if the 10:00 run takes more than an hour, the 11:00 run will be skipped.

If the cron defines months, days of the month, or week days or a single time slot the it relies on a day change, hence if a *hybrid clock* is defined, then it will be set to *complete* at the beginning of the *suite* , without running the corresponding job. Otherwise under a hybrid clock the *suite* would never *complete*.

Since a cron never completes, it would not be wise to use with repeat attributes, since repeat requires completion in order to increment.

For python see ecflow.Cron and ecflow.Node.add_cron . For text BNF see cron

date

This defines a date dependency for a node.

There can be multiple date dependencies. In this case the node is free to run when any of dates occur.

The European format is used for dates, which is: dd.mm.yy as in 31.12.2007. Any of the three number fields can be expressed with a wildcard * to mean any valid value. Thus, 01.*.* means the first day of every month of every year.

If a *hybrid clock* is defined, any node held by a date dependency will be set to *complete* at the beginning of the *suite* , without running the corresponding job. Otherwise under a hybrid clock the *suite* would never *complete* .

For python see: ecflow.Date and ecflow.Node.add_date . For text BNF see date

day

This defines a day dependency for a node.

There can be multiple day dependencies. If any of day's occur the effect is to have **or** type behaviour.

If a *hybrid clock* is defined, any node held by a day dependency will be set to *complete* at the beginning of the *suite* , without running the corresponding job. Otherwise under a hybrid clock the *suite* would never *complete* .

For python see: ecflow.Day and ecflow.Node.add_day . For text BNF see day

defstatus

Defines the default *status* for a task/family to be assigned to the *node* when the begin command is issued.

By default *node* gets queued when you use begin on a *suite* . defstatus is useful in preventing suites from running automatically once begun or in setting tasks complete so they can be run selectively.

For python see ecflow.DState and ecflow.Node.add_defstatus . For text BNF see defstatus

dependencies

Dependencies are attributes of node, that can suppress/hold a *task* from taking part in *job creation* .

They include *trigger* , *date* , *day* , *time* , *today* , *cron* , *complete expression* , *inlimit* and *limit* .

A *task* that is dependent cannot be started as long as some dependency is holding it or any of its **parent** *node* s.

The *ecflow_server* will check the dependencies every minute, during normal *scheduling* **and** when any *child command* causes a state change in the *suite definition* .

directives

Directives appear in a ecf script. (i.e. typically .ecf file, but could be .py file). Directives start with a % character. This is referred to as *ECF_MICRO* character.

The directives are used in two main context.

> - *Preprocessing directives. In this case the directive starts as the **first** character on a line in a* ecf script *file. See the table below which shows the allowable values. Only one directive is allowed on the line.*
> - *Variable directives. We use two* ECF_MICRO *characters i.e. %VAR%, in this case they can occur **anywhere** on the line and in any number.*
>
>   ```
>   %CAR% %TYPE% %WISHLIST%
>   ```
>
>   *These directives take part in* variable substitution .
>
>   *If the micro characters are not paired (i.e.. uneven) then* variable substitution *cannot take place hence an error message is issued.*
>
>   ```
>   port=%ECF_PORT       # error issued since '%' micro character are not paired.
>   ```
>
>   *However an uneven number of micro character are allowed, **If** the line begins with '#' comment charcter.*
>
>   ```
>                        # This is a comment line with a single micro character % no error issued
>   # port=%ECF_PORT again no error issued
>   ```

Directives are expanded during *pre-processing* . Examples include:

Symbol	Meaning
%include <filename>	%ECF_INCLUDE% directory is searched for the filename and the contents included into the job file. If that variable is not defined ECF_HOME is used. If the ECF_INCLUDE is defined but the file does not exist, then we look in ECF_HOME. This allows specific files to be placed in ECF_INCLUDE and the more general/common include files to be placed in ECF_HOME. This is the recommended format. releases > 4.0.8 allow ECF_INCLUDE to have multiple paths. i.e.. When ECF_INCLUDE -> path1:path2:path3 %include <filename> -> path1/filename || path2/filename || path3/filename || ECF_HOME/filename It should be noted that if one include file includes another, then includes are processed in a depth first manner.
%include "./filename" %include "../filename" %include "filename"	%include "./filename" -> script_file_location/./filename %include "../filename" -> script_file_location/../filename %include "filename" -> %ECF_HOME%/%SUITE%/%FAMILY%/filename Include the contents of the file: %ECF_HOME%/%SUITE%/%FAMILY%/filename into the job file
%include filename	Include the contents of the file filename into the output. The only form that can be used safely must start with a slash '/'
%includeonce <filename>	Same as %include, but file is **only** included once. Subsequent %includeonce of the same 'filename' are ignored and removed. Introduced in ecFlow release 4.6.0
%includenopp filename	Same as %include, but the file is not interpreted at all.
%comment	Start's a comment, which is ended by %end directive. The section enclosed by %comment - %end is removed during *pre-processing*
%manual	Start's a manual, which is ended by %end directive. The section enclosed by %manual - %end is removed during *pre-processing* . The manual directive is used to create the *manual page* show in *ecflow_ui*
%nopp	Stop pre-processing until a line starting with %end is found. No interpretation of the text will be done(i.e.. no variable substitutions)
%end	End processing of %comment or %manual or %nopp
%ecfmicro CHAR	Change the directive character, to the character given. If set in an include file the effect is retained for the rest of the job(or until set again). It should be noted that the ecfmicro directive specified in the *ecf script* file, does **not** effect the variable substitution for ECF_JOB_CMD, ECF_KILL_CMD or ECF_STATUS_CMD variables. They still use *ECF_MICRO* . If no ecfmicro directive exists, we default to using *ECF_MICRO* from the *suite definition*

From ecflow release 4.4.0, use of %VAR% (variable substitution) can be a part of the include filename. i.e.

- %include <%file%.h>   # %file% must be defined, on the task, or on the parent hierarchy
- %include %INCLUDEFILE:<file>%   # use %INCLUDEFILE% if defined (on the task, or on the parent hierarchy,
                                                             # and **MUST** follow one of format above. "./filename", "../filename", "filename", <filename>) **otherwise** use <file>

Care should be taken to avoid spaces in the variable values.

ecf file location algorithm	*ecflow_server* and job creation checking uses the following algorithm to locate the '.ecf' file corresponding to a *task* . To search for files with a different extension, i.e. to look for python file '.py'. Override ECF_EXTN variable. Its default value is '.ecf' • ECF_SCRIPT First it uses the generated variable ECF_SCRIPT to locate the script. This variable is generated from: <ECF_HOME>/<path to task>.<ECF_EXTN> Hence if the task path is /suite/f1/f2/t1, then ECF_SCRIPT=<ECF_HOME>/suite/f1/f2/t1.<ECF_EXTN> • ECF_FETCH (user variable) file is obtained from running the command after some postfix arguments are added. (Output of popen) • ECF_SCRIPT_CMD(user variable) file is obtained from running the command. (Output of popen) • ECF_FILES Second it checks for the user defined ECF_FILES variable. If defined the value of this variable must correspond to a directory. This directory is searched in reverse order(i.e. prune root) i.e.. let's assume we have a task: /o/12/fc/model and ECF_FILES is defined as: /home/ecmwf/emos/def/o/ECFfiles The ecFlow will use the following search pattern. 1. */home/ecmwf/emos/def/o/ECFfiles/o/12/fc/model.ecf* 2. */home/ecmwf/emos/def/o/ECFfiles/12/fc/model.ecf* 3. */home/ecmwf/emos/def/o/ECFfiles/fc/model.ecf* 4. */home/ecmwf/emos/def/o/ECFfiles/model.ecf* If the directory does not exist, the server will try variable substitution. This allows additional configuration. edit ECF_FILES /home/ecmwf/emos/def/o/%FILE_DIR:ECFfiles% The search can be reversed, by adding a variable ECF_FILES_LOOKUP, with a value of "prune_leaf". (from ecflow 4.12.0) Then ecFlow will use the following search pattern. 1. /home/ecmwf/emos/def/o/ECFfiles/o/12/fc/model.ecf 2. /home/ecmwf/emos/def/o/ECFfiles/o/12/model.ecf 3. /home/ecmwf/emos/def/o/ECFfiles/o/model.ecf 4. /home/ecmwf/emos/def/o/ECFfiles/model.ecf However please be aware this will **also** affect the search in ECF_HOME • ECF_HOME Thirdly it searches for the script in reverse order using ECF_HOME (i.e. like ECF_FILES) If this fails, then the *task* is placed into the *aborted* state. We can check that file can be located before loading the suites into the server. Note: The addition of variable with a name ECF_FILES_LOOKUP and value 'prune_leaf', affects the search in **BOTH** ECF_FILES and ECF_HOME • Checking job creation • *ecflow.Defs.check_job_creation*
ecf script	The ecFlow script refers to an '.ecf' file. However it could be any file, i.e. perl, python. By overriding ECF_EXTN, any ascii file is possible. The script file is transformed into the *job file* by the *job creation* process. The base name of the script file **must** match its corresponding *task* . i.e.. t1.ecf , corresponds to the task of name 't1'. The script if placed in the ECF_FILES directory, may be re-used by multiple tasks belonging to different families, providing the *task* name matches. The ecFlow script is typically similar to a UNIX shell script, with special pre-processing directives. Equally it can use any file, perl, python, java,ruby. The differences, however, includes the addition of 'c' like pre-processing *directives* and ecFlow *variable* 's. Also the script *must* include calls to the **init** and **complete** *child command* s so that the *ecflow_server* is aware when the job starts (i.e... changes state to *active*) and finishes (i.e.. changes state to *complete*)
ECF_DUMMY_TASK	This is a user variable that can be added to *task* to indicate that there is no associated *ecf script* file. If this variable is added to *suite* or *family* then all child tasks are treated as dummy. This stops the server from reporting an error during *job creation* . edit ECF_DUMMY_TASK ''
ECF_EXTN	defines the extension for the script that will be turned into a job file. This has a default value of '.ecf'. But could be any extension.This is used by the server as part of 'ecf file location algorithm'

ECF_FETCH	{{experimental}} This is used to specify a command, whose output can be used as a job script. The ecflow server will run the command with popen. Hence create care needs to be taken not to doom the server, with command that can hang. As this could severely affect servers ability to schedule jobs. `edit ECF_FETCH my_custom_cmd.sh` After variable substitution, the server will add the following. `my_custom_cmd.sh -s <task_name>.<ECF_EXTN>   # to extract the script and create the job` `my_custom_cmd.sh -i                       # to extract the includes` `my_custom_cmd.sh -m <task_name>.<ECF_EXTN>   # to extract the manual, i.e. for display in the info tab` `my_custom_cmd.sh -c <task_name>.<ECF_EXTN>   # to extract the comments` The output of running these commands (-s) is used to create the job.
ECF_HOME	This is user defined variable; it has four functions: • it is used as a prefix portion of the path of the job files created by ecFlow server; see the description of the ECF_JOB generated variable. • it is a default directory where ecFlow server looks for scripts (with file extension defined by ECF_EXTN,default is .ecf); overridden by ECF_FILES user defined variable. See the "ecf file location algorithm" entry for more detail. • it is a default directory where ecFlow server looks for include files; overridden by ECF_INCLUDE user defined variable. See the "directives" entry for more detail. • it is used as a default prefix portion of the job output path (the ECF_JOBOUT generated variable); overridden by ECF_OUT user defined variable. See descriptions of ECF_JOBOUT and ECF_OUT variables for more detail.
ECF_INCLUDE	This is a user defined variable. It is used to specify directory locations, that are used to search for include files. edit ECF_INCLUDE /home/fred/course/include # a single directory edit ECF_INCLUDE /home/fred/course/include:/home/fred/course/include2:/home/fred/course/include_me # set of directories to search
ECF_JOB	This is a generated *variable* . If defines the path name location of the job file. The variable is composed as: ECF_HOME/ECF_NAME.job<ECF_TRYNO>
ECF_JOB_CMD	This variable should point to a script that can submit the job. (i.e. to the queuing system, via, SLURM,PBS). The ecFlow server will detect abnormal termination of this command. Hence for errors in the job file, should call 'ecflow_client --abort", then exits cleanly. Otherwise server detects abnormal job termination, and abort flag is set. Which will prevent job re-queue(due to ECF_TRIES). If the job also sends an abort, zombies can be created. If ECF_JOB_CMD command fails, and the task is in a submitted state, then the task is set to the aborted state. However if the task was active or complete, then we do NOT abort the task. Instead the zombie flag is set. (since ecflow 4.17.1)
ECF_JOBOUT	This is a generated *variable* . This variable defines the path name for the job output file. The variable is composed as following. If ECF_OUT is specified: ECF_OUT/ECF_NAME.ECF_TRYNO otherwise: ECF_HOME/ECF_NAME.ECF_TRYNO

ECF_LISTS

This is the server variable. The variable specifies the path to the White list file. This file controls who has read/write access to the server via the **user** commands.

The user name can be found using linux, **id** command and is typically the login name. The file has a very simple format.

The file path specified by ECF_LISTS environment, is read by the server on start up. The contents of the white list can be modified, and reloaded by the server.

(However the path to the white-list file can NOT be modified after the server has started)

If ECF_LISTS is not set, the server will look for a file named <host>.<port>.ecf.lists (i.e.. my_host.3141.ecf.lists) in same directory where the server was started.

If the file specified by ECF_LISTS or <host>.<port>.ecf.lists, does not exist or exists but is empty, then all users will have read/write access to suites on the server.

Special care must be taken, so that user reloading the white list file does not remove write access for the administrator.

re load white list file

```
ecflow_client --help=reloadwsfile
ecflow_client --reloadwsfile
```

read write access for specific users

```
4.4.14   # this is a comment, the first non-comment line must include a version.

# These users have read and write access to the server
uid1  # user uid1,uid2,cog have read and write access to the server
uid2
cog

# Read only users
-fred  # users fred,bill and jake have read only access
-bill
-jake
```

example where all users have read access

```
4.4.14   # this is a comment, the first non-comment line must include a version.

# These users have read and write access to the server
uid1  # user uid1,uid2,cog have read and write access to the server
uid2
cog

# User with read access
-*    # all users have read access
```

From ecflow release 4.1.0, users can be restricted via node paths

```
4.4.5
fred             # has read /write access to all suites
-joe             # has read access to all suites

*  /x /y    # all users have read/write access to suites /x /y
-* /w /z    # all users have read access to suites /w /z

user1 /a,/b,/c  # user1 has read/write access to suite /a /b /c
user2 /a
user2 /b
user2 /c       # user2 has read write access to suite /a /b /c
user3 /a /b /c # user3 has read write access to suite /a /b /c

-user4 /a,/b,/c  # user4 has read access to suite /a /b /c
-user5 /a
-user5 /b
-user5 /c    # user5 has read access to suite /a /b /c
-user6 /a /b /c   # user6 has read access to suite /a /b /c
```

ECF_PASSWD	This is environment variable that point to a password file for both client and server.
	
	This enables password based authentication for ecFlow **user** commands.
	
	The password file is required for the client and server.
	
	Example client password file. The same file can be used for multiple servers
	
	`4.5.0`
	`# <user> <host> <port> <passwd>`
	`user1 machine1 3141 xxxty`
	`user1 machine2 3142 shhert`
	
	Example server password file for machine1 and port 3141
	
	`4.5.0`
	`user1 machine1 3141 xxxty`
	`user2 machine1 3141 bbsdd7`
	
	The server administrator needs to set Unix file permissions, so that this file is **only** readable by ecFlow server and the administrator.

ECF_MICRO	This is a suite and generated *variable* . The default value is %. This variable is used in *variable substitution* during command invocation and default directive character during *pre-processing* . It can be overridden, but must be replaced by a single character.

ECF_NAME	This is a generated *variable* . It defines the path name of the task. It will typically be used inside script file, referring to the corresponding task.
	
	t1.ecf
	
	`%include <head.h>`
	`....`
	`ecflow_client --alter change variable "fred" "bill" %ECF_NAME% # change variable on corresponding task`
	`...`
	`%include <tail.h>`

ECF_NO_SCRIPT

This is a user variable, that can be added to a Node.(introduced with ecFlow release 4.3.0). It is used to inform the ecflow_server that there is no **SCRIPT** associated with a task.

However unlike ECF_DUMMY_TASK, the task can still be submitted provided the ECF_JOB_CMD is set up.

This is suitable for very **lightweight** tasks that want to minimize latency. The output can still be seen, if it is redirected to ECF_JOBOUT. Care must be taken to ensure the path to ecflow_client is accessible.

ECF_NO_SCRIPT examples

```
family no_script
  edit ECF_NO_SCRIPT "1"  # the server will not look for .ecf files
  edit ECFLOW_CLIENT ecflow_client
  edit DIROUT %VERBOSE%
  edit SILENT ""
  edit VERBOSE " > %ECF_JOBOUT 2>&1"

task non_script_task
   edit ECF_JOB_CMD "export ECF_PASS=%ECF_PASS%;export ECF_PORT=%ECF_PORT%;export ECF_HOST=%ECF_HOST%;export ECF_NAME=%ECF_NAME%;export ECF_TRYNO=%ECF_TRYNO%; %ECF_CLIENT% --init=$$; echo 'test test_ecf_no_script' %DIROUT% && %ECF_CLIENT% --complete"
   # this command is not expected to fail. hence no error handling.(i.e.. will stay active)

 task ecf_no_script
  edit ECF_JOB_CMD "ecf_no_script --pass %ECF_PASS% --host %ECF_HOST% --port %ECF_PORT% " # %DIROUT%
  # ecf_no_script contains init, complete, call to ecflow_client and trapping to raise abort
  # use this approach for robust error handling

 task ymd2jul
  edit ECF_JOB_CMD "ECF_PASS=%ECF_PASS% ECF_NAME=%ECF_NAME% /usr/local/bin/ymd2jul.sh -p %ECF_PORT% -n %ECF_HOST% -r /%SUITE%/%FAMILY% -y %YMD% > %ECF_JOBOUT% 2>&1 &"
  # /usr/local/bin/ymd2jul.sh can be called on command line or as ecflow_client
endfamily
```

ECF_PASS

This is a generated *variable* . During job generation process in the server, a unique password is generated and stored in the task. It then replaces %ECF_PASS% in the scripts(.ecf), with the actual value. When the job runs, ecflow_client reads this, as an environment variable, and passes it to the server. The server then compares this password with the one held on the task. This is used as a part of the authentication for child commands, and is used to detect zombies.

The authentication process can be bypassed, and allow the job to proceed (i.e.. when the user is **sure** that there is only a single process, trying to communicate with the server), by adding it as a user variable. i.e..

    ecflow_client --alter add variable ECF_PASS FREE <path to task>

This functionality is also available in the GUI. Select a task. RMB->Special->Free password.

However it is important not leave this in place, as it will always bypass the authentication. Just delete the variable.

ECF_SCRIPT

This is a generated *variable* . If defines the path name for the *ecf script*

ECF_SCRIPT_CMD

[[experimental]]

This allows the output of running a command to be treated as a script. The command is run after variable substitution. The output is obtained from running the system function **popen** in the server. Great care should be taken when running this command, to ensure errors in the command do not crash the server. This approach could be used for short lived tasks, where extremely low latency is required. Commands that take more than 20s can interfere with job scheduling and should be avoided. Could possibly be used to checkout a script from a version control system.

If the output contains %include,%manual,%noop they are treated in the same manner as a normal '.ecf' script.

Here the output of the 'cat' command is treated as a script

```
suite test
   family family
      task check
         edit ECF_SCRIPT_CMD "cat /tmp/ECF_SCRIPT_CMD/family/check.ecf"
      task t1
         trigger check == complete
         edit ECF_SCRIPT_CMD "cat /tmp/ECF_SCRIPT_CMD/family/t1.ecf"
   endfamily
endsuite
```

ECF_TRIES

This is generated variable added at the server level with a default value of 2. It can be overridden by the user and controls the number of times job should re-run should it abort. Provided:

- the task/job has NOT been killed(user action)
- The job process(created from .ecf or .py) exited cleanly and not with exit 1 || sys.exit(1) as process death is captured by the server. Always ensure your script exits cleanly. i.e. exit(0)
- the task has NOT been set to abort by the user(user action)
- job creation has not failed . i.e. task pre-processing(include file expansion,variable substitution, change of file permission for job file)
- the value of the variable ECF_TRIES must be convertible to an integer.

Please note this allows your scripts to be self-aware of the number times it is being run. i.e.

task.ecf

```
%include <head.h>
"echo do some work\n";
if [ %ECF_TRYNO% -eq 1 ] ; then
    echo "first attempt"
    .....
fi
if [ %ECF_TRYNO% -eq 2 ] ; then
    echo "first attempt failed, trying a different approach, clean data, etc"
    .....
fi
%include <tail.h>
```

ECF_TRYNO

This is a generated *variable* that is used in file name generation. It represents the current try number for the *task* .

It can also be referenced inside .ecf script, to allow the job to take a different course dependent on the ECF_TRYNO.

After **begin** it is set to 1. The number is advanced if the job is re-run. It is re-set back to 1 after a **re-queue**.

It is used in output and *job file* numbering. (i.e.. It avoids overwriting the *job file* output during multiple re-runs)

ECF_OUT

This is user/suite variable that specifies a directory PATH. It controls the location of job output(stdout and stderr of the process) on a **remote** file system.

It provides an alternate location for the job and cmd output files. If it exists, it is used as a base for ECF_JOBOUT, but it is also used to search for the output by ecFlow, when asked by *ecflow_ui* /CLI.

If the output is in ECF_OUT/ECF_NAME.ECF_TRYNO it is returned, otherwise ECF_HOME/ECF_NAME.ECF_TRYNO is used.

The user **must ensure** that all the directories exists, including suite/family. If this is not done, you may well find task remains **stuck** in a submitted state.

At ECMWF our submission scripts will ensure that directories exists.

ecFlow

Is the ECMWF work flow manager.

A general purpose application designed to schedule a **large** number of computer process in a heterogeneous environment.

Helps computer jobs design, submission and monitoring both in the research and operation departments.

ecflow_client

This executable is a command line program; it is used for **all** communication with the *ecflow_server* .

To see the full range of commands that can be sent to the *ecflow_server* type the following in a UNIX shell:

> *ecflow_client –help*

This functionality is also provided by python *Client Server API* .

The following variables affect the execution of ecflow_client.

Since the *ecf script* can call ecflow_client(i.e.. *child command*) then typically some are set in an include header. i.e.. *head.h* .

Environment Variable common for user and child commands

Variable Name	Explanation	Compulsory	Example
ECF_PORT	Port number of the *ecflow_server*	Yes/No	We can use: ecflow_client --port 3141 , As an alternative to specifying the ECF_PORT. Must match the port of the server
ECF_HOST	Name of the host running the *ecflow_server*	Yes/No	We can use: ecflow --host machine1, As an alternative to specifying ECF_HOST
NO_ECF	If set exits ecflow_client immediately with success. This allows the scripts to be tested independent of the server	No	export NO_ECF=1
ECF_DENIED	If server denies client communication and this flag is set, exit with an error. Avoids 24hr hour connection attempt to *ecflow_server* .	No	export ECF_DENIED=1
ECF_SSL	For secure socket communication with server. Requires client/server built with openssl libs	No	export ECF_SSL=1 # Use same certificate for multiple server export ECF_SSL=<host>.<port> # Use server specific certificates. Alternatively to avoid setting environmental variables we can use: ecflow_client --ssl ... The client will first look for $HOME/.ecflowrc/ssl/server.crt **then** $HOME/.ecflowrc/ssl/<host>.<port>.crt

Environment Variables for child commands

Variable Name	Explanation	Compulsory	Example
ECF_NAME	Path to the task	Yes	/suite/family/task
ECF_PASS	Jobs password. Generated by the server, will replace %ECF_PASS% in the scripts,during job generation. Used for authenticating child commands.	Yes	(generated)
ECF_RID	Remote id. Allow easier job kill, and disambiguate a zombie from the real job.	Yes	(generated)
ECF_TRYNO	The number of times the job has run. This is allocated by the server and used in job/output file name generation.	No	(generated)
ECF_HOSTFILE	File that lists alternate hosts to try, if connection to main host fails	No	/home/user/avi/.ecfhostfile
ECF_TIMEOUT	Maximum time in seconds for the child(init,abort,complete,etc) client to deliver message to the server	No	24*3600 (default value), export ECF_TIMEOUT=360
ECF_ZOMBIE_TIMEOUT	Maximum time in seconds for the child(init,abort,complete,etc) **zombie** client to get a reply from the server.	No	12*3600 (default value) export ECF_ZOMBIE_TIMEOUT=360

Variables specific to User commands

Variable Name	Explanation	Compulsory	Example
ECF_PASSWD	path to the client password file, used for password based authentication.	No	export ECF_PASSWD=polonius.3141.ecf.passwd
ECF_USER	When user need to pose as another user. i.e. when users id on the client machine, doesn't match his id on the remote server. Requires password file	No	export ECF_USER=fred To avoid setting environment variable we can use: ecflow_client --user fred ......

ecflow_server

This executable is the server.

It is responsible for *scheduling* the jobs and responding to *ecflow_client* requests

Multiple servers can be run on the same machine/host providing they are assigned a unique port number.

The server record's all request's in the log file.

The server will periodically(See ECF_CHECKINTERVAL) write out a *check point* file.

The following environment variables control the execution of the server and may be set before the start of the server. ecflow_server will start happily without any of these variables being set, since all of them have default values.

Variable Name	Explanation	Default value
ECF_HOME	Home for all the *ecFlow* files	Current working directory
ECF_PORT	Server port number. Must be unique	3141
ECF_LOG	History or log file	<host>.<port>.ecf.log
ECF_CHECK	Name of the checkpoint file	<host>.<port>.ecf.check
ECF_CHECKOLD	Name of the backup checkpoint file	<host>.<port>.ecf.check.b
ECF_CHECKINTERVAL	Interval in second to save *check point* file	120
ECF_LISTS	White list file. Controls read/write access to the server for each user	<host>.<port>.ecf.lists
ECF_TASK_THRESHOLD	Report in log file all task/job that take longer than given threshold. Used to debug/instrument, those scripts that are very large.	4000ms, (release 4.0.6 default was 2000ms), where 1000ms = 1 second
ECF_PASSWD	path to server password file, used to authenticate user commands. Use when **ALL** should be password authenticated	<host>.<port>.ecf.passwd
ECF_CUSTOM_PASSWD	path to server password file, used to authenticate user commands. Use when a **small** number of users need to be password authenticated. Typically client would use: • ecflow_client --user=fred .... • export ECF_USER=fred; ecflow_client ...	<host>.<port>.ecf.custom_passwd
ECF_PRUNE_NODE_LOG	When the checkpoint point file is loaded, node log history older than 30 days is automatically pruned. The variable allows this value to be changed. Setting the variable to zero, means there will be no pruning. All history is preserved at the cost increasing server memory, and time taken to write checkpoint file.	export ECF_PRUNE_NODE_LOG=40 Prune node log history older than 40 days, upon reload of checkpoint file.
ECF_SSL	For secure socket communication with client. Requires client/server built with openssl libs	export ECF_SSL=1 #Use same certificate for multiple server export ECF_SSL=<host>.<port> # Use server specific certificates. Alternatively to avoid setting environmental variables we can use: ecflow_server --ssl ... || ecflow_start.sh -s The server will then first look for $HOME/.ecflowrc/ssl/server.crt **then** $HOME/.ecflowrc/ssl/<host>.<port>.crt

The server can be in several states. The default when first started is *halted* , See *server states*

ecflow_ui

ecflow_ui executable in the new GUI based client. It is used to visualise and monitor the hierarchical structure of the *suite definition* .

ecflowview

ecflowview executable is the GUI based client, that is used to visualise and monitor the hierarchical structure of the *suite definition.*

(**this is deprecated for ecflow 5 series**)

> *state changes in the* node *'s and the* ecflow_server *, using colour coding*
>
> *Attributes of the nodes and any* dependencies
>
> ecf script *file and the corresponding* job file

event

The purpose of an event is to signal partial completion of a *task* and to be able to trigger another job which is waiting for this partial completion.

There can be many events and they are displayed as nodes.

The event is updated by placing the –event *child command* in a *ecf script* . Events on family nodes can be set using 'ecflow_client --alter' command.

An event has a number and possibly a name. If it is only defined as a number, its name is the text representation of the number without leading zeroes.

For python see: ecflow.Event and ecflow.Node.add_event For text BNF see event

If the event *child command* s, results in a *zombie* , then the default action if for the server to **fob**, this allows the ecflow_client command to exit normally. (i.e. without any errors).

This default can be overridden by using a zombie attribute.

Events can be referenced in *trigger* and *complete expression* s.

extern	This allows an external *node* to be used in a *trigger* expression. All *node* ‘s in *trigger* ‘s must be known to *ecflow_server* by the end of the load command. No cross-suite *dependencies* are allowed unless the names of tasks outside the suite are declared as external. An external *trigger* reference is considered unknown if it is not defined when the *trigger* is evaluated. You are strongly advised to avoid cross-suite *dependencies* . Families and suites that depend on one another should be placed in a single *suite* . If you think you need cross-suite dependencies, you should consider merging the suites together and have each as a top-level family in the merged suite. For BNF see extern
family	A family is an organisational entity that is used to provide hierarchy and grouping. It consists of a collection of *task* ‘s and families. Typically you place tasks that are related to each other inside the same family, analogous to the way you create directories to contain related files. For python see ecflow.Family . For BNF see family It serves as an intermediate *node* in a *suite definition* .
halted	Is a *ecflow_server* state. See *server states*
hybrid clock	A hybrid *clock* is a complex notion: the date and time are not connected. The date has a fixed value during the complete execution of the *suite* . This will be mainly used in cases where the suite does not *complete* in less than 24 hours. This guarantees that all tasks of this suite are using the same *date* . On the other hand, the time follows the time of the machine. Hence the *date* never changes unless specifically altered or unless the suite restarts, either automatically or from a begin command. Under a hybrid *clock* any *node* held by a *date* , *day* or cron dependency will be set to complete at the beginning of the suite. (i.e.. without its job ever running). Otherwise the *suite* would never *complete* .

inlimit

The inlimit works in conjunction with *limit* / ecflow.Limit for providing simple load management

inlimit is added to the *node* that needs to be limited.

Limiting tasks, only allow 5 tasks to run in parallel

```
suite suite
    limit disk 100
    family anon
        inlimit /suite:disk 5
        task t1
        ...
        task t100
    endfamily
endsuite
```

Limiting Families, only two families can run in parallel. The tasks are unconstrained

```
suite test
 limit fam 2
 family f1
     inlimit -n fam
     task t1
     ....
 endfamily
 family f2
     inlimit -n fam
     task t1
     ....
 endfamily
 family f3
     inlimit -n fam
     task t1
     ....
 endfamily
endsuite
```

Limit submission.

```
# Hence we could have more than 2 active jobs, since we are only control the number in the submitted state.
# If we removed the -s then we can only have two active jobs running at one time
suite test_limit_on_submission
    limit disk 2
    family anon
        inlimit -s disk   # Inlimit submission
        task t1
        task t2
        ....
    endfamily
endsuite
```

For python see ecflow.InLimit and ecflow.Node.add_inlimit . For text BNF see *inlimit*

job creation	Job creation or task invocation can be initiated manually via *ecflow_ui* but also by the *ecflow_server* during *scheduling* when a *task* (and *all* of its parent *node* s) is free of its *dependencies* . The process of job creation includes: > *o Generating a unique password ECF_PASS, which is placed in* ecf script *during* pre-processing *. See* head.h > > *o Locating* ecf script *files , corresponding to the* task *in the* suite definition *, See* ecf file location algorithm > > *o* pre-processing *the contents of the* ecf script *file* The steps above transforms an *ecf script* to a *job file* that can be submitted by performing *variable substitution* on the ECF_JOB_CMD *variable* and invoking the command. The running jobs will communicate back to the *ecflow_server* by calling *child command* 's and user commands. This causes *status* changes on the *node* 's in the *ecflow_server* and flags can be set to indicate various events. If a *task* is to be treated as a dummy task(i.e.. is used as a scheduling task) and is not meant to be run, then a variable of name *ECF_DUMMY_TASK* can be added. `task.add_variable("ECF_DUMMY_TASK", "")`
job file	The job file is created by the *ecflow_server* during *job creation* using the *ECF_TRYNO variable* It is derived from the *ecf script* after expanding the pre-processing *directives* . It has the form <task name>.job< *ECF_TRYNO* >, i.e.. t1.job1. Note job creation checking will create a job file with an extension with zero. i.e.. '.job0'. See ecflow.Defs.check_job_creation When the job is run the output file has the *ECF_TRYNO* as the extension. i.e.. t1.1 where 't1' represents the task name and '1' the *ECF_TRYNO*
label	A label has a name and a value and is a way of **displaying** information in *ecflow_ui* By placing a label *child command* s in the *ecf script* the user can be informed about progress in *ecflow_ui* . Labels can be added to family nodes. To change the labels, scripts should use: 'ecflow_client --alter change label <label_name> <new_value> /path/to/family_node/with/label If the label *child command* s, results in a zombie then the default action if for the server to **fob**, this allows the ecflow_client command to exit normally. (i.e. without any errors). This default can be overridden by using a zombie attribute. For python see ecflow.Label and ecflow.Node.add_label . For text BNF see label

late

Define a tag for a node to be late. A node can only have **one** late attribute. The late attribute only applies to a task. You can define it on a Suite/Family in which case it will be inherited. Any late defined lower down the hierarchy will override the aspect(submitted,active, complete) defined higher up.

- -s submitted: The time node can stay submitted (format [+]hh:mm). submitted is always relative, so + is simple ignored, if present. If the node stays submitted longer than the time specified, the late flag is set
- -a Active   : The time of day the node must have become active (format hh:mm). If the node is still queued or submitted, the late flag is set
- -c Complete : The time node must become complete (format {+}hh:mm). If relative, time is taken from the time the node became active, otherwise node must be complete by the time given.

```
suite late
   family familyName
      task t1
              late -s +00:15 -a 20:00 -c +02:00
      task t2
              late -a 20:00 -c +02:00 -s +00:15
      task t3
              late -c +02:00 -a 20:00  -s +00:15
      task t4
              late  -s 00:02 -c +00:05
      task t5
              late  -s 00:01 -a 14:30 -c +00:01
    endfamily
endsuite
```

Suites and families cannot be late, but you can define a late tag for submitted in a suite, to be inherited by the families and tasks. When a node is classified as being late, the only action *ecflow_server* takes is to set a flag. *ecflow_ui* will display these alongside the *node* name as an icon (and optionally pop up a window).

```
suite late
   late -s +00:15    # report late for all task taking longer than 15 minutes in submitted state
   family familyName
      late -c +02:00 # all child task that take longer than 2 hours to complete should raise a late flag
      task t1
              # effective late -s +00:05 -c +02:00
              late -s +00:05
      task t2
              # effective late  -s +00:15 -c +02:00
      task t5
              # effective late  -c +03:00 -a 18:00 -s +00:15
              late -c +03:00 -a 18:00
    endfamily
endsuite
```

The late attribute can be added/deleted to any suite/family/task.

```
ecflow_client --alter add    late "-s 00:15"                    <path-to-node>
ecflow_client --alter change late "-s 00:01 -a 14:30 -c +00:01" <path-to-node>
ecflow_client --alter delete late                               <path-to-node>
```

For python see ecflow.Late and ecflow.Node.add_late . For text BNF see late

limit

Limits provide simple load management by limiting the number of tasks submitted by a specific *ecflow_server* .

Typically you either define limits on *suite* level or define a separate suite to hold limits so that they can be used by multiple suites.

Setting limits on a separate suite, has the benefit that by setting the limit value to zero, you can control task submission over a number of suites.

Limits

```
suite suiteName
    limit sg1  10
    limit mars 10
endsuite
```

The limits are used in conjunction with *inlimit*

The limit max value can be changed on the command line

```
>ecflow_client --alter change limit_max <limit-name> <new-limit-value> <path-to-limit>
>ecflow_client --alter change limit_max limit 2 /suite
```

It can also be changed in python:

```
#!/usr/bin/env python2.7
import ecflow
try:
   ci = ecflow.Client()
   ci.alter("/suite","change","limit_max","limit", "2")
except RuntimeError, e:
   print "Failed: " + str(e)
```

For python see ecflow.Limit and ecflow.Node.add_limit . For BNF see limit and *inlimit*

manual page

Manual pages are part of the *ecf script* .

This is to ensure that the manual page is updated when the *ecf script* is updated. The manual page is a very important operational tool allowing you to view a description of a task, and possibly describing solutions to common problems. The *pre-processing* can be used to extract the manual page from the script file and is visible in *ecflow_ui* . The manual page is the text contained within the %manual and %end *directives* . They can be seen using the manual button on *ecflow_ui* .

The text in the manual page in **not** included in the *job file* .

There can be multiple manual sections in the same *ecf script* file. When viewed they are simply concatenated. It is good practice to modify the manual pages when the script changes.

The manual page may have the %include *directives.*

Suite and families may also have a manual page. These will also be available in the GUI. Ecflow will look for a file <node_name>.man (where node_name is the name of suite or family) using a backwards search algorithm first in ECF_FILES directory, then ECF_HOME directory. Note that errors in variable pre-processing are ignored inside of a manual section. It should also be noted that for family and suite manuals, the %manual and %end directives are not strictly necessary, as the whole file is treated as a manual.

If we have family: /suite/big/f1, ecflow will search for "f1.man" in:

```
<ECF_FILES>/suite/big/f1.man
<ECF_FILES>/suite/f1.man
<ECF_FILES>/f1.man
<ECF_HOME>/suite/big/f1.man
<ECF_HOME>/suite/f1.man
<ECF_HOME>/f1.man
```

meter

The purpose of a meter is to signal proportional completion of a task and to be able to trigger another job which is waiting on this proportional completion.

The meter is updated by placing the –meter *child command* in a *ecf script* .

Meters can be added to family nodes. To change the meters, in the scripts should use: ecflow_client --alter change meter <meter_name> <new_value> /path/to/family_node/with/meter

For python see: ecflow.Meter and ecflow.Node.add_meter . For text BNF see meter

If the meter *child command* s, results in a zombie, then the default action if for the server to **fob** , this allows the ecflow_client command to exit normally. (i.e. without any errors). This default can be overridden by using a zombie attribute.

Meter's can be referenced in *trigger* and *complete expression* expressions.

node

suite , *family* and *task* form a hierarchy. Where a *suite* serves as the root of the hierarchy. The *family* provides the intermediate nodes, and the *task* provide the leaf's.

Collectively *suite* , *family* and *task* can be referred to as nodes.

For python see ecflow.Node .

pre-processing

Pre-processing takes place during *job creation* and acts on *directives* specified in *ecf script* file.

This involves:

> *o expanding any include file* directives *. i.e.. similar to 'c' language pre-processing*
>
> *o removing comments and manual* directives
>
> *o performing* variable substitution

queued

Is a *node status* .

After the begin command, the task **without** a *defstatus* are placed into the queued state

real clock

A *suite* using a real *clock* will have its *clock* matching the clock of the machine. Hence the *date* advances by one day at midnight.

repeat

Repeats provide looping functionality. There can only be a single repeat on a *node* .

> *repeat day step # only for suites*
>
> *repeat integer VARIABLE start end [step]*
>
> *repeat enumerated VARIABLE first [second [third ...]]*
>
> *repeat string VARIABLE str1 [str2 ...]*
>
> *repeat file VARIABLE filename*
>
> *repeat date VARIABLE yyyymmdd yyyymmdd [delta]*
>
> *repeat datelist VARIABLE yyyymmdd yyyymmdd .....*

The repeat variable name is available as a generated variable.

The **repeat date** defines additional generated variables(from ecflow 4.7.0) , which are scoped with prefix of the variable name i.e.

- <variable>   # the default, the value is the current date
- <variable>_YYYY   # The year
- <variable>_MM   # the month
- <variable>_DD   # The day of the month
- <variable>_DOW   # day of the week
- <variable>_JULIAN   # the julian value for the date

For example:

repeat date generated variables, accessible for trigger expressions

```
repeat date YMD 20090101 20220101
# The following generated variables, are accessible for trigger expressions
# YMD, YMD_YYYY, YMD_MM, YMD_DD, YMD_DOW,YMD_JULIAN
```

As the repeat variable changes so do the generated variables. (See the tutorial for an example. Repeat)

If a repeat is added to a family/suite, then the repeat will **ONLY** loop(and automatically re-queue its children) if **all** the children are complete.

Hence additional care needs to be taken. i.e.. if the parent node has a repeat and the child has a cron attribute then the cron will always force a re-queue on the node once it has run, and hence will stop the parent from looping.

If we use relative time attribute. i.e. time +02:00, under a repeat, then the time is relative to the repeat re-queue.

The repeat VARIABLE can be used in *trigger* and *complete expression* expressions. Depending on the kind of repeat the value can vary:

```
 RepeatDate       -> value
 RepeatDateList   -> value
 RepeatString     -> index  ( will always return a index)
 RepeatInteger    -> value
 RepeatEnumerated -> value | index  ( return value at index if cast-able to integer, otherwise return index )
 RepeatDay        -> value
```

If a "repeat date" or "repeat datelist" VARIABLE is used in a trigger expression then date arithmetic is used, when the expression uses addition and subtraction. i.e..

```
defs = ecflow.Defs()
s1 = defs.add_suite("s1");
t1 = s1.add_task("t1").add_repeat( ecflow.RepeatDate("YMD",20090101,20091231,1) );
t2 = s1.add_task("t2").add_trigger("t1:YMD - 1 eq 20081231");
assert t2.evaluate_trigger(), "Expected trigger to evaluate. 20090101 - 1  == 20081231"
```

```
When we use relative time attributes under a Repeat. They are automatically reset when the repeat loops. Take for example:
```

```
suite s1
  family hc00
    repeat integer HYEAR 1993 2017
    time +00:01                    # when the repeat loops delay starting task a, for 1 minute
    task a
    task b
      trigger a  == complete
  endfamily
endsuite
```

```
Now when task 'a' and Task 'b' complete, the repeat is incremented, and any relative time attributes are reset.
In this case effectively delaying the starting of task 'a' for 1 minute.

For python see  ecflow.Node.add_repeat ,  ecflow.Repeat ,  ecflow.RepeatDate ,  ecflow.RepeatEnumerated  repeat
```

running

Is a *ecflow_server* state. See *server states*

scheduling

The *ecflow_server* is responsible for *task* scheduling.

It will check *dependencies* in the *suite definition* every minute. If these *dependencies* are free, the *ecflow_server* will submit the task. See *job creation* .

server states

The following tables reflects the *ecflow_server* capabilities in the different states

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

shutdown

Is a *ecflow_server* state. See *server states*

status

Each *node* in *suite definition* has a status.

Status reflects the state of the *node* . In *ecflow_ui* the background colour of the text reflects the status.

task status are: *unknown* , *queued* , *submitted* , *active* , *complete* , *aborted* and *suspended*

ecflow_server status are: *shutdown* , *halted* , *running* this is shown on the root node in *ecflow_ui*

submitted

Is a *node status* .

When the *task dependencies* are resolved/free the *ecflow_server* places the task into a submitted state. However if the ECF_JOB_CMD fails, the task is placed into the *aborted* state

suite

A suite is organisational entity. It is serves as the root *node* in a *suite definition* .

It should be used to hold a set of jobs that achieve a common function. It can be used to hold user *variable* s that are common to all of its children.

Only a suite node can have a *clock* .

Suite generated variables:

- SUITE   The name of the suite
- ECF_TIME   23:30 the current suite time
- TIME   2330 time as integer, Can be used in a trigger expression, ideally using <=, <, >=, >
- YYYY   The year as an integer
- DOW   Day of the week, as an integer. Sunday=0,Monday=1,etc
- DOY   Day of the year, as an integer
- DAY   The days as a string, i.e. monday
- DD   Day of the month as an integer.
- MM   The month as an integer
- MONTH   as a string
- ECF_DATE   YYYYMMDD   year,month,day of the month as 8 digit integer
- ECF_JULIAN The julian value of the current date(added in ecflow 4.7.0)
- ECF_CLOCK <day>:<month>:<day of week>:<day of year>. i.e. Tuesday:December:2:348

It is a collection of *family* 's, *variable* 's, *repeat* and a single *clock* definition. For a complete list of attributes look at BNF for suite . For python see ecflow.Suite .

suite definition

The suite definition is the hierarchical *node* tree.

It describes how your *task* 's run and interact.

It can built up using:

> - *Ascii text file by following the rules defined in the ecFlow* Definition file Grammar .
>
>   *Hence any language can be used, to generate this format.*
> - Suite Definition API

Once the definition is built, it can be loaded into the *ecflow_server* (either via command line, or with the python api) and started. It can be monitored by *ecflow_ui*

suspended

Is a *node* state. A *node* can be placed into the suspended state via a *defstatus* or via *ecflow_ui*

A suspended *node* including any of its children cannot take part in *scheduling* until the node is resumed.

task	A task represents a job that needs to be carried out. It serves as a leaf *node* in a *suite definition* Only tasks can be submitted. A job inside a task *ecf script* should generally be re-entrant so that no harm is done by rerunning it, since a task may be automatically submitted more than once if it aborts. For python see ecflow.Task . For text BNF see task

time dependencies	This includes, time,today, day date, cron.
	
	When we have multiple time dependencies on the same task, then time dependency of the same type are **or'ed** together, and **and'ed** with the different types.
	
	This task will run on the 17th of February 2017 at 10am
	
	`task xx`
	`   time 10:00`
	`   date 17.2.2017`
	
	Run task xx. at 10am and 8pm, on the 17th and 19th of February 2017, that is four times in all. Notice the task is queued in between and completes only after the last run
	
	`task xx`
	`   time 10:00`
	`   time 20:00`
	`   date 17.2.2017`
	`   date 19.2.2017`

time	This defines a time dependency for a node.
	
	Time is expressed in the format [h]h:mm. Only numeric values are allowed.
	
	There can be multiple time dependencies for a node, but overlapping times may cause unexpected results.
	
	The task is free to run when the time is 10:00 or 11:00
	
	`task t`
	`   time 10:00`
	`   time 11:00`
	
	To define a series of times, specify the start time, end time and a time increment.
	
	If the start time begins with '+', times are relative to the beginning of the suite **or,** in repeated families, relative to the beginning/re-queue of the repeated family.
	
	If the time the job takes to complete is longer than the interval a time 'slot' is missed, e.g.
	
	`    time 10:00 20:00 01:00`
	
	if the 10:00 run takes more than an hour, the 11:00 run will never occur.
	
	For python see ecflow.Time and ecflow.Node.add_time . For BNF see time

today	Like *time* , but If the suites begin time is **past** the time given for the "today" , then the *node* is free to run (as far as the time dependency is concerned).
	
	For example:
	
	`task x`
	`   today 10:00`
	
	If we begin or re-queue the *suite* at 9.00 am, then the *task* in held until 10.00 am. However if we begin or re-queue the suite at 11.00am, the *task* is run immediately.
	
	Now lets look at time:
	
	`task x`
	`   time 10:00`
	
	If we begin or re-queue the *suite* at 9.00am, then the *task* in held until 10.00 am. If we begin or re-queue the *suite* at 11.00am, the *task* is still held.
	
	If the time the job takes to complete is longer than the interval a 'slot' is missed, e.g.
	
	`    today 10:00 20:00 01:00`
	
	if the 10:00 run takes more than an hour, the 11:00 run will never occur.
	
	For python see ecflow.Today . For text BNF see today

trigger	Triggers defines a dependency for a *task* or *family* . There can be only one trigger dependency per *node* , but that can be a complex boolean expression of the *status* of several nodes. Triggers cannot be added to the suite node. A node with a trigger can only be activated when its trigger has expired. A trigger holds the node as long as the trigger's expression evaluation returns false. Trigger evaluation occurs when ever the *child command* communicates with the server. i.e.. whenever there is a state change in the suite definition and at least once every 60 seconds

The keywords in trigger expressions are: *unknown* , *suspended* , *complete* , *queued* , *submitted* , *active* , *aborted* and **clear** and **set** for *event* status.

Triggers can also reference Node attributes like *event* , *meter* , *variable* , *repeat* and generated variables and limits. Triggers can also reference the late, zombie and archived flag on a node. Trigger evaluation for node attributes uses integer arithmetic:

- *event* has the integer value of 0(clear) and set(1)
- *meter* values are integers hence they are used as is
- *variable* value is converted to an integer, otherwise 0 is used. This can include the **generated variables** and **suite time variables.** See example below
- *repeat string* : We use the index values as integers. See example below
- *repeat enumerated* : If the value at the index is convertible to a integer it is used, otherwise we use the index values as integers. See example below
- *repeat integer* : Use the implicit integer values
- *repeat date* : Use the date values as integers. Use of plus/minus on repeat date variable uses date arithmetic
- *repeat datelist* : Use the date values as integers. allows for an arbitrary date list
- limit : the limit value is used as an integer. This allows a degree of prioritisation amongst tasks under a limit
- *late* : The value is stored in a flag, and is a simple boolean. Used to signify when a task is late.

Here are some examples:

Trigger examples

```
suite suite
   limit top_level_limit 20
   task a
      event EVENT
      meter METER 1 100 50
      edit  VAR_DATE 20170701
      edit  VAR_STRING "captain scarlett"      # This is not convertible to an integer, if referenced will
use '0'
      late -c +02:00                           # add late flag if task takes longer than 2 hours to complete
   family f1
      edit SLEEP 2
      repeat string NAME a b c d e f               # This has values: a(0),b(1), c(3), d(4), e(5), f(6) i.e..
index
      family f2
         repeat integer VALUE 5 10                 # This has values: 5,6,7,8,9,10
         family f3
            repeat enumerated ENUM_VAR red green blue   # red(0), green(1), blue(2)
            task t1
               repeat date DATE 19991230 20000102 # This has values: 19991230,19991231,20000101,20000102
               # Here :VALUE, :NAME, :SLEEP will match with the first event,meter,user variable,repeat
variable or generated variable, up the parent hierarchy
               trigger :VALUE == 5 and :NAME == 0 and :SLEEP == 2 # references f2:VALUE,f1:NAME,f1:SLEEP new
for 4.7.0 release
         endfamily
      endfamily
   endfamily
   family f2
      inlimit /suite:top_level_limit
      task event_meter
          trigger /suite/a:EVENT == set and /suite/a:METER >= 30
      task variable
          trigger /suite/a:VAR_DATE >= 20170801 and /suite/a:VAR_STRING == 0
      task repeat_string
          trigger /suite/f1:NAME >= 4
      task repeat_integer
          trigger /suite/f1/f2:VALUE >= 7
      task repeat_enumerated
          trigger /suite/f1/f2/f3:ENUM_VAR >= 1
      task repeat_date
          trigger /suite/f1/f2/f3/t1:DATE >= 19991231
      task repeat_date_arithmitic
          # Using plus/minus on a repeat DATE will use date arithmetic
          # Since the starting value of DATE is 19991230, this task will run
          # straight away
          trigger /suite/f1/f2/f3/t1:DATE - 1 == 19991229
      task use_repeat_date_yyyy
          trigger /suite/f1/f2/f3/t1:DATE_YYYY == 2000    # DATE_YYYY(year)is a generated variable for
repeat date DATE 19991230 20000102
      task use_repeat_date_generated_mm
          trigger /suite/f1/f2/f3/t1:DATE_MM == 2         # DATE_MM(month) is a generated variable for
repeat date DATE 19991230 20000102
      task use_repeat_date_generated_dd
          trigger /suite/f1/f2/f3/t1:DATE_DD == 30        # DATE_DD(day of the month) is a generated
variable for repeat date DATE 19991230 20000102
      task use_repeat_date_generated_dow
          trigger /suite/f1/f2/f3/t1:DATE_DOW == 0        # DATE_DOW(day of week, 0-sunday,1-monday,etc) is
```

```
a generated variable for repeat date DATE 19991230 20000102
      task use_repeat_date_generated_julian
          trigger /suite/f1/f2/f3/t1:DATE_JULIAN > cal::date_to_julian(/suite/a:VAR_DATE)  # DATE_JULIAN(the
julian of the date) is a generated variable for repeat date DATE 19991230 20000102
      task with_trigger_that_ref_a_limit
          trigger /suite:top_level_limit < 5        # low priority task, only valid when system is not loaded
      task trigger_with_ref_to_late_flag
          trigger /suite/a<flag>late                # Only triggers if task /suite/a is late
      task trigger_with_ref_to_zombie_flag
          trigger /suite/a<flag>zombie              # Only triggers if task /suite/a is a zombie
      task trigger_with_ref_to_archived_flag
          trigger /suite/f1<flag>archived           # Only triggers if family /suite/f1 is archived -> only
family/suite can be archived
   endfamily
   family time_trigger
      trigger /suite:DOW == 0  or /suite:DOW == 1   # DOW is a generated variable on the suite representing
DAY of the week. i.e. Sundày and Monday in this case
      task with_time
         trigger /suite:TIME > 1330                 # TIME is a generated variable on the suite , same as
time > 13:30
   endfamily
 endsuite
```

What happens when we have multiple node attributes of the same name, referenced in trigger expressions ?

Trigger priority when name clashes

```
task foo
   event  blah
   meter  blah 0 200 50
   edit   blah 10
   repeat enumerated blah red green blue
task bar
   trigger foo:blah >= 0      # which 'blah' do we reference ?
```

In this case ecFlow will use the following precedence:

- *event*
- *meter*
- *variable*
- *repeat*
- generated variables
- limits

Hence in the example above expression ‘foo:blah >= 0’ will reference the event.

For python see ecflow.Expression and ecflow.Node.add_trigger

unknown	Is a *node status* . This is the default *node status* when a *suite definition* is loaded into the *ecflow_server*
user commands	User commands are any client to server requests that are **not** *child command* s.
variable	ecFlow makes heavy use of different kinds of variables.There are several kinds of variables: *Environment variables: which are set in the UNIX shell before the* ecFlow *starts. These control* ecflow_server *, and* ecflow_client *.* *suite definition variables: Also referred to as user variables. These control* ecflow_server *, and* ecflow_client *and are available for use in* job file *.* *Generated variables: These are generated within the* suite definition *node tree during* job creation *and are available for use in the* job file *.* Variables can be referenced in *trigger* and *complete expression* s . The value part of the variable should be convertible to an integer otherwise a default value of 0 is used. For python see ecflow.Node.add_variable . For BNF see variable
variable inheritance	When a *variable* is needed at *job creation* time, it is first sought in the *task* itself. If it is not found in the *task* , it is sought from the task’s parent and so on, up through the *node* levels until found. For any *node* , there are two places to look for variables. Suite definition variables are looked for first, and then any generated variables.

variable substitution	Takes place during *pre-processing* *or* command invocation.(i.e.. ECF_JOB_CMD,ECF_KILL_CMD,etc) It involves searching each line of *ecf script* file or command, for *ECF_MICRO* character. typically '%' The text between two % character, defines a variable. i.e.. %VAR% This variable is searched for in the *suite definition* . First the suite definition variables(sometimes referred to as user variables) are searched and then Repeat variable name, and finally the generated variables. If no variable is found then the same search pattern is repeated up the node tree. The value of the *variable* is replaced between the % characters. If the micro character are not paired and an error message is written to the log file, and the task is placed into the *aborted* state. If the variable is not found in the *suite definition* during pre-processing then *job creation* fails, and an error message is written to the log file, and the task is placed into the *aborted* state. To avoid this, variables in the *ecf script* can be defined as: `%VAR:replacement%` This is similar to %VAR% but if VAR is not found in the *suite definition* then 'replacement' is used.
virtual clock	Like *real clock* until the *ecflow_server* is suspended (i.e.. *shutdown* or *halted*), the suites *clock* is also suspended. Hence will honour relative times in *cron* , *today* and *time* dependencies. It is possible to have a combination of hybrid/real and virtual. More useful when we want complete adherence to time related dependencies at the expense being out of sync with system time.
zombie	Zombies are running jobs that fail authentication when communicating with the *ecflow_server* *child command* s like (init, event,meter, label, abort,complete) are placed in the *ecf script* file and are used to communicate with the *ecflow_server* . The *ecflow_server* authenticates each connection attempt made by the *child command* . Authentication can fail for a number of reasons: • *password(ECF_PASS) supplied with the* child command *, does not match the one in the* ecflow_server. (ecf_passwd) • *path name(ECF_NAME) supplied with the* child command *, does not locate a* task *in the* ecflow_server. (ecf_path) • *process id(ECF_RID) supplied with* child command *, does not correspond with the one stored in the* ecflow_server. (ecf_pid) • task *is already* active *, but receives another init* child command (ecf) • task *is already* complete *, but receives another* child command *(ecf)* • task *is already* aborted *, but receives another* child command *(ecf)* When authentication fails the job is considered to be a zombie. The *ecflow_server* will keep a note of the zombie for a period of time, before it is automatically removed. However the removed zombie, may well re-appear. This is because each *child command* will continue attempting to contact the *ecflow_server* for 24 hours. This is configurable see ECF_TIMEOUT/ECF_ZOMBIE_TIMEOUT on *ecflow_client* For python see ecflow.ZombieAttr , ecflow.ZombieUserActionType There are several types of zombies see *zombie type* and ecflow.ZombieType
zombie attribute	The zombie attribute defines how a *zombie* should be handled in an **automated** fashion. Very careful consideration should be taken before this attribute is added as it may hide a genuine problem. It can be added to any *node* . But is best defined at the *suite* or *family* level. If there is no zombie attribute the default behaviour for init,complete,wait and abort *child command* s, is to block, whereas for label, event, meter the default behaviour is to **fob**. (from version 4.0.4, previously all *child command* s blocked). To add a zombie attribute in python, please see: ecflow.ZombieAttr
zombie type	See *zombie* and class ecflow.ZombieAttr for further information. How do zombies arise. • Server crashed (or terminated and restarted) and the recovered *check point* file is out of date. • A *task* is repeatedly re-run, earlier copies will not be remembered. • Job sent by another *ecflow_server* , but which cannot talk to the original *ecflow_server* • Network glitches/network down • errors in script, i.e. multiple calls to init, complete • errors in job submission i.e. job submitted twice. There are several types of zombies: • *path* — ○ *The task path cannot be found in the server, because node tree was deleted, replaced,reload, server crashed or backup server does not have node tree.* ○ *Jobs could have been created, via server* scheduling *or by* user commands • *user* — *Job is created by* user commands *like, rerun, re-queue. User zombies are differentiated from server(scheduled) since they are automatically created when the force option is used and we have tasks in an* active *or* submitted *states.* • *ecf* — *Jobs are created as part of the normal* scheduling. Two init commands or task complete or aborted but receives another child cmd • ***ecf_pid*** — ***pid mismatched, Job scheduled twice . Check submitter*** • ***ecf_passwd*** — ***Password mismatch, PID matches, system has re-cycled PID or hacked job file?*** • ***ecf_pid_passwd*** — ***Both PID and password mismatch. Re-queue & submit of active job?*** The type of the zombie is not fixed and may change.