The Cylc Suite Engine
User Guide

7.3.0

Released Under the GNU GPL v3.0 Software License

Copyright (C) 2008-2017 NIWA

Hilary Oliver

April 11, 2017

Abstract

Cylc ("silk") is a metascheduler[1] for cycling environmental forecasting suites containing many forecast models and associated processing tasks. Cylc has a novel self-organising scheduling algorithm: a pool of task proxy objects, that each know just their own inputs and outputs, negotiate dependencies so that correct scheduling emerges naturally at run time. Cylc does not group tasks artificially by forecast cycle[2] (each task has a private cycle time and is self-spawning - there is no suite-wide cycle time) and handles dependencies within and between cycles equally so that tasks from multiple cycles can run at once to the maximum possible extent. This matters in particular whenever the external driving data [3] for upcoming cycles are available in advance: cylc suites can catch up from delays very quickly, parallel test suites can be started behind the main operation to catch up quickly, and one can likewise achieve greater throughput in historical case studies; the usual sequence of distinct forecast cycles emerges naturally if a suite catches up to real time operation. Cylc can easily use existing tasks and can run suites distributed across a heterogenous network. Suites can be stopped and restarted in any state of operation, and they dynamically adapt to insertion and removal of tasks, and to delays or failures in particular tasks or in the external environment: tasks not directly affected will carry on cycling as normal while the problem is addressed, and then the affected tasks will catch up as quickly as possible. Cylc has comprehensive command line and graphical interfaces, including a dependency graph based suite control GUI. Other notable features include suite databases; a fast simulation mode; a structured, validated suite definition file format; dependency graph plotting; task event hooks for centralized alerting; and cryptographic suite security.

---

[1]A metascheduler determines when dependent jobs are *ready to run* and then submits them to run by other means, usually a batch queue scheduler. The term can also refer to an aggregate view of multiple distributed resource managers, but that is not the topic of this document. We drop the "meta" prefix from here on because a metascheduler is also a type of scheduler.

[2]A *forecast cycle* comprises all tasks with a common *cycle time* (later referred to here as *cycle point*) i.e. the analysis time or nominal start time of a forecast model, or that of the associated forecast model(s) for other tasks.

[3]Forecast suites are typically driven by real time observational data or timely model fields from an external forecasting system.

Contents

List of Figures

1 Introduction: How Cylc Works

1.1 Scheduling Forecast Suites

Environmental forecasting suites generate forecast products from a potentially large group of interdependent scientific models and associated data processing tasks. They are constrained by availability of external driving data: typically one or more tasks will wait on real time observations and/or model data from an external system, and these will drive other downstream tasks, and so on. The dependency diagram for a single forecast cycle point in such a system is a *Directed Acyclic Graph* as shown in Figure 1 (in our terminology, a *forecast cycle point* is comprised of all tasks with a common *cycle point*, which is the nominal analysis time or start time of the forecast models in the group). In real time operation processing will consist of a series of distinct forecast cycle points that are each initiated, after a gap, by arrival of the new cycle point's external driving data.

From a job scheduling perspective task execution order in such a system must be carefully controlled in order to avoid dependency violations. Ideally, each task should be queued for execution at the instant its last prerequisite is satisfied; this is the best that can be done even if queued tasks are not able to execute immediately because of resource contention.

1.2 EcoConnect

Cylc was developed for the EcoConnect Forecasting System at NIWA (National Institute of Water and Atmospheric Research, New Zealand). EcoConnect takes real time atmospheric and stream flow observations, and operational global weather forecasts from the Met Office (UK), and uses these to drive global sea state and regional data assimilating weather models, which in turn drive regional sea state, storm surge, and catchment river models, plus tide prediction, and a large number of associated data collection, quality control, preprocessing, post-processing, product generation, and archiving tasks.[4] The global sea state forecast runs once daily. The regional weather forecast runs four times daily but it supplies surface winds and pressure to several downstream models that run only twice daily, and precipitation accumulations to catchment river models that run on an hourly cycle assimilating real time stream flow observations and using the most recently available regional weather forecast. EcoConnect runs on heterogeneous distributed hardware, including a massively parallel supercomputer and several Linux servers.

1.3 Dependence Between Tasks

1.3.1 Intra-cycle Dependence

Most dependence between tasks applies within a single forecast cycle point. Figure 1 shows the dependency diagram for a single forecast cycle point of a simple example suite of three forecast models (*a, b,* and *c*) and three post processing or product generation tasks (*d, e* and *f*). A scheduler capable of handling this must manage, within a single forecast cycle point, multiple parallel streams of execution that branch when one task generates output for several downstream tasks, and merge when one task takes input from several upstream tasks.

---

[4]Future plans for EcoConnect include additional deterministic regional weather forecasts and a statistical ensemble.

15
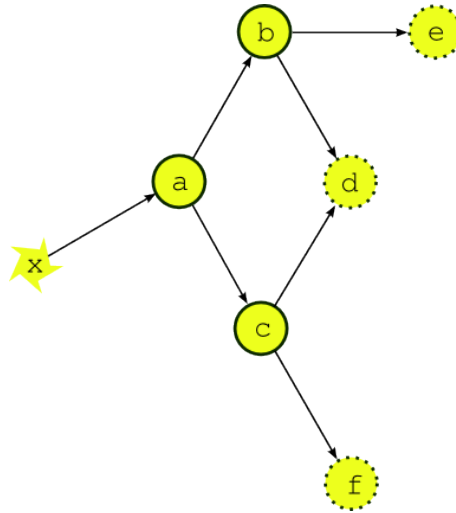
Figure 1: The dependency graph for a single forecast cycle point of a simple example suite. Tasks *a, b,* and *c* represent forecast models, *d, e* and *f* are post processing or product generation tasks, and *x* represents external data that the upstream forecast model depends on.
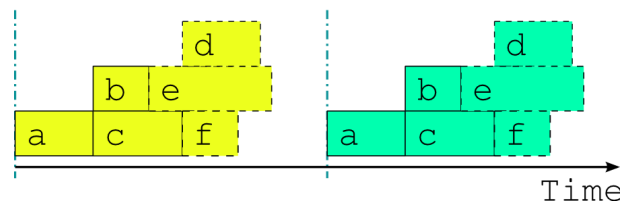
Figure 2: The optimal job schedule for two consecutive cycle points of our example suite during real time operation, assuming that all tasks trigger off upstream tasks finishing completely. The horizontal extent of a task bar represents its execution time, and the vertical blue lines show when the external driving data becomes available.
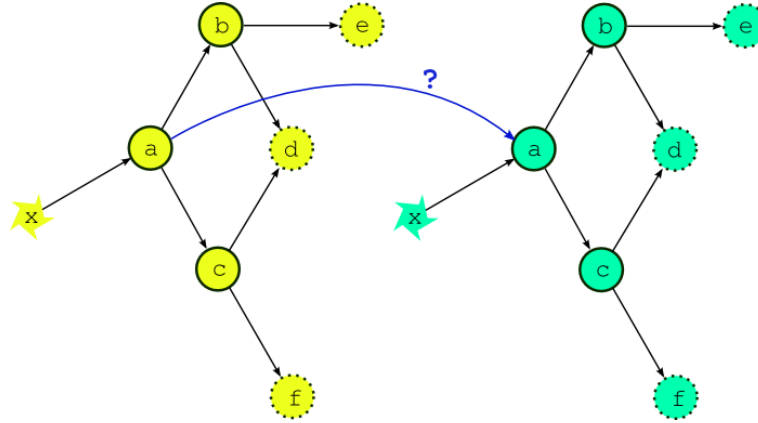
Figure 3: If the external driving data is available in advance, can we start running the next cycle point early?
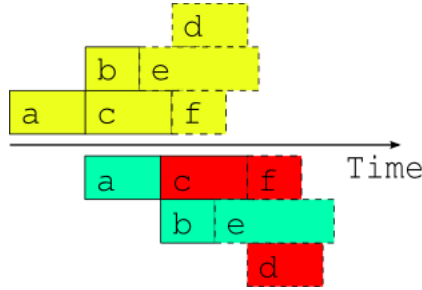
Figure 4: A naive attempt to overlap two consecutive cycle points using the single-cycle-point dependency graph. The red shaded tasks will fail because of dependency violations (or will not be able to run because of upstream dependency violations).

Figure 2 shows the optimal job schedule for two consecutive cycle points of the example suite in real time operation, given execution times represented by the horizontal extent of the task bars. There is a time gap between cycle points as the suite waits on new external driving data. Each task in the example suite happens to trigger off upstream tasks *finishing*, rather than off any intermediate output or event; this is merely a simplification that makes for clearer diagrams.

Now the question arises, what happens if the external driving data for upcoming cycle points is available in advance, as it would be after a significant delay in operations, or when running a historical case study? While the forecast model *a* appears to depend only on the external data *x* at this stage of the discussion, in fact it would typically also depend on its own previous instance for the model *background state* used in initializing the new forecast. Thus, as alluded to in Figure 3, task *a* could in principle start as soon as its predecessor has finished. Figure 4 shows, however, that starting a whole new cycle point at this point is dangerous - it results in dependency violations in half of the tasks in the example suite. In fact the situation could be
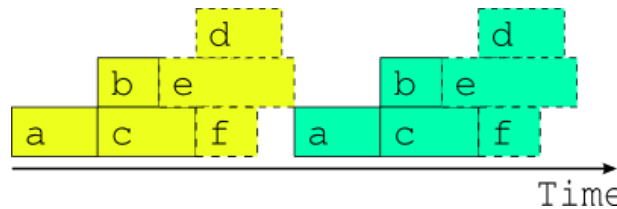
Figure 5: The best that can be done *in general* when inter-cycle dependence is ignored.

even worse than this - imagine that task *b* in the first cycle point is delayed for some reason *after* the second cycle point has been launched. Clearly we must consider handling inter-cycle dependence explicitly or else agree not to start the next cycle point early, as is illustrated in Figure 5.

1.3.2   Inter-Cycle Dependence

Forecast models typically depend on their own most recent previous forecast for background state or restart files of some kind (this is called *warm cycling*) but there can also be inter-cycle dependence between different tasks. In an atmospheric forecast analysis suite, for instance, the weather model may generate background states for observation processing and data-assimilation tasks in the next cycle point as well as for the next forecast model run. In real time operation inter-cycle dependence can be ignored because it is automatically satisfied when one cycle point finishes before the next begins. If it is not ignored it drastically complicates the dependency graph by blurring the clean boundary between cycle points. Figure 6 illustrates the problem for our simple example suite assuming minimal inter-cycle dependence: the warm cycled models (*a*, *b*, and *c*) each depend on their own previous instances.

For this reason, and because we tend to see forecasting suites in terms of their real time characteristics, other metaschedulers have ignored inter-cycle dependence and are thus restricted to running entire cycle points in sequence at all times. This does not affect normal real time operation but it can be a serious impediment when advance availability of external driving data makes it possible, in principle, to run some tasks from upcoming cycle points before the current cycle point is finished - as was suggested at the end of the previous section. This can occur, for instance, after operational delays (late arrival of external data, system maintenance, etc.) and to an even greater extent in historical case studies and parallel test suites started behind a real time operation. It can be a serious problem for suites that have little downtime between forecast cycle points and therefore take many cycle points to catch up after a delay. Without taking account of inter-cycle dependence, the best that can be done, in general, is to reduce the gap between cycle points to zero as shown in Figure 5. A limited crude overlap of the single cycle point job schedule may be possible for specific task sets but the allowable overlap may change if new tasks are added, and it is still dangerous: it amounts to running different parts of a dependent system as if they were not dependent and as such it cannot be guaranteed that some unforeseen delay in one cycle point, after the next cycle point has begun, (e.g. due to resource contention or task failures) won't result in dependency violations.

Figure 7 shows, in contrast to Figure 4, the optimal two cycle point job schedule obtained by respecting all inter-cycle dependence. This assumes no delays due to resource contention or otherwise - i.e. every task runs as soon as it is ready to run. The scheduler running this suite must be able to adapt dynamically to external conditions that impact on multi-cycle-point scheduling in the presence of inter-cycle dependence or else, again, risk bringing the system down with dependency violations.

To further illustrate the potential benefits of proper inter-cycle dependency handling, Figure 8 shows an operational delay of almost one whole cycle point in a suite with little downtime between cycle points. Above the time axis is the optimal schedule that is possible in principle when inter-cycle dependence is taken into account, and below it is the only safe schedule possible *in general* when it is ignored. In the former case, even the cycle point immediately after the delay is hardly affected, and subsequent cycle points are all on time, whilst in the latter case it takes five full cycle points to catch up to normal real time operation.
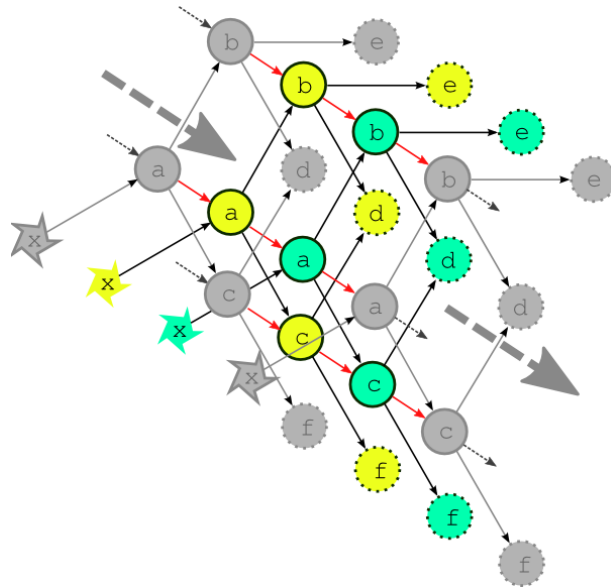
Figure 6: The complete dependency graph for the example suite, assuming the least possible inter-cycle dependence: the forecast models (a, b, and c) depend on their own previous instances. The dashed arrows show connections to previous and subsequent forecast cycle points.
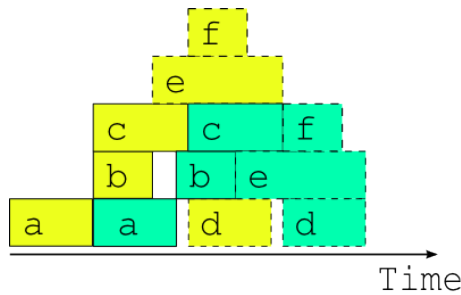
Figure 7: The optimal two cycle job schedule when the next cycle's driving data is available in advance, possible in principle when inter-cycle dependence is handled explicitly.
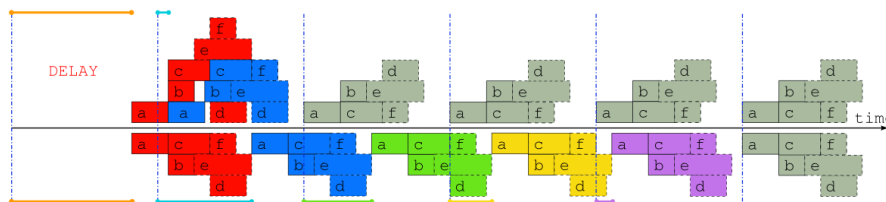
Figure 8: Job schedules for the example suite after a delay of almost one whole forecast cycle point, when inter-cycle dependence is taken into account (above the time axis), and when it is not (below the time axis). The colored lines indicate the time that each cycle point is delayed, and normal "caught up" cycle points are shaded gray.

19
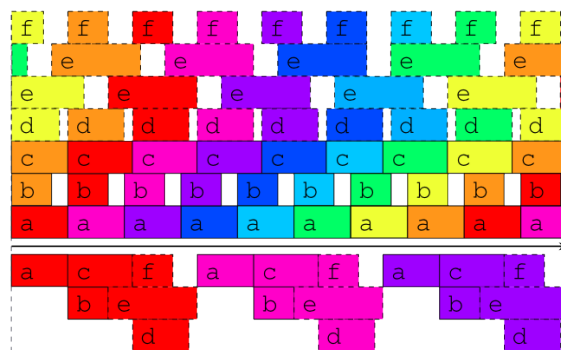
Figure 9: Job schedules for the example suite in case study mode, or after a long delay, when the external driving data are available many cycle points in advance. Above the time axis is the optimal schedule obtained when the suite is constrained only by its true dependencies, as in Figure 3, and underneath is the best that can be done, in general, when inter-cycle dependence is ignored.

Similarly, Figure 9 shows example suite job schedules for an historical case study, or when catching up after a very long delay; i.e. when the external driving data are available many cycle points in advance. Task *a*, which as the most upstream forecast model is likely to be a resource intensive atmosphere or ocean model, has no upstream dependence on co-temporal tasks and can therefore run continuously, regardless of how much downstream processing is yet to be completed in its own, or any previous, forecast cycle point (actually, task *a* does depend on co-temporal task *x* which waits on the external driving data, but that returns immediately when the data is available in advance, so the result stands). The other forecast models can also cycle continuously or with a short gap between, and some post processing tasks, which have no previous-instance dependence, can run continuously or even overlap (e.g. *e* in this case). Thus, even for this very simple example suite, tasks from three or four different cycle points can in principle run simultaneously at any given time.

In fact, if our tasks are able to trigger off internal outputs of upstream tasks (message triggers) rather than waiting on full completion, then successive instances of the forecast models could overlap as well (because model restart outputs are generally completed early in the forecast) for an even more efficient job schedule.

1.4   The Cylc Scheduling Algorithm

Cylc manages a pool of proxy objects that represent the real tasks in a suite. Task proxies know how to run the real tasks that they represent, and they receive progress messages from the tasks as they run (usually reports of completed outputs). There is no global cycling mechanism to advance the suite; instead individual task proxies have their own private cycle point and spawn their own successors when the time is right. Task proxies are self-contained - they know their own prerequisites and outputs but are not aware of the wider suite. Inter-cycle dependence is not treated as special, and the task pool can be populated with tasks with many different cycle points. The task pool is illustrated in Figure 10. *Whenever any task changes state due to completion of an output, every task checks to see if its own prerequisites have been satisfied.* In effect, cylc gets a pool of tasks to self-organize by negotiating their own dependencies so that optimal scheduling, as described in the previous section, emerges naturally at run time.
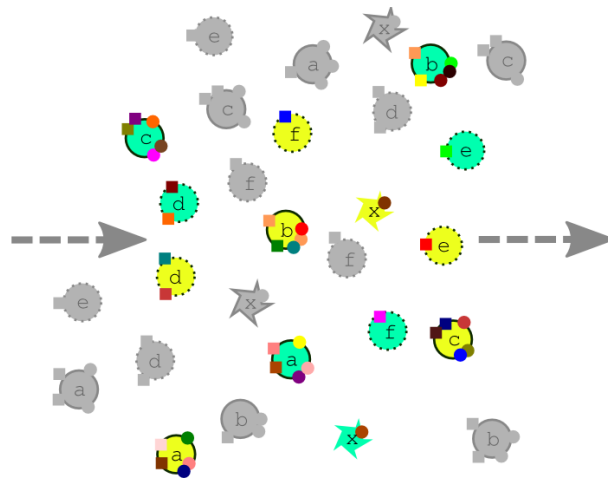
Figure 10: How cylc sees a suite, in contrast to the multi-cycle-point dependency graph of Figure 6. Task colors represent different cycle points, and the small squares and circles represent different prerequisites and outputs. A task can run when its prerequisites are satisfied by the outputs of other tasks in the pool.

2   Cylc Screenshots

Figure 11: gcylc graph and dot views.

Figure 12: gcylc text view.

Figure 13: gscan multi-suite state summary GUI.
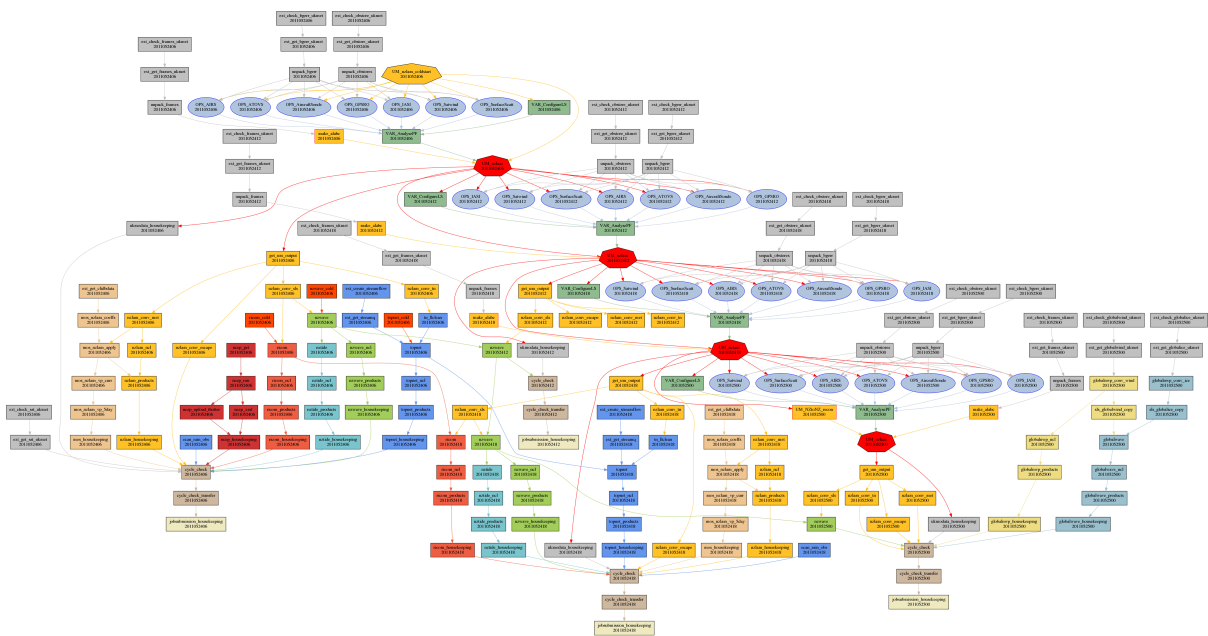
Figure 14: A large-ish suite graphed by cylc.

3   Installation

Cylc runs on Unix variants, usually Linux, and including Apple OS X.

3.1   External Software Packages

Python `>=` 2.6 is required (but not yet Python 3). Python should already be installed in your Linux system. https://python.org.

The following packages are highly recommended, but are technically optional as you can construct and run suites without dependency graph visualisation or the Cylc GUIs:

- **PyGTK** - GUI toolkit (should be in your system Python): http://www.pygtk.org.
- **Graphviz** - graph layout engine (tested 2.36.0): http://www.graphviz.org.
- **Pygraphviz** - Python Graphviz interface (tested 1.2): http://pygraphviz.github.io/. To build this you may need some *devel* packages too:
    - python-devel
    - graphviz-devel

The User Guide is generated from LaTeXsource files by running `make` in the top level Cylc directory. The specific packages required may vary by distribution, e.g.:

- texlive
- texlive-tocloft
- texlive-framed
- texlive-preprint (for `fullpage.sty`)
- texlive-tex4ht

To generate the HTML User Guide **ImageMagick** is also needed.

In most modern Linux distributions all of the software above can be installed via the system package manager. Otherwise download packages manually and follow their native installation instructions. To check that all (non LaTeXpackages) are installed properly:

```
$ cylc check-software
Checking for Python >= 2.6 ... found 2.7.6 ... ok
Checking for non-Python packages:
 + Graphviz ... ok
Checking for Python packages:
 + pygraphviz ... ok
 + pygtk ... ok
```

If errors are reported then the packages concerned are either not installed or not in your Python search path. (Note that `cylc check-software` has become quite trivial as we've removed or bundled some former dependencies, but in future we intend to make it print a comprehensive list of library versions etc. to include in with bug reports.)

3.2   Software Bundled With Cylc

Cylc bundles several third party packages which do not need to be installed separately.

- **cherrypy 6.0.2** (slightly modified): a pure Python HTTP framework that we use as a web server for communication between server processes (suite daemons) and client programs

(running tasks, GUIs, CLI commands). Client communication is via the Python **requests** library if available (recommended) or else pure Python via **urllib2**.
http://www.cherrypy.org/
http://docs.python-requests.org/

- **Jinja2 2.8**: a full featured template engine for Python, and its dependency **MarkupSafe 0.23**; both BSD licensed.
http://jinja.pocoo.org/
http://www.pocoo.org/projects/markupsafe/

- the **xdot** graph viewer (modified), LGPL licensed:
https://github.com/jrfonseca/xdot.py

3.3 Installing Cylc Itself

Cylc releases can be downloaded from from https://cylc.github.io/cylc.

The wrapper script `admin/cylc-wrapper` should be installed as `cylc` somewhere in the system executable search path (e.g. `/usr/local/bin/`) and modified slightly to point to a location such as `/opt/cylc/` where successive Cylc releases will be unpacked side by side.

To install Cylc for the first time simply unpack the release tarball in that location, type `make` in it, and set your site defaults in a site config file (below).

Installing subsequent releases is just a matter of unpacking the new tarballs next to the previous releases, running `make` in them, and copying in (possibly with modifications) the previous site config file.

3.3.1 Local User Installation

It is easy to install Cylc under your own user account if you don't have root or sudo access to the system: just put the central Cylc wrapper in `$HOME/bin/` (making sure that is in your `$PATH`) and modify it to point to a directory such as `$HOME/cylc/` where you will unpack and install release tarballs. Local installation of third party dependencies like Graphviz is also possible, but that depends on the particular installation methods used and is outside of the scope of this document.

3.3.2 Create A Site Config File

Site and user global config files define some important parameters that affect all suites, some of which may need to be customized for your site. See 6 for how to generate an initial site file and where to install it. All legal site and user global config items are defined in B.

3.3.3 Configure Site Environment on Job Hosts

If your users submit task jobs to hosts other than the hosts they use to run their suites, you should ensure that the job hosts have the correct environment for running cylc. A cylc suite generates task job scripts that normally invoke `bash`. The job will attempt to source the first of these files it finds to set up its environment:

- `${HOME}/.cylc/job-init-env.sh`
- `${CYLC_DIR}/conf/job-init-env.sh`
- `${CYLC_DIR}/conf/job-init-env-default.sh`

The `${CYLC_DIR}/conf/job-init-env-default.sh` file is provided in the cylc distribution, and will attempt to source `/etc/profile` and `${HOME}/.profile`. If this behaviour is not desirable, you should override it by adding a `${CYLC_DIR}/conf/job-init-env.sh` file and populate it with the appropriate contents.

3.4   Automated Tests

The cylc test battery is primarily intended for developers to check that changes to the source code don't break existing functionality. Note that some test failures can be expected to result from suites timing out, even if nothing is wrong, if you run too many tests in parallel. See `cylc test-battery --help`.

4   Cylc Terminology

4.1   Jobs and Tasks

A *job* is a program or script that runs on a computer, and a *task* is a workflow abstraction - a node in the suite dependency graph - that represents a job.

4.2   Cycle Points

A *cycle point* is a particular date-time (or integer) point in a sequence of date-time (or integer) points. Each cylc task has a private cycle point and can advance independently to subsequent cycle points. It may sometimes be convenient, however, to refer to the "current cycle point" of a suite (or the previous or next one, etc.) with reference to a particular task, or in the sense of all tasks instances that "belong to" a particular cycle point. But keep in mind that different tasks may pass through the "current cycle point" (etc.) at different times as the suite evolves.

5   Workflows For Cycling Systems

A model run and associated processing may need to be cycled for the following reasons:

- In real time forecasting systems, a new forecast may be initiated at regular intervals when new real time data comes in.
- Batch scheduler queue limits may require that long single runs be split into many smaller runs with incremental processing of associated inputs and outputs.

Cylc provides two ways of constructing workflows for cycling systems: *cycling workflows* and *parameterized tasks*.

5.1   Cycling Workflows

This is cylc's classic cycling mode as described in the Introduction. Each instance of a cycling job is represented by a new instance of *the same task*, with a new cycle point. The suite configuration defines patterns for extending the workflow on the fly, so it can keep running indefinitely if necessary. For example, to cycle `model.exe` on a monthly sequence we could define a single task `model`, an initial cycle point, and a monthly sequence. Cylc then generates the date-time sequence and creates a new task instance for each cycle point as it comes up. Workflow dependencies are defined generically with respect to the "current cycle point" of the tasks involved.

This is the only sensible way to run very large suites or operational suites that need to continue cycling indefinitely. The cycling is configured with standards-based ISO 8601 date-time *recurrence expressions*. Multiple cycling sequences can be used at once in the same suite. See Section 9.3.

5.2   Parameterized Tasks as a Proxy for Cycling

It is also possible to run cycling jobs with a pre-defined static workflow in which each instance of a cycling job is represented by *a different task*: as far as the abstract workflow is concerned there is no cycling. The sequence of tasks can be constructed efficiently, however, using cylc's built-in suite parameters (9.6.7) or explicit Jinja2 loops (9.7).

For example, to run `model.exe` 12 times on a monthly cycle we could loop over an integer parameter `R = 0, 1, 2, ..., 11` to define tasks `model-R0, model-R1, model-R2, ...model-R11`, and the parameter values could be multiplied by the interval `P1M` (one month) to get the start point point for the corresponding model run.

This method is only good for smaller workflows of finite duration because every single task has to be mapped out in advance, and cylc has to be aware of all of them throughout the entire run. Additionally Cylc's *cycling workflow* capabilities (above) are more powerful, more flexible, and generally easier to use (Cylc will do the date-time arithmetic for you, for instance), so that is the recommended way to drive most cycling systems.

The primary use for parameterized tasks in cylc is to generate ensembles and other groups of related tasks at the same cycle point, not as a proxy for cycling.

5.3   Mixed Cycling Workflows

For completeness we note that parameterized cycling can be used within a cycling workflow. For example, in a daily cycling workflow long (daily) model runs could be split into four shorter runs by parameterized cycling. A simpler six-hourly cycling workflow should be considered first, however.

6   Global (Site, User) Configuration Files

Cylc site and user global configuration files contain settings that affect all suites. Some of these, such as the range of network ports used by cylc, should be set at site level,

```
# cylc site global config file
/path/to/cylc/conf/global.rc
# Deprecated path to cylc site global config file
/path/to/cylc/conf/siterc/site.rc
```

Others, such as the preferred text editor for suite definitions, can be overridden by users,

```
# cylc user global config file
~/.cylc/global.rc
# Deprecated cylc user global config file
~/.cylc/user.rc
```

The `cylc get-site-config` command retrieves current global settings consisting of cylc defaults overridden by site settings, if any, overridden by user settings, if any. To generate an initial site or user global config file:

```
$ cylc get-site-config > $HOME/.cylc/global.rc
```

Settings that do not need to be changed should be deleted or commented out of user global config files so that they don't override future changes to the site file.

Legal items, values, and system defaults are documented in (B).

7   Tutorial

This section provides a hands-on tutorial introduction to basic cylc functionality.

7.1   User Config File

Some settings affecting cylc's behaviour can be defined in site and user *global config files*. For example, to choose the text editor invoked by cylc on suite definitions:

```
# $HOME/.cylc/global.rc
[editors]
    terminal = vim
    gui = gvim -f
```

- For more on site and user global config files see 6 and B.

7.1.1   Configure Environment on Job Hosts

If you submit task jobs to hosts other than the hosts you use to run your suites, you may need to customise the environment for running cylc. A cylc suite generates task job scripts that normally invoke `bash`. The job will attempt to source the first of these files it finds to set up its environment:

- `${HOME}/.cylc/job-init-env.sh`
- `${CYLC_DIR}/conf/job-init-env.sh`
- `${CYLC_DIR}/conf/job-init-env-default.sh`

The `${CYLC_DIR}/conf/job-init-env-default.sh` file is provided in the cylc distribution, and will attempt to source `/etc/profile` and `${HOME}/.profile`. If this behaviour is not desirable, your site administrator should have overridden it by adding a `${CYLC_DIR}/conf/job-init-env.sh` file and populate it with the appropriate contents. If customisation is still required, you can add your own `${HOME}/.cylc/job-init-env.sh` file and populate it with the appropriate contents.

7.2   User Interfaces

You should have access to the cylc command line (CLI) and graphical (GUI) user interfaces once cylc has been installed as described in Section 3.3.

7.2.1   Command Line Interface (CLI)

The command line interface is unified under a single top level `cylc` command that provides access to many sub-commands and their help documentation.

```
$ cylc help       # Top level command help.
$ cylc run --help # Example command-specific help.
```

Command help transcripts are printed in E and are available from the GUI Help menu.

Cylc is *scriptable* - the error status returned by commands can be relied on.

7.2.2   Graphical User Interface (GUI)

The cylc GUI covers the same functionality as the CLI, but it has more sophisticated suite monitoring capability. It can start and stop suites, or connect to suites that are already running; in either case, shutting down the GUI does not affect the suite itself.

```
$ gcylc & # or:
$ cylc gui & # Single suite control GUI.
$ cylc gscan & # Multi-suite monitor GUI.
```

Clicking on a suite in gscan, shown in Figure 13, opens a gcylc instance for it.

7.3   Suite Definitions

Cylc suites are defined by extended-INI format `suite.rc` files (the main file format extension is section nesting). These reside in *suite definition directories* that may also contain a `bin` directory and any other suite-related files.

- For more on the suite definition file format, see 9 and A.

7.4   Suite Registration

Suite registration creates a run directory (under `~/cylc-run/` by default) and populates it with authentication files and a symbolic link to a suite definition directory. Cylc commands that parse suite definitions can take the file path or the suite name as input. Commands that interact with running suites have to target the suite by name.

```
# Target a suite by file path:
$ cylc validate /path/to/my/suite/suite.rc
$ cylc graph /path/to/my/suite/suite.rc

# Register a suite:
$ cylc register my.suite /path/to/my/suite/

# Target a suite by name:
$ cylc graph my.suite
$ cylc validate my.suite
```

```
$ cylc run my.suite
$ cylc stop my.suite
# etc.
```

7.5   Suite Passphrases

Registration (above) also generates a suite-specific passphrase file under `.service/` in the suite run directory. It is loaded by the suite daemon at start-up and used to authenticate connections from client programs.

Possession of a suite's passphrase file gives full control over it. Without it, the information avaiable to a client is determined by the suite's public access privilege level.

For more on connection authentication, suite passphrases, and public access, see 12.6.

7.6   Import The Example Suites

Run the following command to copy cylc's example suites and register them for your own use:

```
$ cylc import-examples /tmp
```

7.7   Rename The Imported Tutorial Suites

Suites can be renames by simply renaming (i.e. moving) their run directories. Make the tutorial suite names shorter, and print their locations with `cylc print`:

```
$ mv ~/cylc-run/$(cylc --version)/examples/tutorial ~/cylc-run/tut
$ cylc print -ya tut
tut/oneoff/jinja2  | /tmp/cylc-examples/7.0.0/tutorial/oneoff/jinja2
tut/cycling/two    | /tmp/cylc-examples/7.0.0/tutorial/cycling/two
tut/cycling/three  | /tmp/cylc-examples/7.0.0/tutorial/cycling/three
# ...
```

See `cylc print --help` for other display options.

7.8   Suite Validation

Suite definitions can be validated to detect syntax (and other) errors:

```
# pass:
$ cylc validate tut/oneoff/basic
Valid for cylc-6.0.0
$ echo $?
0
# fail:
$ cylc validate my/bad/suite
Illegal item: [scheduling]special tusks
$ echo $?
1
```

7.9   Hello World in Cylc

suite: `tut/oneoff/basic`

Here's the traditional *Hello World* program rendered as a cylc suite:

```
title = "The cylc Hello World! suite"
[scheduling]
    [[dependencies]]
        graph = "hello"
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
```

Cylc suites feature a clean separation of scheduling configuration, which determines *when* tasks are ready to run; and runtime configuration, which determines *what* to run (and *where* and *how* to run it) when a task is ready. In this example the `[scheduling]` section defines a single task called `hello` that triggers immediately when the suite starts up. When the task finishes the suite shuts down. That this is a *dependency graph* will be more obvious when more tasks are added. Under the `[runtime]` section the `script` item defines a simple inlined implementation for `hello`: it sleeps for ten seconds, then prints `Hello World!`, and exits. This ends up in a *job script* generated by cylc to encapsulate the task (below) and, thanks to some defaults designed to allow quick prototyping of new suites, it is submitted to run as a background job on the suite host. In fact cylc even provides a default task implementation that makes the entire `[runtime]` section technically optional:

```
title = "The minimal complete runnable cylc suite"
[scheduling]
    [[dependencies]]
        graph = "foo"
# (actually, 'title' is optional too ... and so is this comment)
```

(the resulting *dummy task* just prints out some identifying information and exits).

7.10   Editing Suites

The text editor invoked by cylc on suite definitions is determined by cylc site and user global config files, as shown above in 7.2. Check that you have renamed the tutorial examples suites as described just above and open the *Hello World* suite definition in your text editor:

```
$ cylc edit tut/oneoff/basic # in-terminal
$ cylc edit -g tut/oneoff/basic & # or GUI
```

Alternatively, start gcylc on the suite:

```
$ gcylc tut/oneoff/basic &
```

and choose *Suite* →*Edit* from the menu.

The editor will be invoked from within the suite definition directory for easy access to other suite files (in this case there are none). There are syntax highlighting control files for several text editors under `/path/to/cylc/conf/`; see in-file comments for installation instructions.

7.11 Running Suites

7.11.1 CLI

Run `tut/oneoff/basic` using the `cylc run` command. As a suite runs detailed timestamped information is written to a *suite log* and progress can be followed with cylc's suite monitoring tools (below). By default a running suite daemonizes after printing a short message so that you can exit the terminal or even log out without killing the suite:

```
$ cylc run tut/oneoff/basic
        . _ .
         | _ |
        | | |                        The Cylc Suite Engine [7.0.0]
        | | |                        Copyright (C) 2008-2017 NIWA
._____._. ._| |._____.
| .___| | | | | .___|
| !___| !_! | | | !___.  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
!_____!___. |_!_____!  This program comes with ABSOLUTELY NO WARRANTY;
    .___! |              see 'cylc warranty'.  It is free software, you
    !_____!                are welcome to redistribute it under certain
                          conditions; see 'cylc conditions'.

*** listening on nwp-1:43027 ***

To view suite daemon contact information:
 $ cylc get-suite-contact tut/oneoff/basic

Other ways to see if the suite is still running:
 $ cylc scan -n '\btut/oneoff/basic\b' nwp-1
 $ cylc ping -v --host=nwp-1 tut/oneoff/basic
 $ ssh nwp-1 "pgrep -a -P 1 -fu $USER 'cylc-r.* \btut/oneoff/basic\b'"
```

If you're quick enough (this example only takes 10-15 seconds to run) the `cylc scan` command will detect the running suite:

```
$ cylc scan
tut/oneoff/basic oliverh@nwp-1:43027
```

Note you can use the `--no-detach` and `--debug` options to `cylc-run` to prevent the suite from daemonizing (i.e. to make it stay attached to your terminal until it exits).

When a task is ready cylc generates a *job script* to run it, by default as a background jobs on the suite host. The job process ID is captured, and job output is directed to log files in standard locations under the suite run directory.

Log file locations relative to the suite run directory look like `job/1/hello/01/` where the first digit is the *cycle point* of the task `hello` (for non-cycling tasks this is just '1'); and the final `01` is the *submit number* (tasks can be made to retry on failure or manually retriggered - each time a new log directory is used to avoid overwriting previous output).

The suite shuts down automatically once all tasks have succeeded.

7.11.2 GUI

The cylc GUI can start and stop suites, or (re)connect to suites that are already running:

```
$ cylc gui tut/oneoff/basic &
```

Use the tool bar *Play* button, or the *Control→Run* menu item, to run the suite again. You may want to alter the suite definition slightly to make the task take longer to run. Try right-clicking on the `hello` task to view its output logs. The relative merits of the three *suite views* - dot, text, and graph - will be more apparent later when we have more tasks. Closing the GUI does not affect the suite itself.

7.12   Discovering Running Suites

Suites that are currently running can be detected with command line or GUI tools:

```
# list currently running suites and their port numbers:
$ cylc scan
tut/oneoff/basic oliverh@nwp-1:43001

# GUI summary view of running suites:
$ cylc gscan &
```

The scan GUI is shown in Figure 13; clicking on a suite in it opens gcylc.

7.13   Task Identifiers

At run time, task instances are identified by *name*, which is determined entirely by the suite definition, and a *cycle point* which is usually a date-time or an integer:

```
foo.20100808T00Z    # a task with a date-time cycle point
bar.1               # a task with an integer cycle point (could be non-cycling)
```

Non-cycling tasks usually just have the cycle point `1`, but this still has to be used to target the task instance with cylc commands.

7.14   Job Submission: How Tasks Are Executed

suite: `tut/oneoff/jobsub`

Task *job scripts* are generated by cylc to wrap the task implementation specified in the suite definition (environment, script, etc.) in error trapping code, messaging calls to report task progress back to the suite daemon, and so forth. Job scripts are written to the *suite job log directory* where they can be viewed alongside the job output logs. They can be accessed at run time by right-clicking on the task in the cylc GUI, or printed to the terminal:

```
$ cylc cat-log tut/oneoff/basic hello.1
```

This command can also print the suite log (and stdout and stderr for suites in daemon mode) and task stdout and stderr logs (see `cylc cat-log --help`). A new job script can also be generated on the fly for inspection:

```
$ cylc jobscript tut/oneoff/basic hello.1
```

Take a look at the job script generated for `hello.1` during the suite run above. The custom scripting should be clearly visible toward the bottom of the file.

The `hello` task in the first tutorial suite defaults to running as a background job on the suite host. To submit it to the Unix `at` scheduler instead, configure its job submission settings as in `tut/oneoff/jobsub`:

```
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
        [[[job]]]
            batch system = at
```

Run the suite again after checking that `atd` is running on your system.

Cylc supports a number of different batch systems. Tasks submitted to external batch queuing systems like `at`, `PBS`, `SLURM`, `Moab`, or `LoadLeveler`, are displayed as *submitted* in the cylc GUI until they start executing.

- For more on task job scripts, see 10.1.
- For more on batch systems, see 11.1.

7.15 Locating Suite And Task Output

If the `--no-detach` option is not used, suite stdout and stderr will be directed to the suite run directory along with the time-stamped suite log file, and task job scripts and job logs (task stdout and stderr). The default suite run directory location is `$HOME/cylc-run`:

```
$ tree $HOME/cylc-run/tut/oneoff/basic/
|-- .service                  # location of run time service files
|    |-- contact              # detail on how to contact the running suite
|    |-- db                   # private suite run database
|    |-- passphrase           # passphrase for client authentication
|    |-- source               # symbolic link to source directory
|    |-- ssl.cert             # SSL certificate for the suite server
|    '-- ssl.pem              # SSL private key
|-- cylc-suite.db             # back compat symlink to public suite run database
|-- share                     # suite share directory (not used in this example)
|-- work                      # task work space (sub-dirs are deleted if not used)
|    '-- 1                       # task cycle point directory (or 1)
|        '-- hello               # task work directory (deleted if not used)
|-- log                       # suite log directory
|    |-- db                      # public suite run database
|    |-- job                     # task job log directory
|    |    '-- 1                     # task cycle point directory (or 1)
|    |        '-- hello             # task name
|    |            |-- 01               # task submission number
|    |            |   |-- job             # task job script
|    |            |   '-- job-activity.log # task job activity log
|    |            |   |-- job.err          # task stderr log
|    |            |   |-- job.out          # task stdout log
|    |            |   '-- job.status       # task status file
|    |            '-- NN -> 01             # symlink to latest submission number
|    '-- suite                   # suite daemon log directory
|        |-- err                    # suite daemon stderr log (daemon mode only)
|        |-- out                    # suite daemon stdout log (damon mode only)
|        '-- log                    # suite daemon event log (timestamped info)
```

The suite run database files, suite environment file, and task status files are used internally by cylc. Tasks execute in private `work/` directories that are deleted automatically if empty when the task finishes. The suite `share/` directory is made available to all tasks (by `$CYLC_SUITE_SHARE_DIR`) as a common share space. The task submission number increments from 1 if a task retries on failure; this is used a sub-directory of the log tree to avoid overwriting log files from earlier job submissions.

The top level run directory location can be changed in site and user config files if necessary, and the suite share and work locations can be configured separately because of the potentially larger disk space requirement.

Task job logs can be viewed by right-clicking on tasks in the gcylc GUI (so long as the task proxy is live in the suite), manually accessed from the log directory (of course), or printed to the terminal with the `cylc cat-log` command:

```
# suite logs:
$ cylc cat-log   tut/oneoff/basic          # suite event log
```

```
$ cylc cat-log -o tut/oneoff/basic           # suite stdout log
$ cylc cat-log -e tut/oneoff/basic           # suite stderr log
# task logs:
$ cylc cat-log    tut/oneoff/basic hello.1    # task job script
$ cylc cat-log -o tut/oneoff/basic hello.1    # task stdout log
$ cylc cat-log -e tut/oneoff/basic hello.1    # task stderr log
```

- For a web-based interface to suite and task logs (and much more), see *Rose* in 13.
- For more on environment variables supplied to tasks, such as `$CYLC_SUITE_SHARE_DIR`, see 9.4.7.

7.16   Remote Tasks

suite: `tut/oneoff/remote`

The `hello` task in the first two tutorial suites defaults to running on the suite host. To make it run on a remote host instead change its runtime configuration as in `tut/oneoff/remote`:

```
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
        [[[remote]]]
            host = server1.niwa.co.nz
```

For remote task hosting to work several requirements must be satisfied:

- Non-interactive ssh must be enabled from the suite host account to the task host account, for task job submission.
- Your shell initialization (.profile, .bashrc, .cshrc, etc) on the remote host must not produce any standard output as it may confuse commands such as scp. See `http://www.openssh.com/faq.html#2.9` for more information.
- Network settings must allow communication *back* from the task host to the suite host, either by network ports or ssh, unless the last-resort one way *task polling* communication method is used.
- Cylc must be installed on the task host. Other software dependencies like graphviz are not required there.
- Any files needed by a remote task must be installed on the task host. In this example there is nothing to install because the implementation of `hello` is inlined in the suite definition and thus ends up entirely contained within the task job script.

If your username is different on the task host the `[[[remote]]]` section also supports an `owner=username` item, or your `$HOME/.ssh/config` file can be configured for username translation.

If you configure a task host according to the requirements cylc will create remote log directories, source login scripts on the remote host to ensure cylc is visible there, send the task job script over, and submit it to run there by the configured batch system:

Remote task job logs are saved to the suite run directory on the task host, not on the suite host. They can be retrieved by right-clicking on the task in the GUI, or to have cylc pull them back to the suite host automatically do this:

```
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
        [[[remote]]]
            host = server1.niwa.co.nz
            retrieve job logs = True
```

This suite will attempt to `rsync` job logs from the remote host each time a task job completes.

Some batch systems have considerable delays between the time when the job completes and when it writes the job logs in its normal location. If this is the case, you can configure an initial delay and retry delays for job log retrieval by setting some delays. E.g.:

```
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
        [[[remote]]]
            host = server1.niwa.co.nz
            retrieve job logs = True
            # Retry after 10 seconds, 1 minute and 3 minutes
            retrieve job logs retry delays = PT10S, PT1M, PT3M
```

Finally, if the disk space of the suite host is limited, you may want to set `[[[remote]]]retrieve job logs max size=S` The value of SIZE can be anything that is accepted by the `--max-size=SIZE` option of the `rsync` command. E.g.:

```
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
        [[[remote]]]
            host = server1.niwa.co.nz
            retrieve job logs = True
            # Don't get anything bigger than 10MB
            retrieve job logs max size = 10M
```

It is worth noting that cylc uses the existence of a job's `job.out` or `job.err` in the local file system to indicate a successful job log retrieval. If `retrieve job logs max size`SIZE= is set and both `job.out` and `job.err` are bigger than `SIZE` then cylc will consider the retrieval as failed. If retry delays are specified, this will trigger some useless (but harmless) retries. If this occurs regularly, you should try the following:

- Reduce the verbosity of STDOUT or STDERR from the task.
- Redirect the verbosity from STDOUT or STDERR to an alternate log file.
- Adjust the size limit with tolerance to the expected size of STDOUT or STDERR.

- For more on remote tasks see 9.4.9
- For more on task communications, see 12.2.
- For more on suite passphrases and authentication, see 7.5 and 12.6.

7.17   Task Triggering

suite: `tut/oneoff/goodbye`

To make a second task called `goodbye` trigger after `hello` finishes successfully, return to the original example, `tut/oneoff/basic`, and change the suite graph as in `tut/oneoff/goodbye`:

```
[scheduling]
    [[dependencies]]
        graph = "hello => goodbye"
```

or to trigger it at the same time as `hello`,

```
[scheduling]
    [[dependencies]]
        graph = "hello & goodbye"
```

and configure the new task's behaviour under `[runtime]`:

```
[runtime]
    [[goodbye]]
        script = "sleep 10; echo Goodbye World!"
```

Run `tut/oneoff/goodbye` and check the output from the new task:

```
$ cat ~/cylc-run/tut/oneoff/goodbye/log/job/1/goodbye/01/job.out
  # or
$ cylc cat-log -o tut/oneoff/goodbye goodbye.1
JOB SCRIPT STARTING
cylc (scheduler - 2014-08-14T15:09:30+12): goodbye.1 started at 2014-08-14T15:09:30+12
cylc Suite and Task Identity:
  Suite Name  : tut/oneoff/goodbye
  Suite Host  : oliverh-34403dl.niwa.local
  Suite Port  : 43001
  Suite Owner : oliverh
  Task ID     : goodbye.1
  Task Host   : nwp-1
  Task Owner  : oliverh
  Task Try No.: 1

Goodbye World!
cylc (scheduler - 2014-08-14T15:09:40+12): goodbye.1 succeeded at 2014-08-14T15:09:40+12
JOB SCRIPT EXITING (TASK SUCCEEDED)
```

7.17.1 Task Failure And Suicide Triggering

suite: `tut/oneoff/suicide`

Task names in the graph string can be qualified with a state indicator to trigger off task states
other than success:

```
    graph = """
a => b         # trigger b if a succeeds
c:submit => d # trigger d if c submits
e:finish => f # trigger f if e succeeds or fails
g:start  => h # trigger h if g starts executing
i:fail   => j # trigger j if i fails
            """
```

A common use of this is to automate recovery from known modes of failure:

```
    graph = "goodbye:fail => really_goodbye"
```

i.e. if task `goodbye` fails, trigger another task that (presumably) really says goodbye.

Failure triggering generally requires use of *suicide triggers* as well, to remove the recovery task
if it isn't required (otherwise it would hang about indefinitely in the waiting state):

```
[scheduling]
    [[dependencies]]
        graph = """hello => goodbye
            goodbye:fail => really_goodbye
          goodbye => !really_goodbye # suicide"""
```

This means if `goodbye` fails, trigger `really_goodbye`; and otherwise, if `goodbye` succeeds, remove
`really_goodbye` from the suite.

Try running `tut/oneoff/suicide`, which also configures the `hello` task's runtime to make it fail,
to see how this works.

- For more on suite dependency graphs see 9.3.
- For more on task triggering see 9.3.5.

7.18   Runtime Inheritance

suite: `tut/oneoff/inherit`

The `[runtime]` section is actually a *multiple inheritance* hierarchy. Each subsection is a *namespace* that represents a task, or if it is inherited by other namespaces, a *family*. This allows common configuration to be factored out of related tasks very efficiently.

```
title = "Simple runtime inheritance example"
[scheduling]
    [[dependencies]]
        graph = "hello => goodbye"
[runtime]
    [[root]]
        script = "sleep 10; echo $GREETING World!"
    [[hello]]
        [[[environment]]]
            GREETING = Hello
    [[goodbye]]
        [[[environment]]]
            GREETING = Goodbye
```

The `[root]` namespace provides defaults for all tasks in the suite. Here both tasks inherit `script` from `root`, which they customize with different values of the environment variable `$GREETING`. Note that inheritance from `root` is implicit; from other parents an explicit `inherit = PARENT` is required, as shown below.

- For more on runtime inheritance, see 9.4.

7.19   Triggering Families

suite: `tut/oneoff/ftrigger1`

Task families defined by runtime inheritance can also be used as shorthand in graph trigger expressions. To see this, consider two "greeter" tasks that trigger off another task `foo`:

```
[scheduling]
    [[dependencies]]
        graph = "foo => greeter_1 & greeter_2"
```

If we put the common greeting functionality of `greeter_1` and `greeter_2` into a special `GREETERS` family, the graph can be expressed more efficiently like this:

```
[scheduling]
    [[dependencies]]
        graph = "foo => GREETERS"
```

i.e. if `foo` succeeds, trigger all members of `GREETERS` at once. Here's the full suite with runtime hierarchy shown:

```
title = "Triggering a family of tasks"
[scheduling]
    [[dependencies]]
        graph = "foo => GREETERS"
[runtime]
    [[root]]
        pre-script = "sleep 10"
    [[foo]]
        # empty (creates a dummy task)
    [[GREETERS]]
        script = "echo $GREETING World!"
```

```
[[greeter_1]]
    inherit = GREETERS
    [[[environment]]]
        GREETING = Hello
[[greeter_2]]
    inherit = GREETERS
    [[[environment]]]
        GREETING = Goodbye
```

(Note that we recommend given ALL-CAPS names to task families to help distinguish them from task names. However, this is just a convention).

Experiment with the `tut/oneoff/ftrigger1` suite to see how this works.

7.20   Triggering Off Of Families

suite: `tut/oneoff/ftrigger2`

Tasks (or families) can also trigger *off* other families, but in this case we need to specify what the trigger means in terms of the upstream family members. Here's how to trigger another task `bar` if all members of `GREETERS` succeed:

```
[scheduling]
    [[dependencies]]
        graph = """foo => GREETERS
            GREETERS:succeed-all => bar"""
```

Verbose validation in this case reports:

```
$ cylc val -v tut/oneoff/ftrigger2
...
Graph line substitutions occurred:
  IN: GREETERS:succeed-all => bar
  OUT: greeter_1:succeed & greeter_2:succeed => bar
...
```

Cylc ignores family member qualifiers like `succeed-all` on the right side of a trigger arrow, where they don't make sense, to allow the two graph lines above to be combined in simple cases:

```
[scheduling]
    [[dependencies]]
        graph = "foo => GREETERS:succeed-all => bar"
```

Any task triggering status qualified by `-all` or `-any`, for the members, can be used with a family trigger. For example, here's how to trigger `bar` if all members of `GREETERS` finish (succeed or fail) and any of them them succeed:

```
[scheduling]
    [[dependencies]]
        graph = """foo => GREETERS
    GREETERS:finish-all & GREETERS:succeed-any => bar"""
```

(use of `GREETERS:succeed-any` by itself here would trigger `bar` as soon as any one member of `GREETERS` completed successfully). Verbose validation now begins to show how family triggers can simplify complex graphs, even for this tiny two-member family:

```
$ cylc val -v tut/oneoff/ftrigger2
...
Graph line substitutions occurred:
  IN: GREETERS:finish-all & GREETERS:succeed-any => bar
  OUT: ( greeter_1:succeed | greeter_1:fail ) & \
       ( greeter_2:succeed | greeter_2:fail ) & \
```

```
        ( greeter_1:succeed | greeter_2:succeed ) => bar
...
```

Experiment with `tut/oneoff/ftrigger2` to see how this works.

- For more on family triggering, see 9.3.5.9.

7.21   Suite Visualization

You can style dependency graphs with an optional `[visualization]` section, as shown in `tut/oneoff/ftrigger2`:

```
[visualization]
    default node attributes = "style=filled"
    [[node attributes]]
        foo = "fillcolor=#6789ab", "color=magenta"
        GREETERS = "fillcolor=#ba9876"
        bar = "fillcolor=#89ab67"
```

To display the graph in an interactive viewer:

```
$ cylc graph tut/oneoff/ftrigger2 &    # dependency graph
$ cylc graph -n tut/oneoff/ftrigger2 & # runtime inheritance graph
```

It should look like Figure 15 (with the GREETERS family node expanded on the right).

Graph styling can be applied to entire families at once, and custom "node groups" can also be defined for non-family groups.

7.22   External Task Scripts

suite: `tut/oneoff/external`

The tasks in our examples so far have all had inlined implementation, in the suite definition, but real tasks often need to call external commands, scripts, or executables. To try this, let's return to the basic Hello World suite and cut the implementation of the task `hello` out to a file `hello.sh` in the suite bin directory:

```
#!/bin/sh

set -e

GREETING=${GREETING:-Goodbye}
echo "$GREETING World! from $0"
```

Make the task script executable, and change the `hello` task runtime section to invoke it:

```
title = "Hello World! from an external task script"
[scheduling]
    [[dependencies]]
        graph = "hello"
[runtime]
    [[hello]]
        pre-script = sleep 10
        script = hello.sh
        [[[environment]]]
            GREETING = Hello
```
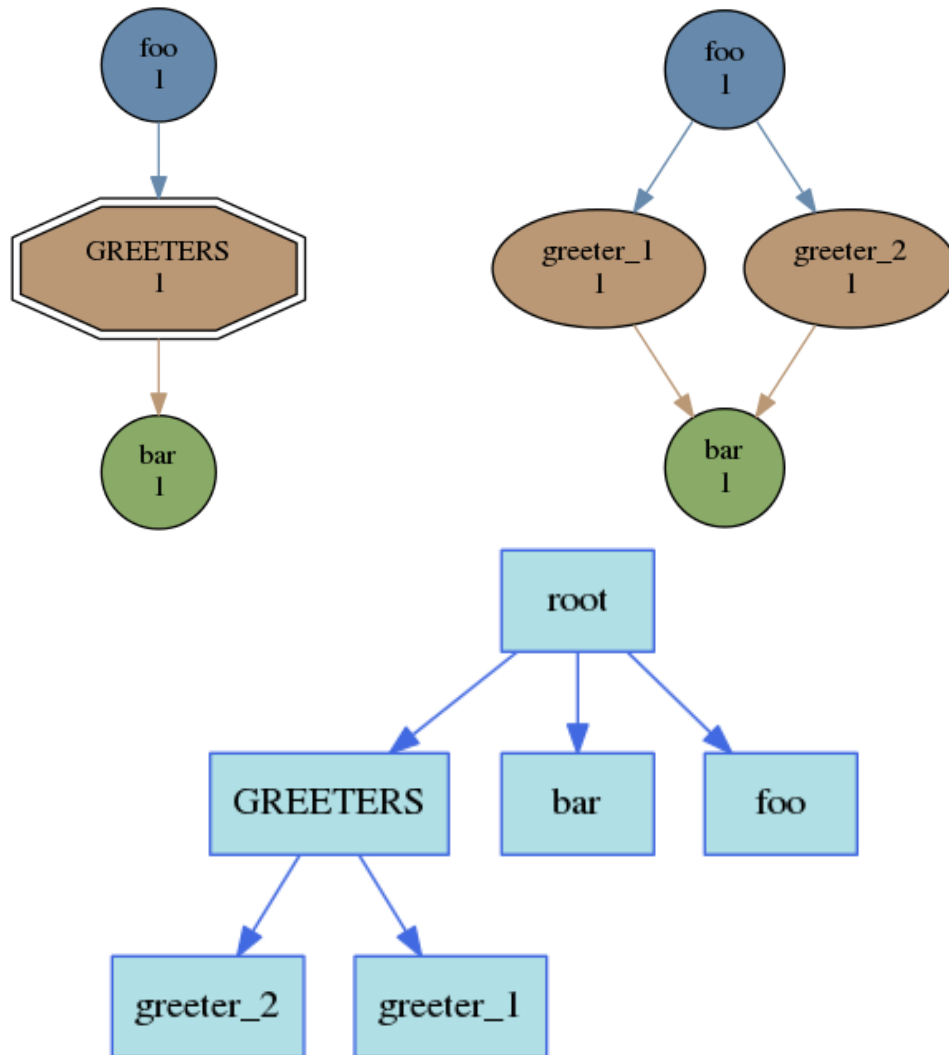
Figure 15: The *tut/oneoff/ftrigger2* dependency and runtime inheritance graphs

If you run the suite now the new greeting from the external task script should appear in the `hello` task stdout log. This works because cylc automatically adds the suite bin directory to `$PATH` in the environment passed to tasks via their job scripts. To execute scripts (etc.) located elsewhere you can refer to the file by its full file path, or set `$PATH` appropriately yourself (this could be done via `$HOME/.profile`, which is sourced at the top of the task job script, or in the suite definition itself).

Note the use of `set -e` above to make the script abort on error. This allows the error trapping code in the task job script to automatically detect unforeseen errors.

7.23   Cycling Tasks

suite: `tut/cycling/one`

So far we've considered non-cycling tasks, which finish without spawning a successor.

Cycling is based around iterating through date-time or integer sequences. A cycling task may run at each cycle point in a given sequence (cycle). For example, a sequence might be a set of date-times every 6 hours starting from a particular date-time. A cycling task may run for each date-time item (cycle point) in that sequence.

There may be multiple instances of this type of task running in parallel, if the opportunity arises and their dependencies allow it. Alternatively, a sequence can be defined with only one valid cycle point - in that case, a task belonging to that sequence may only run once.

Open the `tut/cycling/one` suite:

```
title = "Two cycling tasks, no inter-cycle dependence"
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20130808T00
    final cycle point = 20130812T00
    [[dependencies]]
        [[[T00,T12]]] # 00 and 12 hours UTC every day
            graph = "foo => bar"
[visualization]
    initial cycle point = 20130808T00
    final cycle point = 20130809T00
    [[node attributes]]
        foo = "color=red"
        bar = "color=blue"
```

The difference between cycling and non-cycling suites is all in the `[scheduling]` section, so we will leave the `[runtime]` section alone for now (this will result in cycling dummy tasks). Note that the graph is now defined under a new section heading that makes each task under it have a succession of cycle points ending in 00 or 12 hours, between specified initial and final cycle points (or indefinitely if no final cycle point is given), as shown in Figure 16.

If you run this suite instances of `foo` will spawn in parallel out to the *runahead limit*, and each `bar` will trigger off the corresponding instance of `foo` at the same cycle point. The runahead limit, which defaults to a few cycles but is configurable, prevents uncontrolled spawning of cycling tasks in suites that are not constrained by clock triggers in real time operation.

Experiment with `tut/cycling/one` to see how cycling tasks work.
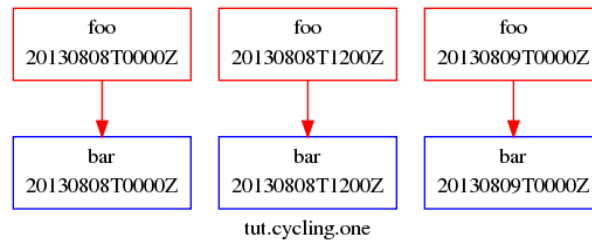
Figure 16: The `tut/cycling/one` suite

7.23.0.1 ISO 8601 Date-Time Syntax

The suite above is a very simple example of a cycling date-time workflow. More generally, cylc comprehensively supports the ISO 8601 standard for date-time instants, intervals, and sequences. Cycling graph sections can be specified using full ISO 8601 recurrence expressions, but these may be simplified by assuming context information from the suite - namely initial and final cycle points. One form of the recurrence syntax looks like `Rn/start-date-time/period` (`Rn` means run `n` times). In the example above, if the initial cycle point is always at 00 or 12 hours then `[[[T00,T12]]]` could be written as `[[[PT12H]]]`, which is short for `[[[R/initial-cycle-point/PT12H/]]]` - i.e. run every 12 hours indefinitely starting at the initial cycle point. It is possible to add constraints to the suite to only allow initial cycle points at 00 or 12 hours e.g.

```
[scheduling]
    initial cycle point = 20130808T00
    initial cycle point constraints = T00, T12
```

- For a comprehensive description of ISO 8601 based date-time cycling, see 9.3.4.5
- For more on runahead limiting in cycling suites, see 12.12.

7.23.1 Inter-Cycle Triggers

suite: `tut/cycling/two`

The `tut/cycling/two` suite adds inter-cycle dependence to the previous example:

```
[scheduling]
    [[dependencies]]
        # Repeat with cycle points of 00 and 12 hours every day:
        [[[T00,T12]]]
            graph = "foo[-PT12H] => foo => bar"
```

For any given cycle point in the sequence defined by the cycling graph section heading, `bar` triggers off `foo` as before, but now `foo` triggers off its own previous instance `foo[-PT12H]`. Date-time offsets in inter-cycle triggers are expressed as ISO 8601 intervals (12 hours in this case). Figure 17 shows how this connects the cycling graph sections together.

Experiment with this suite to see how inter-cycle triggers work. Note that the first instance of `foo`, at suite start-up, will trigger immediately in spite of its inter-cycle trigger, because cylc ignores dependence on points earlier than the initial cycle point. However, the presence of an inter-cycle trigger usually implies something special has to happen at start-up. If a model depends on its own previous instance for restart files, for example, then some special process has to generate the initial set of restart files when there is no previous cycle point to do it. The following section shows one way to handle this in cylc suites.
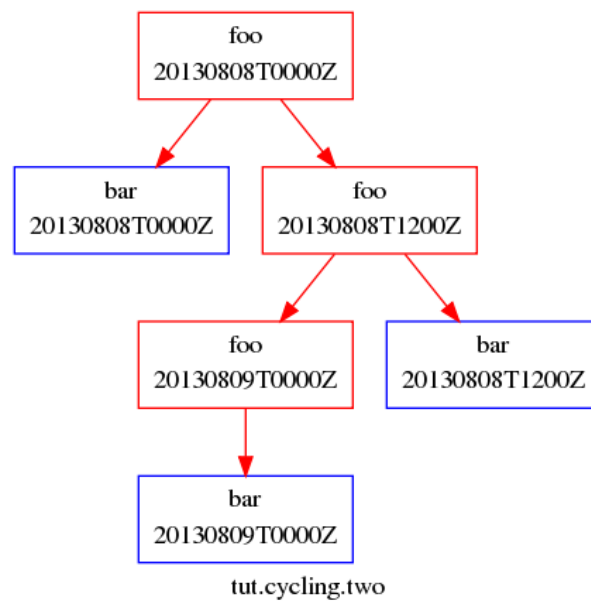
Figure 17: The `tut/cycling/two` suite

7.23.2   Initial Non-Repeating (R1) Tasks

suite: `tut/cycling/three`

Sometimes we want to be able to run a task at the initial cycle point, but refrain from running it in subsequent cycles. We can do this by writing an extra set of dependencies that are only valid at a single date-time cycle point. If we choose this to be the initial cycle point, these will only apply at the very start of the suite.

The cylc syntax for writing this single date-time cycle point occurrence is `R1`, which stands for `R1/no-specified-date-time/no-specified-period`. This is an adaptation of part of the ISO 8601 date-time standard's recurrence syntax (`Rn/date-time/period`) with some special context information supplied by cylc for the `no-specified-*` data.

The `1` in the `R1` means run once. As we've specified no date-time, Cylc will use the initial cycle point date-time by default, which is what we want. We've also missed out specifying the period - this is set by cylc to a zero amount of time in this case (as it never repeats, this is not significant).

For example, in `tut/cycling/three`:

```
[cylc]
    cycle point time zone = +13
[scheduling]
    initial cycle point = 20130808T00
    final cycle point = 20130812T00
    [[dependencies]]
        [[[R1]]]
            graph = "prep => foo"
        [[[T00,T12]]]
            graph = "foo[-PT12H] => foo => bar"
```

This is shown in Figure 18.
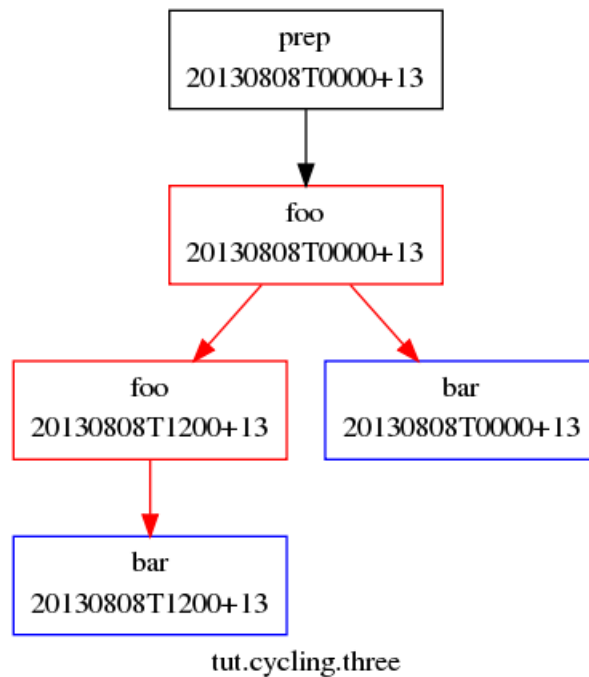
Figure 18: The `tut/cycling/three` suite

Note that the time zone has been set to `+1300` in this case, instead of UTC (`Z`) as before. If no time zone or UTC mode was set, the local time zone of your machine will be used in the cycle points.

At the initial cycle point, `foo` will depend on `foo[-PT12H]` and also on `prep`:

```
prep.20130808T0000+13 & foo.20130807T1200+13 => foo.20130808T0000+13
```

Thereafter, it will just look like e.g.:

```
foo.20130808T0000+13 => foo.20130808T1200+13
```

However, in our initial cycle point example, the dependence on `foo.20130807T1200+13` will be ignored, because that task's cycle point is earlier than the suite's initial cycle point and so it cannot run. This means that the initial cycle point dependencies for `foo` actually look like:

```
prep.20130808T0000+13 => foo.20130808T0000+13
```

- `R1` tasks can also be used to make something special happen at suite shutdown, or at any single cycle point throughout the suite run. For a full primer on cycling syntax, see 9.3.4.5.

7.23.3 Integer Cycling

suite: `tut/cycling/integer`

Cylc can do also do integer cycling for repeating workflows that are not date-time based.

Open the `tut/cycling/integer` suite, which is plotted in Figure 19.
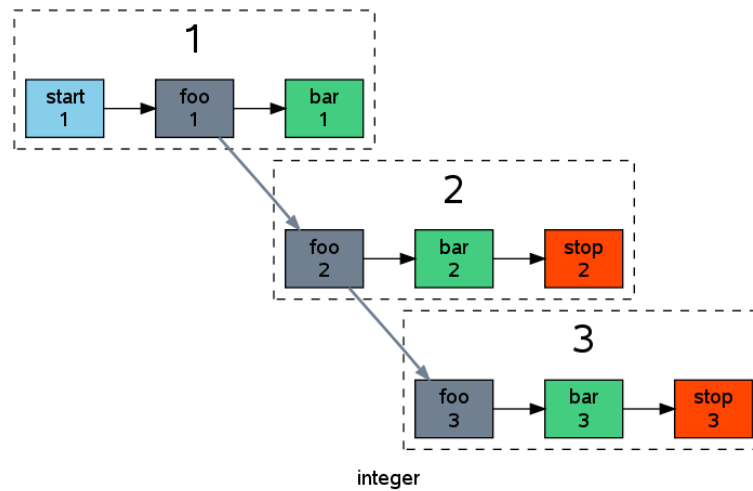
Figure 19: The `tut/cycling/integer` suite

```
[scheduling]
    cycling mode = integer
    initial cycle point = 1
    final cycle point = 3
    [[dependencies]]
        [[[R1]]] # = R1/1/?
            graph = start => foo
        [[[P1]]] # = R/1/P1
            graph = foo[-P1] => foo => bar
        [[[R2/P1]]] # = R2/P1/3
            graph = bar => stop

[visualization]
    [[node attributes]]
        start = "style=filled", "fillcolor=skyblue"
        foo = "style=filled", "fillcolor=slategray"
        bar = "style=filled", "fillcolor=seagreen3"
        stop = "style=filled", "fillcolor=orangered"
```

The integer cycling notation is intended to look similar to the ISO 8601 date-time notation, but it is simpler for obvious reasons. The example suite illustrates two recurrence forms, `Rn/start-point/period` and `Rn/period/stop-point`, simplified somewhat using suite context information (namely the initial and final cycle points). The first form is used to run one special task called `start` at start-up, and for the main cycling body of the suite; and the second form to run another special task called `stop` in the final two cycles. The `P` character denotes period (interval) just like in the date-time notation. `R/1/P2` would generate the sequence of points `1,3,5,...`.

- For more on integer cycling, including a more realistic usage example see 9.3.4.7.

7.24   Jinja2

suite: `tut/oneoff/jinja2`

Cylc has built in support for the Jinja2 template processor, which allows us to embed code in suite definitions to generate the final result seen by cylc.

The `tut/oneoff/jinja2` suite illustrates two common uses of Jinja2: changing suite content or

structure based on the value of a logical switch; and iteratively generating dependencies and runtime configuration for groups of related tasks:

```jinja2
#!jinja2

{% set MULTI = True %}
{% set N_GOODBYES = 3 %}

title = "A Jinja2 Hello World! suite"
[scheduling]
    [[dependencies]]
{% if MULTI %}
        graph = "hello => BYE"
{% else %}
        graph = "hello"
{% endif %}

[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
{% if MULTI %}
    [[BYE]]
        script = "sleep 10; echo Goodbye World!"
    {% for I in range(0,N_GOODBYES) %}
    [[ goodbye_{{I}} ]]
        inherit = BYE
    {% endfor %}
{% endif %}
```

To view the result of Jinja2 processing with the Jinja2 flag `MULTI` set to `False`:

```
$ cylc view --jinja2 --stdout tut/oneoff/jinja2

title = "A Jinja2 Hello World! suite"
[scheduling]
    [[dependencies]]
        graph = "hello"
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
```

And with `MULTI` set to `True`:

```
$ cylc view --jinja2 --stdout tut/oneoff/jinja2

title = "A Jinja2 Hello World! suite"
[scheduling]
    [[dependencies]]
        graph = "hello => BYE"
[runtime]
    [[hello]]
        script = "sleep 10; echo Hello World!"
    [[BYE]]
        script = "sleep 10; echo Goodbye World!"
    [[ goodbye_0 ]]
        inherit = BYE
    [[ goodbye_1 ]]
        inherit = BYE
    [[ goodbye_2 ]]
        inherit = BYE
```

7.25 Task Retry On Failure

suite: `tut/oneoff/retry`

Tasks can be configured to retry a number of times if they fail. An environment variable `$CYLC_TASK_TRY_NUMBER` increments from 1 on each successive try, and is passed to the task to allow different behaviour on the retry:

```
title = "A task with automatic retry on failure"
[scheduling]
    [[dependencies]]
        graph = "hello"
[runtime]
    [[hello]]
        script = """
sleep 10
if [[ $CYLC_TASK_TRY_NUMBER < 3 ]]; then
    echo "Hello ... aborting!"
    exit 1
else
    echo "Hello World!"
fi"""
        [[[job]]]
            execution retry delays = 2*PT6S # retry twice after 6-second delays
```

When a task with configured retries fails, its cylc task proxy goes into the *retrying* state until the next retry delay is up, then it resubmits. It only enters the *failed* state on a final definitive failure.

Experiment with `tut/oneoff/retry` to see how this works.

7.26 Other Users' Suites

If you have read access to another user's account (even on another host) it is possible to use `cylc monitor` to look at their suite's progress without full shell access to their account. To do this, you will need to copy their suite passphrase to

    `$HOME/.cylc/SUITE_OWNER@SUITE_HOST/SUITE_NAME/passphrase`

(use of the host and owner names is optional here - see 12.6.1) *and* also retrieve the port number of the running suite from:

    `~SUITE_OWNER/cylc-run/SUITE_NAME/.service/contact`

Once you have this information, you can run

`$ cylc monitor --user=SUITE_OWNER --port=SUITE_PORT SUITE_NAME`

to view the progress of their suite.

Other suite-connecting commands work in the same way; see 12.10.

7.27 Other Things To Try

Almost every feature of cylc can be tested quickly and easily with a simple dummy suite. You can write your own, or start from one of the example suites in `/path/to/cylc/examples` (see use of `cylc import-examples` above) - they all run "out the box" and can be copied and modified at will.

- Change the suite runahead limit in a cycling suite.
- Stop a suite mid-run with `cylc stop`, and restart it again with `cylc restart`.

- Hold (pause) a suite mid-run with `cylc hold`, then modify the suite definition and `cylc reload` it before using `cylc release` to continue (you can also reload without holding).
- Use the gcylc View menu to show the task state color key and watch tasks in the `task-states` example evolve as the suite runs.
- Manually re-run a task that has already completed or failed, with `cylc trigger`.
- Use an *internal queue* to prevent more than an alotted number of tasks from running at once even though they are ready - see 12.13.
- Configure task event hooks to send an email, or shut the suite down, on task failure.

8   Suite Name Registration

Cylc commands target suites via their names, which are relative path names under the suite run directory (`~/cylc-run/` by default). Suites can be grouped together under sub-directories. E.g.:

```
$ cylc print -t nwp
nwp
 |-oper
 | |-region1  Local Model Region1      /home/oliverh/cylc-run/nwp/oper/region1
 | '-region2  Local Model Region2      /home/oliverh/cylc-run/nwp/oper/region2
 '-test
   '-region1  Local Model TEST Region1 /home/oliverh/cylc-run/nwp/test/region1
```

Suites can be pre-registered with a name using the `cylc register` command. The creates the essential directory structure for the suite, and generates some service files underneath it. Otherwise, `cylc run` will create these files on suite start up.

9   Suite Definition

Cylc suites are defined in structured, validated, *suite.rc* files that concisely specify the properties of, and the relationships between, the various tasks managed by the suite. This section of the User Guide deals with the format and content of the suite.rc file, including task definition. Task implementation - what's required of the real commands, scripts, or programs that do the processing that the tasks represent - is covered in 10; and task job submission - how tasks are submitted to run - is in 11.

9.1   Suite Definition Directories

A cylc *suite definition directory* contains:

- **A suite.rc file**: this is the suite definition.
  - And any include-files used in it (see below; may be kept in sub-directories).
- **A `bin/` sub-directory** (optional)
  - For scripts and executables that implement, or are used by, suite tasks.
  - Automatically added to `$PATH` in task execution environments.
  - Alternatively, tasks can call external commands, scripts, or programs; or they can be scripted entirely within the suite.rc file.
- **A `lib/python/` sub-directory** (optional)

– For custom job submission modules (see 11.7) and local Python modules imported by custom Jinja2 filters (see 9.7.2).
- **Any other sub-directories and files** - documentation, control files, etc. (optional)
  – Holding everything in one place makes proper suite revision control possible.
  – Portable access to files here, for running tasks, is provided through `$CYLC_SUITE_DEF_PATH` (see 9.4.7).
  – Ignored by cylc, but the entire suite definition directory tree is copied when you copy a suite using cylc commands.

A typical example:

```
/path/to/my/suite    # suite definition directory
    suite.rc            # THE SUITE DEFINITION FILE
    bin/                # scripts and executables used by tasks
        foo.sh
        bar.sh
        ...
    # (OPTIONAL) any other suite-related files, for example:
    inc/                # suite.rc include-files
        nwp-tasks.rc
        globals.rc
        ...
    doc/                # documentation
    control/            # control files
    ancil/              # ancillary files
    ...
```

9.2   Suite.rc File Overview

Suite.rc files are an extended-INI format with section nesting.

Embedded template processor expressions may also be used in the file, to programatically generate the final suite definition seen by cylc. Currently the Jinja2 template processor is supported (http://jinja.pocoo.org/docs); see 9.7 for examples. In the future cylc may provide a plug-in interface to allow use of other template engines too.

9.2.1   Syntax

The following defines legal suite.rc syntax:

- **Items** are of the form `item = value`.
- [**Section**] headings are enclosed in square brackets.
- **Sub-section** [[**nesting**]] is defined by repeated square brackets.
- Sections are **closed** by the next section heading.
- **Comments** (line and trailing) follow a hash character: #
- **List values** are comma-separated.
- **Single-line string values** can be single-, double-, or un-quoted.
- **Multi-line string values** are triple-quoted (using single or double quote characters).
- **Boolean values** are capitalized: True, False.
- **Leading and trailing whitespace** is ignored.
- **Indentation** is optional but should be used for clarity.
- **Continuation lines** follow a trailing backslash: \
- **Duplicate sections** add their items to those previously defined under the same section.

- **Duplicate items** override, *except for dependency `graph` strings, which are additive.*
- **Include-files** `%include inc/foo.rc` can be used as a verbatim inlining mechanism.

Suites that embed Jinja2 code (see 9.7) must process to raw suite.rc syntax.

9.2.2   Include-Files

Cylc has native support for suite.rc include-files, which may help to organize large suites. Inclusion boundaries are completely arbitrary - you can think of include-files as chunks of the suite.rc file simply cut-and-pasted into another file. Include-files may be included multiple times in the same file, and even nested. Include-file paths can be specified portably relative to the suite definition directory, e.g.:

```
# include the file $CYLC_SUITE_DEF_PATH/inc/foo.rc:
%include inc/foo.rc
```

9.2.2.1   Editing Temporarily Inlined Suites

Cylc's native file inclusion mechanism supports optional inlined editing:

```
$ cylc edit --inline SUITE
```

The suite will be split back into its constituent include-files when you exit the edit session. While editing, the inlined file becomes the official suite definition so that changes take effect whenever you save the file. See `cylc prep edit --help` for more information.

9.2.2.2   Include-Files via Jinja2

Jinja2 (9.7) also has template inclusion functionality.

9.2.3   Syntax Highlighting For Suite Definitions

Cylc comes with syntax files for a number of text editors:

```
$CYLC_DIR/conf/cylc.vim      # vim
$CYLC_DIR/conf/cylc-mode.el  # emacs
$CYLC_DIR/conf/cylc.lang     # gedit (and other gtksourceview programs)
$CYLC_DIR/conf/cylc.xml      # kate
```

Refer to comments at the top of each file to see how to use them.

9.2.4   Gross File Structure

Cylc suite.rc files consist of a suite title and description followed by configuration items grouped under several top level section headings:

- **[cylc]** - *non task-specific suite configuration*
- **[scheduling]** - *determines when tasks are ready to run*
  - tasks with special behaviour, e.g. clock-trigger tasks
  - the dependency graph, which defines the relationships between tasks
- **[runtime]** - *determines how, where, and what to execute when tasks are ready*

- script, environment, job submission, remote hosting, etc.
- suite-wide defaults in the *root* namespace
- a nested family hierarchy with common properties inherited by related tasks
- [**visualization**] - suite graph styling

9.2.5 Validation

Cylc suite.rc files are automatically validated against a specification that defines all legal entries, values, options, and defaults. This detects formatting errors, typographic errors, illegal items and illegal values prior to run time. Some values are complex strings that require further parsing by cylc to determine their correctness (this is also done during validation). All legal entries are documented in the *Suite.rc Reference* (A).

The validator reports the line numbers of detected errors. Here's an example showing a section heading with a missing right bracket:

```
$ cylc validate my.suite
    [[special tasks]
'Section bracket mismatch, line 19'
```

If the suite.rc file uses include-files `cylc view` will show an inlined copy of the suite with correct line numbers (you can also edit suites in a temporarily inlined state with `cylc edit --inline`).

Validation does not check the validity of chosen batch systems.

9.3 Scheduling - Dependency Graphs

The `[scheduling]` section of a suite.rc file defines the relationships between tasks in a suite - the information that allows cylc to determine when tasks are ready to run. The most important component of this is the suite dependency graph. Cylc graph notation makes clear textual graph representations that are very concise because sections of the graph that repeat at different hours of the day, say, only have to be defined once. Here's an example with dependencies that vary depending on the particular cycle point:

```
[scheduling]
    initial cycle point = 20200401
    final cycle point = 20200405
    [[dependencies]]
        [[[T00,T06,T12,T18]]] # validity (hours)
            graph = """
A => B & C   # B and C trigger off A
A[-PT6H] => A  # Model A restart trigger
                """
        [[[T06,T18]]] # hours
            graph = "C => X"
```

Figure 20 shows the complete suite.rc listing alongside the suite graph. This is a complete, valid, runnable suite (it will use default task runtime properties such as `script`).

9.3.1 Graph String Syntax

Multiline graph strings may contain:

- **blank lines**

```
title = "Dependency Example 1"
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20200401
    final cycle point = 20200405
    [[dependencies]]
        [[[T00,T06,T12,T18]]] # validity
            graph = """
A => B & C   # B and C trigger off A
A[-PT6H] => A
# Model A restart trigger
            """
        [[[T06,T18]]] # hours
            graph = "C => X"
[visualization]
    initial cycle point = 20200401
    final cycle point = 20200401T06
    [[node attributes]]
        X = "color=red"
```
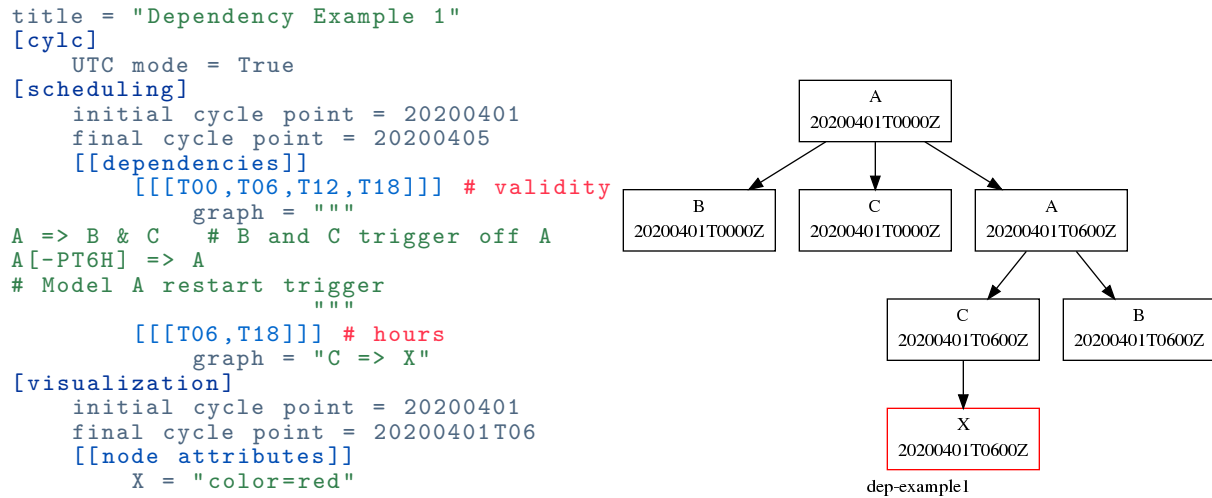
Figure 20: Example Suite

- **arbitrary white space**
- **internal comments:** following the `#` character
- **conditional task trigger expressions** - see below.

9.3.2   Interpreting Graph Strings

Suite dependency graphs can be broken down into pairs in which the left side (which may be a single task or family, or several that are conditionally related) defines a trigger for the task or family on the right. For instance the "word graph" *C triggers off B which triggers off A* can be deconstructed into pairs *C triggers off B* and *B triggers off A*. In this section we use only the default trigger type, which is to trigger off the upstream task succeeding; see 9.3.5 for other available triggers.

In the case of cycling tasks, the triggers defined by a graph string are valid for cycle points matching the list of hours specified for the graph section. For example this graph:

```
[scheduling]
    [[dependencies]]
        [[[T00,T12]]]
            graph = "A => B"
```

implies that B triggers off A for cycle points in which the hour matches 00 or 12.

To define inter-cycle dependencies, attach an offset indicator to the left side of a pair:

```
[scheduling]
    [[dependencies]]
        [[[T00,T12]]]
            graph = "A[-PT12H] => B"
```

This means B[time] triggers off A[time-PT12H] (12 hours before) for cycle points with hours matching 00 or 12. *time* is implicit because this keeps graphs clean and concise, given that the majority of tasks will typically depend only on others with the same cycle point. Cycle point offsets can only appear on the left of a pair, because a pairs define triggers for the right task at cycle point *time*. However, `A => B[-PT6H]`, which is illegal, can be reformulated as a *future*

trigger `A[+PT6H] => B` (see 9.3.5.11). It is also possible to combine multiple offsets within a cycle point offset e.g.

```
[scheduling]
    [[dependencies]]
        [[[T00,T12]]]
            graph = "A[-P1D-PT12H] => B"
```

This means that B[Time] triggers off A[time-P1D-PT12H] (1 day and 12 hours before).

Triggers can be chained together. This graph:

```
graph = """A => B  # B triggers off A
           B => C  # C triggers off B"""
```

is equivalent to this:

```
graph = "A => B => C"
```

Each trigger in the graph must be unique but *the same task can appear in multiple pairs or chains.* Separately defined triggers for the same task have an AND relationship. So this:

```
graph = """A => X  # X triggers off A
           B => X  # X also triggers off B"""
```

is equivalent to this:

```
graph = "A & B => X"  # X triggers off A AND B
```

In summary, the branching tree structure of a dependency graph can be partitioned into lines (in the suite.rc graph string) of pairs or chains, in any way you like, with liberal use of internal white space and comments to make the graph structure as clear as possible.

```
# B triggers if A succeeds, then C and D trigger if B succeeds:
    graph = "A => B => C & D"
# which is equivalent to this:
    graph = """A => B => C
               B => D"""
# and to this:
    graph = """A => B => D
               B => C"""
# and to this:
    graph = """A => B
               B => C
               B => D"""
# and it can even be written like this:
    graph = """A => B # blank line follows:

               B => C # comment ...
               B => D"""
```

9.3.2.1 Splitting Up Long Graph Lines

It is not necessary to use the general line continuation marker \ to split long graph lines. Just break at dependency arrows, or split long chains into smaller ones. This graph:

```
graph = "A => B => C"
```

is equivalent to this:

```
graph = """A => B =>
               C"""
```

and also to this:

```
title = some one-off tasks
[scheduling]
    [[dependencies]]
        graph = "foo => bar & baz => qux"
```
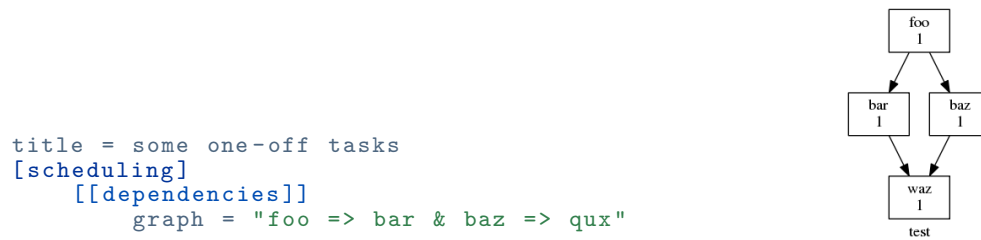
Figure 21: One-off (Non-Cycling) Tasks.

```
title = some cycling tasks
# (no dependence between cycle points)
[scheduling]
    [[dependencies]]
        [[[T00,T12]]]
            graph = "foo => bar & baz =>
```
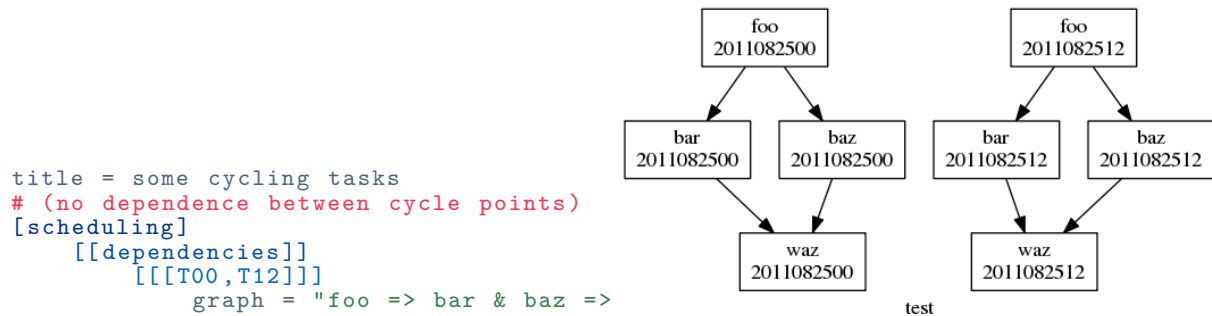
Figure 22: Cycling Tasks.

```
    graph = """A => B
               B => C"""
```

9.3.3   Graph Types

A suite definition can contain multiple graph strings that are combined to generate the final graph.

9.3.3.1   One-off (Non-Cycling)

Figure 21 shows a small suite of one-off non-cycling tasks; these all share a single cycle point (`1`) and don't spawn successors (once they're all finished the suite just exits). The integer `1` attached to each graph node is just an arbitrary label here.

9.3.3.2   Cycling Graphs

For cycling tasks the graph section heading defines a sequence of cycle points for which the subsequent graph section is valid. Figure 22 shows a small suite of cycling tasks.

9.3.4   Graph Section Headings

Graph section headings define recurrence expressions, the graph within a graph section heading defines a workflow at each point of the recurrence. For example in the following scenario:

```
[scheduling]
    [[dependencies]]
```

```
        [[[ T06 ]]]  # A graph section heading
          graph = foo => bar
```

T06 means "Run every day starting at 06:00 after the initial cycle point". Cylc allows you to start (or end) at any particular time, repeat at whatever frequency you like, and even optionally limit the number of repetitions.

Graph section heading can also be used with integer cycling see 9.3.4.7.

9.3.4.1 Syntax Rules

Date-time cycling information is made up of a starting *date-time*, an *interval*, and an optional *limit*.

The time is assumed to be in the local time zone unless you set `[cylc]cycle point time zone` or `[cylc]UTC mode`. The calendar is assumed to be the proleptic Gregorian calendar unless you set `[scheduling]cycling mode`.

The syntax for representations is based on the ISO 8601 date-time standard. This includes the representation of *date-time*, *interval*. What we define for cylc's cycling syntax is our own optionally-heavily-condensed form of ISO 8601 recurrence syntax. The most common full form is: `R[limit?]/[date-time]/[interval]`. However, we allow omitting information that can be guessed from the context (rules below). This means that it can be written as:

```
R[limit?]/[date-time]
R[limit?]//[interval]
[date-time]/[interval]
R[limit?] # Special limit of 1 case
[date-time]
[interval]
```

with example graph headings for each form being:

```
[[[ R5/T00 ]]]              # Run 5 times at 00:00 every day
[[[ R//PT1H ]]]             # Run every hour (Note the R// is redundant)
[[[ 20000101T00Z/P1D ]]]    # Run every day starting at 00:00 1st Jan 2000
[[[ R1 ]]]                  # Run once at the initial cycle point
[[[ 20000101T00Z ]]]        # Run once at 00:00 1st Jan 2000
[[[ P1Y ]]]                 # Run every year
```

Note that T00 is an example of `[date-time]`, with an inferred 1 day period and no limit.

Where some or all *date-time* information is omitted, it is inferred to be relative to the initial date-time cycle point. For example, T00 by itself would mean the next occurrence of midnight that follows, or is, the initial cycle point. Entering `+PT6H` would mean 6 hours after the initial cycle point. Entering `-P1D` would mean 1 day before the initial cycle point. Entering no information for the *date-time* implies the initial cycle point date-time itself.

Where the *interval* is omitted and some (but not all) *date-time* information is omitted, it is inferred to be a single unit above the largest given specific *date-time* unit. For example, the largest given specific unit in T00 is hours, so the inferred interval is 1 day (daily), `P1D`.

Where the *limit* is omitted, unlimited cycling is assumed. This will be bounded by the final cycle point's date-time if given.

Another supported form of ISO 8601 recurrence is: `R[limit?]/[interval]/[date-time]`. This form uses the *date-time* as the end of the cycling sequence rather than the start. For example, `R3/P5D/20140430T06` means:

```
20140420 T06
20140425 T06
20140430 T06
```

This kind of form can be used for specifying special behaviour near the end of the suite, at the final cycle point's date-time. We can also represent this in cylc with a collapsed form:

```
R[limit?]/[interval]
R[limit?]//[date-time]
[interval]/[date-time]
```

So, for example, you can write:

```
[[[ R1//+P0D ]]]    # Run once at the final cycle point
[[[ R5/P1D ]]]      # Run 5 times, every 1 day, ending at the final
                    # cycle point
[[[ P2W/T00 ]]]     # Run every 2 weeks ending at 00:00 following
                    # the final cycle point
[[[ R//T00 ]]]      # Run every 1 day ending at 00:00 following the
                    # final cycle point
```

9.3.4.2   Referencing The Initial And Final Cycle Points

For convenience the caret and dollar symbols may be used as shorthand for the initial and final cycle points. Using this shorthand you can write:

```
[[[ R1/^+PT12H ]]]  # Repeat once 12 hours after the initial cycle point
                    # R[limit]/[date-time]
                    # Equivalent to [[[ R1/+PT12H ]]]
[[[ R1/$ ]]]        # Repeat once at the final cycle point
                    # R[limit]/[date-time]
                    # Equivalent to [[[ R1//+P0D ]]]
[[[ $-P2D/PT3H ]]]  # Repeat 3 hourly starting two days before the
                    # [date-time]/[interval]
                    # final cycle point
```

Note that there can be multiple ways to write the same headings, for instance the following all run once at the final cycle point:

```
[[[ R1/P0Y ]]]      # R[limit]/[interval]
[[[ R1/P0Y/$ ]]]    # R[limit]/[interval]/[date-time]
[[[ R1/$ ]]]        # R[limit]/[date-time]
```

9.3.4.3   Excluding Dates

Datetimes can be excluded from a recurrence by an exclamation mark for example `[[[ PT1D!20000101 ]]]` means run daily except on the first of January 2000.

This syntax can only be used to exclude one datetime from a recurrence. Note that the `^` and `$` symbols (shorthand for the initial and final cycle points) are both datetimes so `[[[ T12!$-PT1D ]]]` is valid.

If using a run limit in combination with an exclusion, the heading might not run the number of times specified in the limit. For example in the following suite `foo` will only run once as its second run has been excluded.

```
[scheduling]
    initial cycle point = 20000101T00Z
    final cycle point = 20000105T00Z
    [[dependencies]]
        [[[ R2/P1D!20000102 ]]]
            graph = foo
```

9.3.4.4   How Multiple Graph Strings Combine

For a cycling graph with multiple validity sections for different hours of the day, the different sections *add* to generate the complete graph. Different graph sections can overlap (i.e. the same hours may appear in multiple section headings) and the same tasks may appear in multiple sections, but individual dependencies should be unique across the entire graph. For example, the following graph defines a duplicate prerequisite for task C:

```
[scheduling]
    [[dependencies]]
        [[[T00,T06,T12,T18]]]
            graph = "A => B => C"
        [[[T06,T18]]]
            graph = "B => C => X"
            # duplicate prerequisite: B => C already defined at T06, T18
```

This does not affect scheduling, but for the sake of clarity and brevity the graph should be written like this:

```
[scheduling]
    [[dependencies]]
        [[[T00,T06,T12,T18]]]
            graph = "A => B => C"
        [[[T06,T18]]]
            # X triggers off C only at 6 and 18 hours
            graph = "C => X"
```

9.3.4.5   Advanced Examples

The following examples show the various ways of writing graph headings in cylc.

```
[[[ R1 ]]]              # Run once at the initial cycle point
[[[ P1D ]]]             # Run every day starting at the initial cycle point
[[[ PT5M ]]]            # Run every 5 minutes starting at the initial cycle
                       # point
[[[ T00/P2W ]]]        # Run every 2 weeks starting at 00:00 after the
                       # initial cycle point
[[[ +P5D/P1M ]]]       # Run every month, starting 5 days after the initial
                       # cycle point
[[[ R1/T06 ]]]         # Run once at 06:00 after the initial cycle point
[[[ R1/P0Y ]]]         # Run once at the final cycle point
[[[ R1/$ ]]]           # Run once at the final cycle point (alternative
                       # form)
[[[ R1/$-P3D ]]]       # Run once three days before the final cycle point
[[[ R3/T0830 ]]]       # Run 3 times, every day at 08:30 after the initial
                       # cycle point
[[[ R3/01T00 ]]]       # Run 3 times, every month at 00:00 on the first
                       # of the month after the initial cycle point
[[[ R5/W-1/P1M ]]]     # Run 5 times, every month starting on Monday
                       # following the initial cycle point
[[[ T00!^ ]]]          # Run at the first occurrence of T00 that isn't the
                       # initial cycle point
[[[ PT1D!20000101 ]]]  # Run every day days excluding 1st Jan 2000
[[[ 20140201T06/P1D ]]]    # Run every day starting at 20140201T06
[[[ R1/min(T00,T06,T12,T18) ]]]  # Run once at the first instance
                                 # of either T00, T06, T12 or T18
                                 # starting at the initial cycle
                                 # point
```

9.3.4.6   Advanced Starting Up

Dependencies that are only valid at the initial cycle point can be written using the `R1` notation (e.g. as in 7.23.2. For example:

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20130808T00
    final cycle point = 20130812T00
    [[dependencies]]
        [[[R1]]]
            graph = "prep => foo"
        [[[T00]]]
            graph = "foo[-P1D] => foo => bar"
```

In the example above, `R1` implies `R1/20130808T00`, so `prep` only runs once at that cycle point (the initial cycle point). At that cycle point, `foo` will have a dependence on `prep` - but not at subsequent cycle points.

However, it is possible to have a suite that has multiple effective initial cycles - for example, one starting at `T00` and another starting at `T12`. What if they need to share an initial task?

Let's suppose that we add the following section to the suite example above:

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20130808T00
    final cycle point = 20130812T00
    [[dependencies]]
        [[[R1]]]
            graph = "prep => foo"
        [[[T00]]]
            graph = "foo[-P1D] => foo => bar"
        [[[T12]]]
            graph = "baz[-P1D] => baz => qux"
```

We'll also say that there should be a starting dependence between `prep` and our new task `baz` - but we still want to have a single `prep` task, at a single cycle.

We can write this using a special case of the `task[-interval]` syntax - if the interval is null, this implies the task at the initial cycle point.

For example, we can write our suite like 23.

This neatly expresses what we want - a task running at the initial cycle point that has one-off dependencies with other task sets at different cycles.

A different kind of requirement is displayed in Figure 24. Usually, we want to specify additional tasks and dependencies at the initial cycle point. What if we want our first cycle point to be entirely special, with some tasks missing compared to subsequent cycle points?

In Figure 24, `bar` will not be run at the initial cycle point, but will still run at subsequent cycle points. `[[[+PT6H/PT6H]]]` means start at `+PT6H` (6 hours after the initial cycle point) and then repeat every `PT6H` (6 hours).

Some suites may have staggered start-up sequences where different tasks need running once but only at specific cycle points, potentially due to differing data sources at different cycle points with different possible initial cycle points. To allow this cylc provides a `min( )` function that can be used as follows:

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20100101T03
    [[dependencies]]
        [[[R1/min(T00,T12)]]]
```

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20130808T00
    final cycle point = 20130812T00
    [[dependencies]]
        [[[R1]]]
            graph = "prep"
        [[[R1/T00]]]
# ^ implies the initial cycle point:
     graph = "prep[^] => foo"
        [[[R1/T12]]]
# ^ is initial cycle point, as above:
     graph = "prep[^] => baz"
        [[[T00]]]
    graph = "foo[-P1D] => foo => bar"
        [[[T12]]]
    graph = "baz[-P1D] => baz => qux"
[visualization]
    initial cycle point = 20130808T00
    final cycle point = 20130810T00
    [[node attributes]]
        foo = "color=red"
        bar = "color=orange"
        baz = "color=green"
        qux = "color=blue"
```
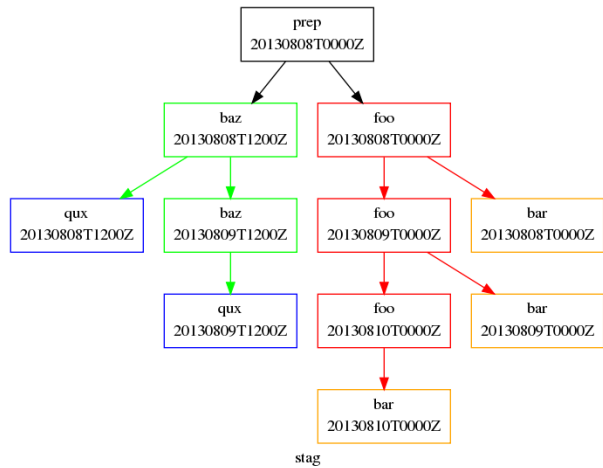

Figure 23: Staggered Start Suite

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20130808T00
    final cycle point = 20130808T18
    [[dependencies]]
        [[[R1]]]
            graph = "setup_foo => foo"
        [[[+PT6H/PT6H]]]
            graph = """
                foo[-PT6H] => foo
                foo => bar
            """
[visualization]
    initial cycle point = 20130808T00
    final cycle point = 20130808T18
    [[node attributes]]
        foo = "color=red"
        bar = "color=orange"
```
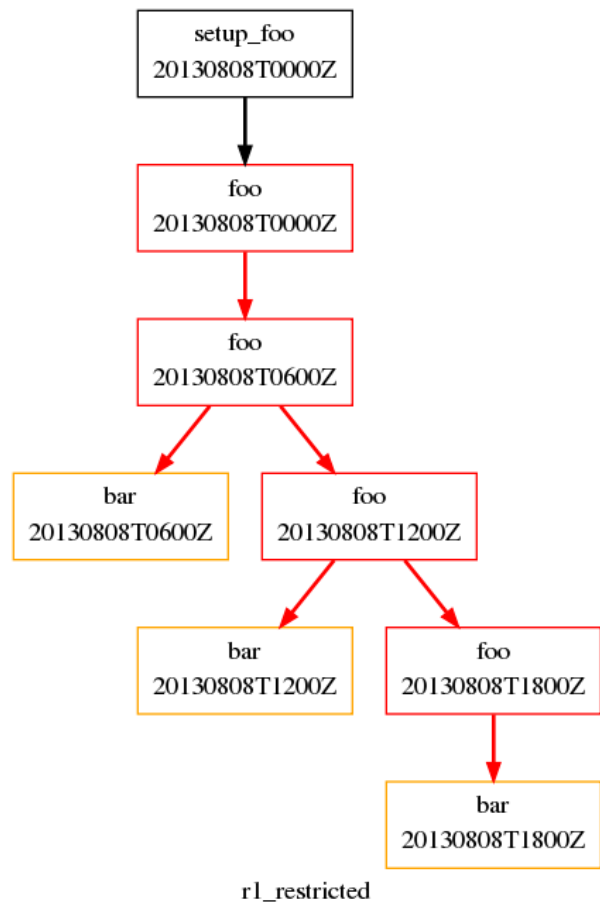

Figure 24:   Restricted First Cycle Point Suite

```
            graph = "prep1 => foo"
     [[[R1/min(T06,T18)]]]
            graph = "prep2 => foo"
     [[[T00,T06,T12,T18]]]
            graph = "foo => bar"
```

In this example the initial cycle point is `20100101T03`, so the `prep1` task will run once at `20100101T12` and the `prep2` task will run once at `20100101T06` as these are the first cycle points after the initial cycle point in the respective `min( )` entries.

9.3.4.7 Integer Cycling

In addition to non-repeating and date-time cycling workflows, cylc can do integer cycling for repeating workflows that are not date-time based.

To construct an integer cycling suite, set `[scheduling]cycling mode = integer`, and specify integer values for the initial and (optional) final cycle points. The notation for intervals, offsets, and recurrences (sequences) is similar to the date-time cycling notation, except for the simple integer values.

The full integer recurrence expressions supported are:

- `Rn/start-point/interval #e.g. R3/1/P2`
- `Rn/interval/end-point #e.g. R3/P2/9`

But, as for date-time cycling, sequence start and end points can be omitted where suite initial and final cycle points can be assumed. Some examples:

```
[[[ R1 ]]]          # Run once at the initial cycle point
                    # (short for R1/initial-point/?)
[[[ P1 ]]]          # Repeat with step 1 from the initial cycle point
                    # (short for R/initial-point/P1)
[[[ P5 ]]]          # Repeat with step 5 from the initial cycle point
                    # (short for R/initial-point/P5)
[[[ R2//P2 ]]]      # Run twice with step 3 from the initial cycle point
                    # (short for R2/initial-point/P2)
[[[ R/+P1/P2 ]]]    # Repeat with step 2, from 1 after the initial cycle point
[[[ R2/P2 ]]]       # Run twice with step 2, to the final cycle point
                    # (short for R2/P2/final-point)
[[[ R1/P0 ]]]       # Run once at the final cycle point
                    # (short for R1/P0/final-point)
```

9.3.4.7.1 Example

The tutorial illustrates integer cycling in 7.23.3, and `$CYLC_DIR/examples/satellite/` is a self-contained example of a realistic use for integer cycling. It simulates the processing of incoming satellite data: each new dataset arrives after a random (as far as the suite is concerned) interval, and is labeled by an arbitrary (as far as the suite is concerned) ID in the filename. A task called `get_data` at the top of the repeating workflow waits on the next dataset and, when it finds one, moves it to a cycle-point-specific shared workspace for processing by the downstream tasks. When `get_data.1` finishes, `get_data.2` triggers and begins waiting for the next dataset at the same time as the downstream tasks in cycle point 1 are processing the first one, and so on. In this way multiple datasets can be processed at once if they happen to come in quickly. A single shutdown task runs at the end of the final cycle to collate results. The suite graph is shown in Figure 25.
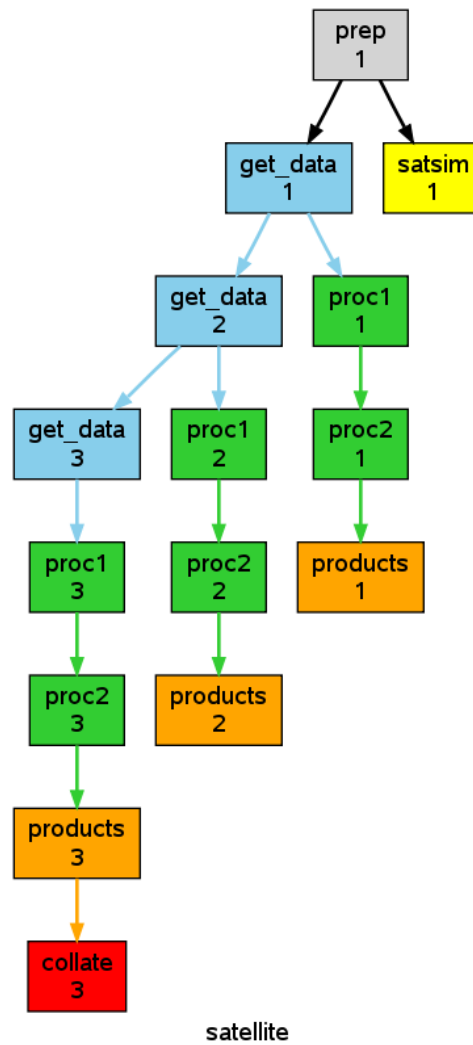
Figure 25: The `examples/satellite` integer suite

9.3.4.7.2   Advanced Integer Cycling Syntax

The same syntax used to reference the initial and final cycle points introduced in 9.3.4.2) for use with datetime cycling can also be used for integer cycling. For example you can write:

```
[[[ R1/^ ]]]       # Run once at the initial cycle point
[[[ R1/$ ]]]       # Run once at the final cycle point
[[[ R3/^/P2 ]]]    # Run three times with step two starting at the
                   # initial cycle point
```

Likewise the syntax introduced in 9.3.4.3 for excluding a particular point from a recurrence also works for integer cycling. For example:

```
[[[ R/P4!8 ]]]          # Run with step 4, to the final cycle point
                        # but not at point 8
[[[ R3/3/P2!5 ]]]       # Run with step 2 from point 3 but not at
                        # point 5
[[[ R/+P1/P6!14 ]]]     # Run with step 6 from 1 step after the
                        # initial cycle point but not at point 14
```

9.3.5   Trigger Types

Trigger type, indicated by *:type* after the upstream task (or family) name, determines what kind of event results in the downstream task (or family) triggering.

9.3.5.1   Success Triggers

The default, with no trigger type specified, is to trigger off the upstream task succeeding:

```
# B triggers if A SUCCEEDS:
    graph = "A => B"
```

For consistency and completeness, however, the success trigger can be explicit:

```
# B triggers if A SUCCEEDS:
    graph = "A => B"
# or:
    graph = "A:succeed => B"
```

9.3.5.2   Failure Triggers

To trigger off the upstream task reporting failure:

```
# B triggers if A FAILS:
    graph = "A:fail => B"
```

Suicide triggers can be used to remove task B here if A does not fail, see 9.3.5.8.

9.3.5.3   Start Triggers

To trigger off the upstream task starting to execute:

```
# B triggers if A STARTS EXECUTING:
    graph = "A:start => B"
```

This can be used to trigger tasks that monitor other tasks once they (the target tasks) start executing. Consider a long-running forecast model, for instance, that generates a sequence of output files as it runs. A postprocessing task could be launched with a start trigger on the model (`model:start => post`) to process the model output as it becomes available. Note, however, that there are several alternative ways of handling this scenario: both tasks could be triggered at the same time (`foo => model & post`), but depending on external queue delays this could result in the monitoring task starting to execute first; or a different postprocessing task could be triggered off a message output for each data file (`model:out1 => post1` etc.; see 9.3.5.5), but this may not be practical if the number of output files is large or if it is difficult to add cylc messaging calls to the model.

9.3.5.4 Finish Triggers

To trigger off the upstream task succeeding or failing, i.e. finishing one way or the other:

```
# B triggers if A either SUCCEEDS or FAILS:
    graph = "A | A:fail => B"
# or
    graph = "A:finish => B"
```

9.3.5.5 Message Triggers

Tasks can also trigger off custom output messages. These must be registered in the `[runtime]` section of the emitting task, and reported using the `cylc message` command in task scripting. The graph trigger notation refers to the item name of the registered output message. The example suite `$CYLC_DIR/examples/message-triggers` illustrates message triggering.

```
title = "test suite for cylc-6 message triggers"

[scheduling]
    initial cycle point = 20140801T00
    final cycle point = 20141201T00
    [[dependencies]]
        [[[P2M]]]
            graph = """foo:out1 => bar
                       foo[-P2M]:out2 => baz"""
[runtime]
    [[foo]]
        script = """
sleep 5
cylc message "file 1 done"
sleep 10
cylc message "file 2 done"
sleep 10"""
        [[[outputs]]]
            out1 = "file 1 done"
            out2 = "file 2 done"
    [[bar, baz]]
        script = sleep 10
```

9.3.5.6 Job Submission Triggers

It is also possible to trigger off a task submitting, or failing to submit:

```
# B triggers if A submits successfully:
    graph = "A:submit => B"
# D triggers if C fails to submit successfully:
    graph = "C:submit-fail => D"
```

```
        graph = """
# D triggers if A or (B and C) succeed
A | B & C => D
# just to align the two graph sections
D => W
# Z triggers if (W or X) and Y succeed
(W|X) & Y => Z
        """
```
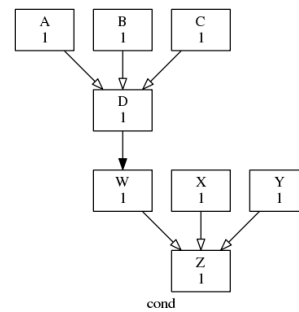

Figure 26: Conditional triggers are plotted with open arrow heads.

A possible use case for submit-fail triggers: if a task goes into the submit-failed state, possibly after several job submission retries, another task that inherits the same runtime but sets a different job submission method and/or host could be triggered to, in effect, run the same job on a different platform.

9.3.5.7 Conditional Triggers

AND operators (`&`) can appear on both sides of an arrow. They provide a concise alternative to defining multiple triggers separately:

```
# 1/ this:
    graph = "A & B => C"
# is equivalent to:
    graph = """A => C
               B => C"""
# 2/ this:
    graph = "A => B & C"
# is equivalent to:
    graph = """A => B
               A => C"""
# 3/ and this:
    graph = "A & B => C & D"
# is equivalent to this:
    graph = """A => C
               B => C
               A => D
               B => D"""
```

OR operators (`|`) which result in true conditional triggers, can only appear on the left,[5]

```
# C triggers when either A or B finishes:
    graph = "A | B => C"
```

Forecasting suites typically have simple conditional triggering requirements, but any valid conditional expression can be used, as shown in Figure 26 (conditional triggers are plotted with open arrow heads).

9.3.5.8 Suicide Triggers

Suicide triggers take tasks out of the suite. This can be used for automated failure recovery. The suite.rc listing and accompanying graph in Figure 27 show how to define a chain of failure

---

[5]An OR operator on the right doesn't make much sense: if "B or C" triggers off A, what exactly should cylc do when A finishes?

```
title = automated failure recovery
description = """
Model task failure triggers diagnosis
and recovery tasks, which take themselves
out of the suite if model succeeds. Model
post processing triggers off model OR
recovery tasks.
              """
[scheduling]
    [[dependencies]]
        graph = """
pre => model
model:fail => diagnose => recover
model => !diagnose & !recover
model | recover => post
                """
[runtime]
    [[model]]
        # UNCOMMENT TO TEST FAILURE:
        # script = /bin/false
```
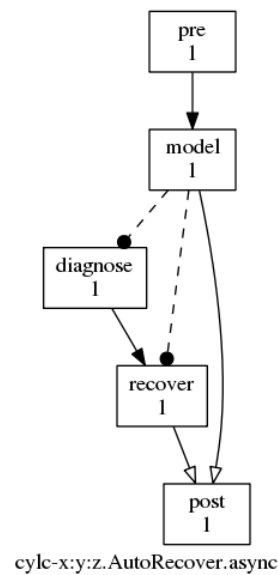
Figure 27: Automated failure recovery via suicide triggers.

recovery tasks that trigger if they're needed but otherwise remove themselves from the suite (you can run the *AutoRecover.async* example suite to see how this works). The dashed graph edges ending in solid dots indicate suicide triggers, and the open arrowheads indicate conditional triggers as usual. Suicide triggers are ignored by default in the graph view, unless you toggle them on with *View -> Options -> Ignore Suicide Triggers*.

Note that multiple suicide triggers combine in the same way as other triggers, so this:

```
foo => !baz
bar => !baz
```

is equivalent to this:

```
foo & bar => !baz
```

i.e. both `foo` and `bar` must succeed for `baz` to be taken out of the suite. If you really want a task to be taken out if any one of several events occurs then be careful to write it that way:

```
foo | bar => !baz
```

A word of warning on the meaning of "bare suicide triggers". Consider the following suite:

```
[scheduling]
    [[dependencies]]
        graph = "foo => !bar"
```

Task `bar` has a suicide trigger but no normal prerequisites (a suicide trigger is not a task triggering prerequisite, it is a task removal prerequisite) so this is entirely equivalent to:

```
[scheduling]
    [[dependencies]]
        graph = """
            foo & bar
          foo => !bar
                """
```

In other words both tasks will trigger immediately, at the same time, and then `bar` will be removed if `foo` succeeds.

If an active task proxy (currently in the submitted or running states) is removed from the suite by a suicide trigger, a warning will be logged.

9.3.5.9   Family Triggers

Families defined by the namespace inheritance hierarchy (9.4) can be used in the graph trigger whole groups of tasks at the same time (e.g. forecast model ensembles and groups of tasks for processing different observation types at the same time) and for triggering downstream tasks off families as a whole. Higher level families, i.e. families of families, can also be used, and are reduced to the lowest level member tasks. Note that tasks can also trigger off individual family members if necessary.

To trigger an entire task family at once:
```
[scheduling]
    [[dependencies]]
        graph = "foo => FAM"
[runtime]
    [[FAM]]     # a family (because others inherit from it)
    [[m1,m2]]   # family members (inherit from namespace FAM)
        inherit = FAM
```

This is equivalent to:
```
[scheduling]
    [[dependencies]]
        graph = "foo => m1 & m2"
[runtime]
    [[FAM]]
    [[m1,m2]]
        inherit = FAM
```

To trigger other tasks off families we have to specify whether to triggering off *all members* starting, succeeding, failing, or finishing, or off *any* members (doing the same). Legal family triggers are thus:
```
[scheduling]
    [[dependencies]]
        graph = """
     # all-member triggers:
    FAM:start-all => one
    FAM:succeed-all => one
    FAM:fail-all => one
    FAM:finish-all => one
      # any-member triggers:
    FAM:start-any => one
    FAM:succeed-any => one
    FAM:fail-any => one
    FAM:finish-any => one
                  """
```

Here's how to trigger downstream processing after if one or more family members succeed, but only after all members have finished (succeeded or failed):
```
[scheduling]
    [[dependencies]]
        graph = """
    FAM:finish-all & FAM:succeed-any => foo
                  """
```

9.3.5.10   Writing Efficient Inter-Family Triggering

While cylc allows writing dependencies between two families it is important to consider the number of dependencies this will generate. In the following example, each member of `FAM2` has dependencies pointing at all the members of `FAM1`.

```
[scheduling]
    [[dependencies]]
        graph = """
    FAM1:succeed-any => FAM2
                """
```

Expanding this out, you generate `N * M` dependencies, where `N` is the number of members of `FAM1` and `M` is the number of members of `FAM2`. This can result in high memory use as the number of members of these families grows, potentially rendering the suite impractical for running on some systems.

You can greatly reduce the number of dependencies generated in these situations by putting dummy tasks in the graphing to represent the state of the family you want to trigger off. For example, if `FAM2` should trigger off any member of `FAM1` succeeding you can create a dummy task `FAM1_succeed_any_marker` and place a dependency on it as follows:

```
[scheduling]
    [[dependencies]]
        graph = """
    FAM1:succeed-any => FAM1_succeed_any_marker => FAM2
                """
[runtime]
...
    [[FAM1_succeed_any_marker]]
        script = true
...
```

This graph generates only `N + M` dependencies, which takes significantly less memory and CPU to store and evaluate.

9.3.5.11   Inter-Cycle Triggers

Typically most tasks in a suite will trigger off others in the same cycle point, but some may depend on others with other cycle points. This notably applies to warm-cycled forecast models, which depend on their own previous instances (see below); but other kinds of inter-cycle dependence are possible too.[6] Here's how to express this kind of relationship in cylc:

```
[dependencies]
    [[PT6H]]
        # B triggers off A in the previous cycle point
        graph = "A[-PT6H] => B"
```

inter-cycle and trigger type (or message trigger) notation can be combined:

```
        # B triggers if A in the previous cycle point fails:
        graph = "A[-PT6H]:fail => B"
```

At suite start-up inter-cycle triggers refer to a previous cycle point that does not exist. This does not cause the dependent task to wait indefinitely, however, because cylc ignores triggers that reach back beyond the initial cycle point. That said, the presence of an inter-cycle trigger

---

[6]In NWP forecast analysis suites parts of the observation processing and data assimilation subsystem will typically also depend on model background fields generated by the previous forecast.

does normally imply that something special has to happen at start-up. If a model depends on its own previous instance for restart files, for instance, then an initial set of restart files has to be generated somehow or the first model task will presumably fail with missing input files. There are several ways to handle this in cylc using different kinds of one-off (non-cycling) tasks that run at suite start-up. They are illustrated in the Tutorial (7.23.1); to summarize here briefly:

- `R1` tasks (recommended):

```
[scheduling]
    [[dependencies]]
        [[[R1]]]
            graph = "prep"
        [[[R1/T00,R1/T12]]]
            graph = "prep[^] => foo"
        [[[T00,T12]]]
            graph = "foo[-PT12H] => foo => bar"
```

`R1`, or `R1/date-time` tasks are the recommended way to specify unusual start up conditions. They allow you to specify a clean distinction between the dependencies of initial cycles and the dependencies of the subsequent cycles.

Initial tasks can be used for real model cold-start processes, whereby a warm-cycled model at any given cycle point can in principle have its inputs satisfied by a previous instance of itself, *or* by an initial task with (nominally) the same cycle point.

In effect, the `R1` task masquerades as the previous-cycle-point trigger of its associated cycling task. At suite start-up initial tasks will trigger the first cycling tasks, and thereafter the inter-cycle trigger will take effect.

If a task has a dependency on another task in a different cycle point, the dependency can be written using the `[offset]` syntax such as `[-PT12H]` in `foo[-PT12H] => foo`. This means that `foo` at the current cycle point depends on a previous instance of `foo` at 12 hours before the current cycle point. Unlike the cycling section headings (e.g. `[[[T00,T12]]]`), dependencies assume that relative times are relative to the current cycle point, not the initial cycle point.

However, it can be useful to have specific dependencies on tasks at or near the initial cycle point. You can switch the context of the offset to be the initial cycle point by using the caret symbol: `^`.

For example, you can write `foo[^]` to mean foo at the initial cycle point, and `foo[^+PT6H]` to mean foo 6 hours after the initial cycle point. Usually, this kind of dependency will only apply in a limited number of cycle points near the start of the suite, so you may want to write it in `R1`-based cycling sections. Here's the example inter-cycle `R1` suite from above again.

```
[scheduling]
    [[dependencies]]
        [[[R1]]]
            graph = "prep"
        [[[R1/T00,R1/T12]]]
            graph = "prep[^] => foo"
        [[[T00,T12]]]
            graph = "foo[-PT12H] => foo => bar"
```

You can see there is a dependence on the initial `R1` task `prep` for `foo` at the first `T00` cycle point, and at the first `T12` cycle point. Thereafter, `foo` just depends on its previous (12 hours ago) instance.

Finally, it is also possible to have a dependency on a task at a specific cycle point.

```
[scheduling]
    [[dependencies]]
        [[[R1/20200202]]]
            graph = "baz[20200101] => qux"
```

However, in a long running suite, a repeating cycle should avoid having a dependency on a task with a specific cycle point (including the initial cycle point) - as it can currently cause performance issue. In the following example, all instances of `qux` will depend on `baz.20200101`, which will never be removed from the task pool.:

```
[scheduling]
    initial cycle point = 2010
    [[dependencies]]
        # Can cause performance issue!
        [[[P1D]]]
            graph = "baz[20200101] => qux"
```

9.3.5.12   Special Sequential Tasks

If a cycling task does not generate files required by its own successor, then successive instances can run in parallel if the opportunity arises. However, if such a task would interfere with its own siblings for internal reasons (e.g. use of a hardwired non cycle dependent temporary file or similar) then it can be forced to run sequentially. This can be done with explicit inter-cycle triggers in the graph:

```
[scheduling]
    [[dependencies]]
        [[[T00,T12]]]
            graph = "foo[-PT12H] => foo => bar"
```

or by declaring the task to be *sequential*:

```
[scheduling]
    [[special tasks]]
        sequential = foo
    [[dependencies]]
        [[[T00,T12]]]
            graph = "foo => bar"
```

The *sequential* declaration also results in each instance of `foo` triggering off its own predecessor, exactly as in the explicit version. The only difference is that implicit triggers will not appear in graph visualizations. The implicit version can also be considerably simpler when the task appears in multiple graph sections or in a non-uniform cycling sequence: this suite:

```
[scheduling]
    [[special tasks]]
        sequential = foo
    [[dependencies]]
        [[[T00,T03,T11]]]
            graph = "foo => bar"
```

is equivalent to this one:

```
[scheduling]
    [[dependencies]]
        [[[T00,T03,T11]]]
            graph = "foo => bar"
        [[[T00]]]
            graph = "foo[-PT13H] => foo"
        [[[T03]]]
            graph = "foo[-PT3H] => foo"
        [[[T11]]]
            graph = "foo[-PT8H] => foo"
```

9.3.5.13   Future Triggers

Cylc also supports inter-cycle triggering off tasks "in the future" (with respect to cycle point):

```
[[dependencies]]
    [[[T00,T06,T12,T18]]]
        graph = """
    # A runs in this cycle:
            A
    # B in this cycle triggers off A in the next cycle.
          A[PT6H] => B
        """
```

(Recall that `A[t+PT6H]` can run before B[t] because tasks in cylc have private cycle points). Future triggers present a problem at the suite shutdown rather than at start-up. Here, `B` at the final cycle point wants to trigger off an instance of `A` that will never exist because it is beyond the suite stop point. Consequently cylc prevents tasks from spawning successors that depend on other tasks beyond the stop point.

9.3.5.14   Clock Triggers

In addition to depending on other tasks (and on external events - see 9.3.5.16) tasks can depend on the wall clock: specifically, they can trigger off a wall clock time expressed as an offset from their own cycle point:

```
[scheduling]
    [[special tasks]]
        clock-trigger = foo(PT2H)
    [[dependencies]]
        [[[T00]]]
            graph = foo
```

Here, `foo[2015-08-23T00]` would trigger (other dependencies allowing) when the wall clock time reaches `2015-08-23T02`. Clock-trigger offsets are normally positive, to trigger some time *after* the wall-clock time is equal to task cycle point.

Clock-triggers have no effect on scheduling if the suite is running sufficiently far behind the clock (e.g. after a delay, or because it is processing archived historical data) that the trigger times, which are relative to task cycle point, have already passed.

9.3.5.15   Clock-Expire Triggers

Tasks can be configured to *expire* - i.e. to skip job submission and enter the *expired* state - if they are too far behind the wall clock when they become ready to run, and other tasks can trigger off this. As a possible use case, consider a cycling task that copies the latest of a set of files to overwrite the previous set: if the task is delayed by more than one cycle there may be no point in running it because the freshly copied files will just be overwritten immediately by the next task instance as the suite catches back up to real time operation. Clock-expire tasks are configured like clock-trigger tasks, with a date-time offset relative to cycle point (A.3.11.2). The offset should be positive to make the task expire if the wall-clock time has gone beyond the cycle point. Triggering off an expired task typically requires suicide triggers to remove the workflow that runs if the task has not expired. Here a task called `copy` expires, and its downstream workflow is skipped, if it is more than one day behind the wall-clock (see also `examples/clock-expire`):

```
[cylc]
   cycle point format = %Y-%m-%dT%H
[scheduling]
    initial cycle point = 2015-08-15T00
    [[special tasks]]
        clock-expire = copy(-P1D)
    [[dependencies]]
        [[[P1D]]]
            graph = """
        model[-P1D] => model => copy => proc
                copy:expired => !proc"""
```

9.3.5.16   External Triggers

In addition to depending on other tasks (and on the wall clock - see 9.3.5.14) tasks can trigger off events reported by an external system. For example, an external process could detect incoming data on an ftp server, and then notify a suite containing a task to retrieve the new data for processing. This is an alternative to long-running tasks that poll for external events.

Note that cylc does not currently support triggering off "filesystem events" (e.g. `inotify` on Linux). However, external watcher processes can use filesystem events to detect triggering conditions, if that is appropriate, before notifying a suite with our general external event system.

The external triggering process must call `cylc ext-trigger` with the name of the target suite, the message that identifies this type of event to the suite, and an ID that distinguishes this particular event instance from others (the name of the target task or its current cycle point is not required). The event ID is just an arbitary string to cylc, but it typically identifies the filename(s) of the latest dataset in some way. When the suite daemon receives the external event notification it will trigger the next instance of any task waiting on that trigger (whatever its cycle point) and then broadcast (see 12.19) the event ID to the cycle point of the triggered task as `$CYLC_EXT_TRIGGER_ID`. Downstream tasks with the same cycle point therefore know the new event ID too and can use it, if they need to, to identify the same new dataset. In this way a whole workflow can be associated with each new dataset, and multiple datasets can be processed in parallel if they happen to arrive in quick succession.

An externally-triggered task must register the event it waits on in the suite scheduling section:

```
# suite "sat-proc"
[scheduling]
    cycling mode = integer
    initial cycle point = 1
    [[special tasks]]
        external-trigger = get-data("new sat X data avail")
    [[dependencies]]
        [[[P1]]]
            graph = get-data => conv-data => products
```

Then, each time a new dataset arrives the external detection system should notify the suite like this:

```
$ cylc ext-trigger sat-proc "new sat X data avail" passX12334a
```

where "sat-proc" is the suite name and "passX12334a" is the ID string for the new event. The suite passphrase must be installed on triggering account.

Note that only one task in a suite can trigger off a particular external message. Other tasks can trigger off the externally triggered task as required, of course.

`$CYLC_DIR/examples/satellite/ext-triggers/suite.rc` is a working example of a simulated satellite processing suite.

External triggers are not normally needed in date-time cycling suites driven by real time data that comes in at regular intervals. In these cases a data retrieval task can be clock-triggered (and have appropriate retry intervals supplied) to submit at the expected data arrival time, so little time if any is wasted in polling. However, if the arrival time of the cycle-point-specific data is highly variable, external triggering may be used with the cycle point embedded in the message:

```
# suite "data-proc"
[scheduling]
    initial cycle point = 20150125T00
    final cycle point   = 20150126T00
    [[special tasks]]
        external-trigger = get-data("data arrived for $CYLC_TASK_CYCLE_POINT")
    [[dependencies]]
        [[[T00]]]
            graph = init-process => get-data => post-process
```

Once the variable-length waiting is finished, an external detection system should notify the suite like this:

```
$ cylc ext-trigger data-proc "data arrived for 20150126T00" passX12334a
```

where "data-proc" is the suite name, the cycle point has replaced the variable in the trigger string, and "passX12334a" is the ID string for the new event. The suite passphrase must be installed on the triggering account. In this case, the event will trigger for the second cycle point but not the first because of the cycle-point matching.

9.3.6   Model Restart Dependencies

Warm-cycled forecast models generate *restart files*, e.g. model background fields, to initialize the next forecast. This kind of dependence requires an inter-cycle trigger:

```
[scheduling]
    [[dependencies]]
        [[[T00,T06,T12,T18]]]
            graph = "A[-PT6H] => A"
```

If your model is configured to write out additional restart files to allow one or more cycle points to be skipped in an emergency *do not represent these potential dependencies in the suite graph* as they should not be used under normal circumstances. For example, the following graph would result in task `A` erroneously triggering off `A[T-24]` as a matter of course, instead of off `A[T-6]`, because `A[T-24]` will always be finished first:

```
[scheduling]
    [[dependencies]]
        [[[T00,T06,T12,T18]]]
            # DO NOT DO THIS (SEE ACCOMPANYING TEXT):
            graph = "A[-PT24H] | A[-PT18H] | A[-PT12H] | A[-PT6H] => A"
```

9.3.7   How The Graph Determines Task Instantiation

A graph trigger pair like `foo => bar` determines the existence and prerequisites (dependencies) of the downstream task `bar`, for the cycle points defined by the associated graph section heading. In general it does not say anything about the dependencies or existence of the upstream task

`foo`. However *if the trigger has no cycle point offset* Cylc will infer that `bar` must exist at the same cycle points as `foo`. This is a convenience to allow this:

```
graph = "foo => bar"
```

to be written as shorthand for this:

```
graph = """foo
           foo => bar"""
```

(where `foo` by itself means `<nothing> => foo`, i.e. the task exists at these cycle points but has no prerequisites - although other prerequisites may be defined for it in other parts of the graph).

Cylc does not infer the existence of the upstream task in offset triggers like `foo[-P1D]=> bar` because, as explained in Section K.5, a typo in the offset interval should generate an error rather than silently creating tasks on an erroneous cycling sequence.

As a result you need to be careful not to define inter-cycle dependencies that cannot be satisfied at run time. Suite validation catches this kind of error if the existence of the cycle offset task is not defined anywhere at all:

```
[scheduling]
    initial cycle point = 2020
    [[dependencies]]
        [[[P1Y]]]
            # ERROR
            graph = "foo[-P1Y] => bar"

$ cylc validate SUITE
'ERROR: No cycling sequences defined for foo'
```

To fix this, use another line in the graph to tell Cylc to define `foo` at each cycle point:

```
[scheduling]
    initial cycle point = 2020
    [[dependencies]]
        [[[P1Y]]]
            graph = """
                foo
                foo[-P1Y] => bar"""
```

But validation does not catch this kind of error if the offset task is defined only on a different cycling sequence:

```
[scheduling]
    initial cycle point = 2020
    [[dependencies]]
        [[[P2Y]]]
            graph = """foo
                # ERROR
                foo[-P1Y] => bar"""
```

This suite will validate OK, but it will stall at runtime with `bar` waiting on `foo[-P1Y]` at the intermediate years where it does not exist. The offset `[-P1Y]` is presumably an error (it should be `[-P2Y]`), or else another graph line is needed to generate `foo` instances on the yearly sequence:

```
[scheduling]
    initial cycle point = 2020
    [[dependencies]]
        [[[P1Y]]]
            graph = "foo"
        [[[P2Y]]]
            graph = "foo[-P1Y] => bar"
```

Similarly the following suite will validate OK, but it will stall at runtime with `bar` waiting on `foo[-P1Y]` in every cycle point, when only a single instance of it exists, at the initial cycle point:

```
[scheduling]
    initial cycle point = 2020
    [[dependencies]]
        [[[R1]]]
            graph = foo
        [[[P1Y]]]
            # ERROR
            graph = foo[-P1Y] => bar
```

9.4   Runtime - Task Configuration

The `[runtime]` section of a suite definition configures what to execute (and where and how to execute it) when each task is ready to run, in a *multiple inheritance hierarchy* of *namespaces* culminating in individual tasks. This allows all common configuration detail to be factored out and defined in one place.

Any namespace can configure any or all of the items defined in the *Suite.rc Reference* (A).

Namespaces that do not explicitly inherit from others automatically inherit from the *root* namespace (below).

Nested namespaces define *task families* that can be used in the graph as convenient shorthand for triggering all member tasks at once, or for triggering other tasks off all members at once - see 9.3.5.9. Nested namespaces can be progressively expanded and collapsed in the dependency graph viewer, and in the gcylc graph and text views. Only the first parent of each namespace (as for single-inheritance) is used for suite visualization purposes.

9.4.1   Namespace Names

Namespace names may contain letters, digits, underscores, and hyphens.

Note that *task names need not be hardwired into task implementations* because task and suite identity can be extracted portably from the task execution environment supplied by the suite daemon (9.4.7) - then to rename a task you can just change its name in the suite definition.

9.4.2   Root - Runtime Defaults

The root namespace, at the base of the inheritance hierarchy, provides default configuration for all tasks in the suite. Most root items are unset by default, but some have default values sufficient to allow test suites to be defined by dependency graph alone. The *script* item, for example, defaults to code that prints a message then sleeps for between 1 and 15 seconds and exits. Default values are documented with each item in A. You can override the defaults or provide your own defaults by explicitly configuring the root namespace.

9.4.3   Defining Multiple Namespaces At Once

If a namespace section heading is a comma-separated list of names then the subsequent configuration applies to each list member. Particular tasks can be singled out at run time using the `$CYLC_TASK_NAME` variable.

As an example, consider a suite containing an ensemble of closely related tasks that each invokes the same script but with a unique argument that identifies the calling task name:

```
[runtime]
    [[ENSEMBLE]]
        script = "run-model.sh $CYLC_TASK_NAME"
    [[m1, m2, m3]]
        inherit = ENSEMBLE
```

For large ensembles Jinja2 template processing can be used to automatically generate the member names and associated dependencies (see 9.7).

9.4.4   Runtime Inheritance - Single

The following listing of the *inherit.single.one* example suite illustrates basic runtime inheritance with single parents.

```
# SUITE.RC
title = "User Guide [runtime] example."
[cylc]
    required run mode = simulation # (no task implementations)
[scheduling]
    initial cycle point = 20110101T06
    final cycle point = 20110102T00
    [[dependencies]]
        [[[T00]]]
            graph = """foo => OBS
                OBS:succeed-all => bar"""
[runtime]
    [[root]] # base namespace for all tasks (defines suite-wide defaults)
        [[[job]]]
            batch system = at
        [[[environment]]]
            COLOR = red
    [[OBS]]   # family (inherited by land, ship); implicitly inherits root
        script = run-${CYLC_TASK_NAME}.sh
        [[[environment]]]
            RUNNING_DIR = $HOME/running/$CYLC_TASK_NAME
    [[land]] # a task (a leaf on the inheritance tree) in the OBS family
        inherit = OBS
        description = land obs processing
    [[ship]] # a task (a leaf on the inheritance tree) in the OBS family
        inherit = OBS
        description = ship obs processing
        [[[job]]]
            batch system = loadleveler
        [[[environment]]]
            RUNNING_DIR = $HOME/running/ship  # override OBS environment
            OUTPUT_DIR = $HOME/output/ship    # add to OBS environment
    [[foo]]
        # (just inherits from root)

    # The task [[bar]] is implicitly defined by its presence in the
    # graph; it is also a dummy task that just inherits from root.
```

9.4.5   Runtime Inheritance - Multiple

If a namespace inherits from multiple parents the linear order of precedence (which namespace overrides which) is determined by the so-called *C3 algorithm* used to find the linear *method resolution order* for class hierarchies in Python and several other object oriented programming languages. The result of this should be fairly obvious for typical use of multiple inheritance in cylc suites, but for detailed documentation of how the algorithm works refer to the official Python documentation here: http://www.python.org/download/releases/2.3/mro/.

The *inherit.multi.one* example suite, listed here, makes use of multiple inheritance:

```
title = "multiple inheritance example"

description = """To see how multiple inheritance works:

 % cylc list -tb[m] SUITE # list namespaces
 % cylc graph -n SUITE # graph namespaces
 % cylc graph SUITE # dependencies, collapse on first-parent namespaces

  % cylc get-config --sparse --item [runtime]ops_s1 SUITE
  % cylc get-config --sparse --item [runtime]var_p2 foo"""
[scheduling]
    [[dependencies]]
        graph = "OPS:finish-all => VAR"

[runtime]
    [[root]]
    [[OPS]]
        script = echo "RUN: run-ops.sh"
    [[VAR]]
        script = echo "RUN: run-var.sh"
    [[SERIAL]]
        [[[directives]]]
            job_type = serial
    [[PARALLEL]]
        [[[directives]]]
            job_type = parallel
    [[ops_s1, ops_s2]]
        inherit = OPS, SERIAL

    [[ops_p1, ops_p2]]
        inherit = OPS, PARALLEL

    [[var_s1, var_s2]]
        inherit = VAR, SERIAL

    [[var_p1, var_p2]]
        inherit = VAR, PARALLEL

[visualization]
    # NOTE ON VISUALIZATION AND MULTIPLE INHERITANCE: overlapping
    # family groups can have overlapping attributes, so long as
    # non-conflictling attributes are used to style each group. Below,
    # for example, OPS tasks are filled green and SERIAL tasks are
    # outlined blue, so that ops_s1 and ops_s2 are green with a blue
    # outline. But if the SERIAL tasks are explicitly styled as "not
    # filled" (by setting "style=") this will override the fill setting
    # in the (previously defined and therefore lower precedence) OPS
    # group, making ops_s1 and ops_s2 unfilled with a blue outline.
    # Alternatively you can just create a manual node group for ops_s1
    # and ops_s2 and style them separately.
    [[node groups]]
        #(see comment above:)
        #serial_ops = ops_s1, ops_s2
    [[node attributes]]
        OPS = "style=filled", "fillcolor=green"
        SERIAL = "color=blue" #(see comment above:), "style="
        #(see comment above:)
        #serial_ops = "color=blue", "style=filled", "fillcolor=green"
```

`cylc get-suite-config` provides an easy way to check the result of inheritance in a suite. You can extract specific items, e.g.:

```
$ cylc get-suite-config --item '[runtime][var_p2]script' \
    inherit.multi.one
echo ''RUN: run-var.sh''
```

or use the `--sparse` option to print entire namespaces without obscuring the result with the

dense runtime structure obtained from the root namespace:

```
$ cylc get-suite-config --sparse --item '[runtime]ops_s1' inherit.multi.one
script = echo ''RUN: run-ops.sh''
inherit = ['OPS', 'SERIAL']
[directives]
   job_type = serial
```

9.4.5.1   Suite Visualization And Multiple Inheritance

The first parent inherited by a namespace is also used as the collapsible family group when visualizing the suite. If this is not what you want, you can demote the first parent for visualization purposes, without affecting the order of inheritance of runtime properties:

```
[runtime]
    [[BAR]]
        # ...
    [[foo]]
        # inherit properties from BAR, but stay under root for visualization:
        inherit = None, BAR
```

9.4.6   How Runtime Inheritance Works

The linear precedence order of ancestors is computed for each namespace using the C3 algorithm. Then any runtime items that are explicitly configured in the suite definition are "inherited" up the linearized hierarchy for each task, starting at the root namespace: if a particular item is defined at multiple levels in the hierarchy, the level nearest the final task namespace takes precedence. Finally, root namespace defaults are applied for every item that has not been configured in the inheritance process (this is more efficient than carrying the full dense namespace structure through from root from the beginning).

9.4.7   Task Execution Environment

The task execution environment contains suite and task identity variables provided by the suite daemon, and user-defined environment variables. The environment is explicitly exported (by the task job script) prior to executing the task `script` (see 11).

Suite and task identity are exported first, so that user-defined variables can refer to them. Order of definition is preserved throughout so that variable assignment expressions can safely refer to previously defined variables.

Additionally, access to cylc itself is configured prior to the user-defined environment, so that variable assignment expressions can make use of cylc utility commands:

```
[runtime]
    [[foo]]
        [[[environment]]]
            REFERENCE_TIME = $( cylc util cycletime --offset-hours=6 )
```

9.4.7.1   User Environment Variables

A task's user-defined environment results from its inherited `[[[environment]]]` sections:

```
[runtime]
    [[root]]
        [[[environment]]]
            COLOR = red
            SHAPE = circle
    [[foo]]
        [[[environment]]]
            COLOR = blue  # root override
            TEXTURE = rough # new variable
```

This results in a task *foo* with `SHAPE=circle`, `COLOR=blue`, and `TEXTURE=rough` in its environment.

9.4.7.2 Overriding Environment Variables

When you override inherited namespace items the original parent item definition is *replaced* by the new definition. This applies to all items including those in the environment sub-sections which, strictly speaking, are not "environment variables" until they are written, post inheritance processing, to the task job script that executes the associated task. Consequently, if you override an environment variable you cannot also access the original parent value:

```
[runtime]
    [[FOO]]
        [[[environment]]]
            COLOR = red
    [[bar]]
        inherit = FOO
        [[[environment]]]
            tmp = $COLOR        # !! ERROR: $COLOR is undefined here
            COLOR = dark-$tmp   # !! as this overrides COLOR in FOO.
```

The compressed variant of this, `COLOR = dark-$COLOR`, is also in error for the same reason. To achieve the desired result you must use a different name for the parent variable:

```
[runtime]
    [[FOO]]
        [[[environment]]]
            FOO_COLOR = red
    [[bar]]
        inherit = FOO
        [[[environment]]]
            COLOR = dark-$FOO_COLOR  # OK
```

9.4.7.3 Task Job Script Variables

These are variables that can be referenced (but should not be modified) in a task job script.

The task job script may export the following environment variables:

```
CYLC_DEBUG                       # Debug mode, true or not defined
CYLC_DIR                         # Location of cylc installation used
CYLC_VERSION                     # Version of cylc installation used

CYLC_CYCLING_MODE                # Cycling mode, e.g. gregorian
CYLC_SUITE_FINAL_CYCLE_POINT     # Final cycle point
CYLC_SUITE_INITIAL_CYCLE_POINT   # Initial cycle point
CYLC_SUITE_NAME                  # Suite name
CYLC_UTC                         # UTC mode, True or False
CYLC_VERBOSE                     # Verbose mode, True or False
TZ                               # Set to "UTC" in UTC mode or not defined

CYLC_SUITE_RUN_DIR               # Location of the suite run directory in
                                 # job host, e.g. ~/cylc-run/foo
CYLC_SUITE_DEF_PATH              # Location of the suite definition directory in
```

```
                                   # job host, e.g. ~/cylc-run/foo
CYLC_SUITE_HOST                    # Host running the suite process
CYLC_SUITE_OWNER                   # User ID running the suite process
CYLC_SUITE_DEF_PATH_ON_SUITE_HOST
                                   # Location of the suite definition directory in
                                   # suite host, e.g. ~/cylc-run/foo
CYLC_SUITE_SHARE_DIR               # Suite (or task!) shared directory (see below)
CYLC_SUITE_WORK_DIR                # Suite work directory (see below)

CYLC_TASK_JOB                      # Task job identifier expressed as
                                   # CYCLE-POINT/TASK-NAME/SUBMIT-NUM
                                   # e.g. 20110511T1800Z/t1/01
CYLC_TASK_CYCLE_POINT              # Cycle point, e.g. 20110511T1800Z
CYLC_TASK_NAME                     # Job's task name, e.g. t1
CYLC_TASK_SUBMIT_NUMBER            # Job's submit number, e.g. 1,
                                   # increments with every submit
CYLC_TASK_TRY_NUMBER               # Number of execution tries, e.g. 1
                                   # increments with automatic retry-on-fail
CYLC_TASK_ID                       # Task instance identifier expressed as
                                   # TASK-NAME.CYCLE-POINT
                                   # e.g. t1.20110511T1800Z
CYLC_TASK_LOG_ROOT                 # Location of the job file
                                   # e.g. ~/cylc-run/foo/log/job/20110511T1800Z/t1/01/job
CYLC_TASK_WORK_DIR                 # Location of task work directory (see below)
                                   # e.g. ~/cylc-run/foo/work/20110511T1800Z/t1
CYLC_TASK_NAMESPACE_HIERARCHY      # Linearised family namespace of the task,
                                   # e.g. root postproc t1

CYLC_TASK_COMMS_METHOD             # Set to "ssh" if communication method is "ssh"
CYLC_TASK_SSH_LOGIN_SHELL          # With "ssh" communication, if set to "True",
                                   # use login shell on suite host
```

There are also some global shell variables that may be defined in the task job script (but not exported to the environment). These include:

```
CYLC_FAIL_SIGNALS                  # List of signals trapped by the error trap
CYLC_VACATION_SIGNALS              # List of signals trapped by the vacation trap
CYLC_SUITE_WORK_DIR_ROOT           # Root directory above the suite work directory
                                   # in the job host
CYLC_TASK_MESSAGE_STARTED_PID      # PID of "cylc task message started" command
CYLC_TASK_WORK_DIR_BASE            # Alternate task work directory,
                                   # relative to the suite work directory
```

9.4.7.4   Suite Share Directories

A *suite share directory* is created automatically under the suite run directory as a share space for tasks. The location is available to tasks as `$CYLC_SUITE_SHARE_DIR`. In a cycling suite, output files are typically held in cycle point sub-directories of the suite share directory.

The top level share and work directory (below) location can be changed (e.g. to a large data area) by a global config setting (see B.9.1.2).

9.4.7.5   Task Work Directories

Task job scripts are executed from within *work directories* created automatically under the suite run directory. A task can get its own work directory from `$CYLC_TASK_WORK_DIR` (or simply `$PWD` if it does not `cd` elsewhere at runtime). By default the location contains task name and cycle point, to provide a unique workspace for every instance of every task. This can be overridden in the suite definition, however, to get several tasks to share the same work directory (see A.4.1.11).

The top level work and share directory (above) location can be changed (e.g. to a large data area) by a global config setting (see B.9.1.2).

9.4.7.6   Environment Variable Evaluation

Variables in the task execution environment are not evaluated in the shell in which the suite is running prior to submitting the task. They are written in unevaluated form to the job script that is submitted by cylc to run the task (10.1) and are therefore evaluated when the task begins executing under the task owner account on the task host. Thus `$HOME`, for instance, evaluates at run time to the home directory of task owner on the task host.

9.4.8   How Tasks Get Access To The Suite Directory

Tasks can use `$CYLC_SUITE_DEF_PATH` to access suite files on the task host, and the suite bin directory is automatically added `$PATH`. If a remote suite definition directory is not specified the local (suite host) path will be assumed with the local home directory, if present, swapped for literal `$HOME` for evaluation on the task host.

9.4.9   Remote Task Hosting

If a task declares an owner other than the suite owner and/or a host other than the suite host, cylc will use non-interactive ssh to execute the task on the `owner@host` account by the configured batch system:

```
[runtime]
    [[foo]]
        [[[remote]]]
            host = orca.niwa.co.nz
            owner = bob
        [[[job]]]
            batch system = pbs
```

For this to work:

- non-interactive ssh/scp is required from the suite host to the task host accounts. Note: Your shell initialization (.profile, .bashrc, .cshrc, etc) on the remote host must not produce any standard output as it may confuse commands such as scp. See http://www.openssh.com/faq.html#2.9 for more information.
- cylc must be installed on task hosts.
    - Optional software dependencies such as graphviz and Jinja2 are not needed on task hosts.
    - If polling task communication is used, there is no other requirement.
    - If SSH task communication is configured, non-interactive ssh is required from the task host to the suite host.
    - If (default) task communication is configured, the task host should have access to the port on the suite host.
- the suite definition directory, or some fraction of its content, can be installed on the task host, if needed.

To learn how to give remote tasks access to cylc, see 12.7.

Tasks running on the suite host under another user account are treated as remote tasks.

Remote hosting, like all namespace settings, can be declared globally in the root namespace, or per family, or for individual tasks.

9.4.9.1   Dynamic Host Selection

Instead of hardwiring host names into the suite definition you can specify a shell command that prints a hostname, or an environment variable that holds a hostname, as the value of the host config item. See A.4.1.14.1.

9.4.9.2   Remote Task Log Directories

Task stdout and stderr streams are written to log files in a suite-specific sub-directory of the *suite run directory*, as explained in 11.2. For remote tasks the same directory is used, but *on the task host.* Remote task log directories, like local ones, are created on the fly, if necessary, during job submission.

9.5   Visualization

The visualization section of a suite definition is used to configure suite graphing, principally graph node (task) and edge (dependency arrow) style attributes. Tasks can be grouped for the purpose of applying common style attributes. See A for details.

9.5.1   Collapsible Families In Suite Graphs

```
[visualization]
    collapsed families = family1, family2
```

Nested families from the runtime inheritance hierarchy can be expanded and collapsed in suite graphs and the gcylc graph view. All families are displayed in the collapsed state at first, unless `[visualization]collapsed families` is used to single out specific families for initial collapsing.

In the gcylc graph view, nodes outside of the main graph (such as the members of collapsed families) are plotted as rectangular nodes to the right if they are doing anything interesting (submitted, running, failed).

Figure 28 illustrates successive expansion of nested task families in the *namespaces* example suite.

9.6   Parameterized Tasks

Cylc can automatically generate tasks and dependencies by expanding parameterized task names over lists of parameter values. Uses for this include:

- generating an ensemble of similar model runs
- generating chains of tasks to process similar datasets
- replicating an entire workflow, or part thereof, over several runs
- splitting a long model run into smaller steps or "chunks" (parameterized cycling)

Note that this can be done with Jinja2 loops too (Section 9.7) but parameterization is much cleaner (nested loops can seriously reduce the clarity of a suite definition).
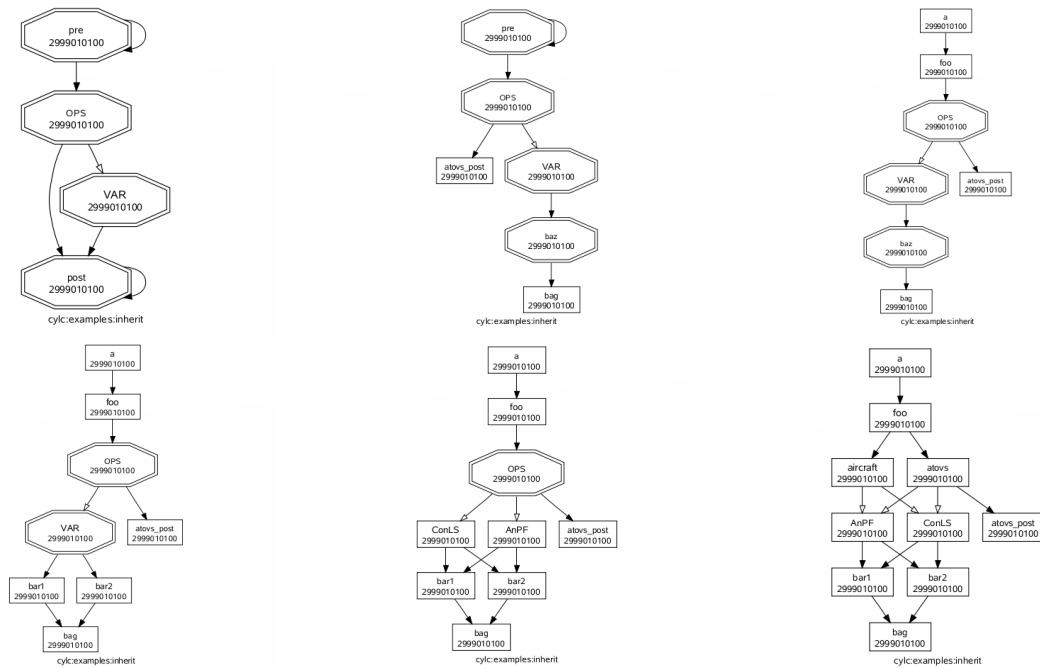
Figure 28: Graphs of the *namespaces* example suite showing various states of expansion of the nested namespace family hierarchy, from all families collapsed (top left) through to all expanded (bottom right). This can also be done by right-clicking on tasks in the gcylc graph view.

9.6.1 Parameter Expansion

Parameter values can be lists of strings, or integer ranges (with inclusive bounds):

```
[cylc]
    [[parameters]]
        obs = ship, buoy, plane
        run = 1..5  # 1, 2, 3, 4, 5
```

Then angle brackets denote use of these parameters throughout the suite definition. For the values above, this parameterized name:

```
model<run>  # for run = 1..2
```

expands to these concrete task names:

```
model_run1, model_run2
```

and this parameterized name:

```
proc<obs>  # for obs = ship, buoy, plane
```

expands to these concrete task names:

```
proc_ship, proc_buoy, proc_plane
```

By default, to avoid any ambiguity, the parameter name appears in the expanded task names for integer values, but not for string values. For example, `model_run1` for `run = 1`, but `proc_ship` for `obs = ship`. However, the default expansion templates can be overridden if need be:

```
[cylc]
    [[parameters]]
        obs = ship, buoy, plane
        run = 1..5
```
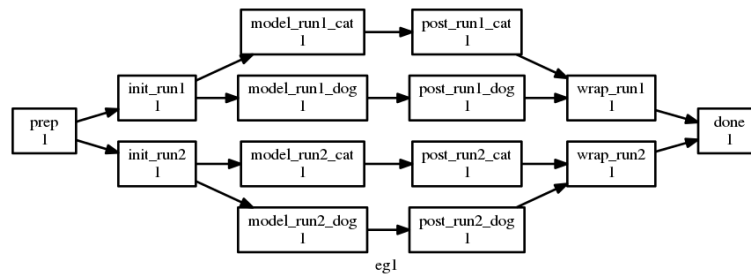
Figure 29: Parameter expansion example.

```
[[parameter templates]]
    run = -R%(run)s  # Make foo<run> expand to foo-R1 etc.
```

(See A.2.12 for more on the string template syntax.)

Any number of parameters can be used at once. This parameterization:

```
model<run,obs>  # for run = 1..2 and obs = ship, buoy, plane
```

expands to these tasks names:

```
model_run1_ship, model_run1_buoy, model_run1_plane,
model_run2_ship, model_run2_buoy, model_run2_plane
```

Here's a simple but complete example suite:

```
[cylc]
    [[parameters]]
        run = 1..2
[scheduling]
    [[dependencies]]
        graph = "prep => model<run>"
[runtime]
    [[model<run>]]
        # ...
```

The result, post parameter expansion, is this:

```
[scheduling]
    [[dependencies]]
        graph = "prep => model_run1 & model_run2"
[runtime]
    [[model_run1]]
        # ...
    [[model_run2]]
        # ...
```

Here's a more complex graph using two parameters (`[runtime]` omitted):

```
[cylc]
    [[parameters]]
        run = 1..2
        mem = cat, dog
[scheduling]
    [[dependencies]]
        graph = """prep => init<run> => model<run,mem> =>
                    post<run,mem> => wrap<run> => done"""
```

Figure 29 shows the result as visualized by `cylc graph`.

9.6.1.1 Zero-Padded Integer Values

Integer parameter values are zero-padded according to the size of their largest value, so `foo<p>` for `p = 9..10` expands to `foo_p09, foo_p10`.

To get thicker padding, prepend extra zeroes to the upper range value: `foo<p>` for `p = 9..010` expands to `foo_p009, foo_p010`.

9.6.2 Passing Parameter Values To Tasks

Parameter values are passed as environment variables to tasks generated by parameter expansion. For the example above, task `model_run2_ship` would get the following environment variables:

```
$CYLC_TASK_PARAM_run   # value "2" etc.
$CYLC_TASK_PARAM_obs   # value "ship" etc.
```

These variables allow tasks to determine which member of a parameterized group they are, and so to vary their behaviour accordingly.

9.6.3 Selecting Specific Parameter Values

Specific parameter values can be singled out in the graph and under `[runtime]` with the notation `<p=5>` (for example). Here's how to make a special task trigger off just the first of a set of model runs:

```
[cylc]
    [[parameters]]
        run = 1..5
[scheduling]
    [[dependencies]]
        graph = """model<run> => post_proc<run>  # general case
                   model<run=1> => check_first_run  # special case"""
[runtime]
    [[model<run>]]
        # config for all "model" runs...
    [[model<run=1>
        # special config (if any) for the first model run...
    #...
```

9.6.4 Selecting Partial Parameter Ranges

The parameter notation does not currently support partial range selection such as `foo<p=5..10>`, but you can achieve the same result by defining a second parameter that covers the partial range and giving it the same expansion template as the full-range parameter. For example:

```
[cylc]
    [[parameters]]
        run = 1..10 # 01, 02, ..., 10
        runx = 1..03  # 01, 02, 03 (note '03' to get correct padding)
    [[parameter templates]]
        run = _R%(run)s
        runx = _R%(runx)s
[scheduling]
    [[dependencies]]
        graph = """model<run> => post<run>
                   model<runx> => checkx<runx>"""
[runtime]
    [[model<run>]]
```

```
        # ...
    #...
```

9.6.5   Parameter Offsets In The Graph

A negative offset notation `<NAME-1>` is interpreted as the previous value in the ordered list of
parameter values. For example, to split a model run into multiple steps with each step depending
on the previous one, this graph:

```
    graph = "model<run-1> => model<run>"  # for run = 1, 2, 3
```

expands to:

```
    graph = """model_run1 => model_run2
               model_run2 => model_run3"""
# or equivalently:
    graph = "model_run1 => model_run2 => model_run3"
```

And this graph:

```
    graph = "proc<size-1> => proc<size>"  # for size = small, big, huge
```

expands to:

```
    graph = """proc_small => proc_big
               proc_big => proc_huge"""
# or equivalently:
    graph = "proc_small => proc_big => proc_huge"
```

9.6.6   Task Families And Parameterization

Task family members can be generated by parameter expansion:

```
[runtime]
    [[FAM]]
    [[member<r>]]
        inherit = FAM
# Result: family FAM contains member_r1, member_r2, etc.
```

Family names can be parameterized too, just like task names:

```
[runtime]
    [[RUN<r>]]
    [[model<r>]]
        inherit = RUN<r>
    [[post_proc<r>]]
        inherit = RUN<r>
# Result: family RUN_r1 contains model_r1 and post_proc_r1,
#         family RUN_r2 contains model_r2 and post_proc_r1, etc.
```

As described in Section 9.3.5.9 family names can be used to trigger all members at once:

```
    graph = "foo => FAMILY"
```

or to trigger off all members:

```
    graph = "FAMILY:succeed-all => bar"
```

or to trigger off any members:

```
    graph = "FAMILY:succeed-any => bar"
```

If the members of `FAMILY` were generated with parameters, you can also trigger them all at once with parameter notation:

```
graph = "foo => member<m>"
```

Similarly, to trigger off all members:

```
graph = "member<m> => bar"
# (member<m>:fail etc., for other trigger types)
```

Family names are still needed in the graph, however, to succinctly express "succeed-any" triggering semantics, and all-to-all or any-to-all triggering:

```
graph = "FAM1:succeed-any => FAM2"
```

(Direct all-to-all and any-to-all family triggering is not recommended for efficiency reasons though - see Section 9.3.5.10).

For family *member-to-member* triggering use parameterized members. For example, if family `OBS_GET` has members `get<obs>` and family `OBS_PROC` has members `proc<obs>` then this graph:

```
graph = "get<obs> => proc<obs>"  # for obs = ship, buoy, plane
```

expands to:

```
get_ship => proc_ship
get_buoy => proc_buoy
get_plane => proc_plane
```

9.6.7   Parameterized Cycling

Two ways of constructing cycling systems are described and contrasted in Section 5. For most purposes use of a proper *cycling workflow* is recommended, wherein cylc incrementally generates the date-time sequence and extends the workflow, potentially indefinitely, at run time. For smaller systems of finite duration, however, parameter expansion can be used to generate a sequence of pre-defined tasks as a proxy for cycling.

Here's a cycling workflow of two-monthly model runs for one year, with previous-instance model dependence (e.g. for model restart files):

```
[scheduling]
    initial cycle point = 2020-01
    final cycle point = 2020-12
    [[dependencies]]
        [[[R1]]]  # Run once, at the initial point.
            graph = "prep => model"
        [[[P2M]]]  # Run at 2-month intervals between the initial and final points.
            graph = "model[-P2M] => model => post_proc & archive"
[runtime]
    [[model]]
        script = "run-model $CYLC_TASK_CYCLE_POINT"
```

And here's how to do the same thing with parameterized tasks:

```
[cylc]
    [[parameters]]
        chunk = 1..6
[scheduling]
    [[dependencies]]
        graph = """prep => model<chunk=1>
                  model<chunk-1> => model<chunk> =>
                    post_proc<chunk> & archive<chunk>"""
[runtime]
```
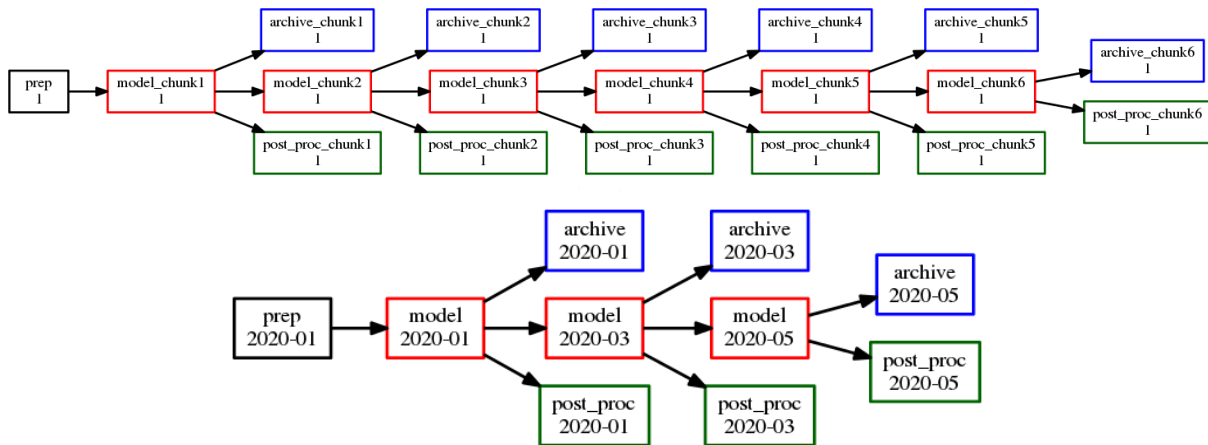
Figure 30: parameterized and cycling versions of the same workflow. The first three cycle points are shown in the cycling case. The parameterized case does not have "cycle points".

```
    [[model<chunk>]]
        script = """
# Compute start date from chunk index and interval, then run the model.
INITIAL_POINT=2020-01
INTERVAL_MONTHS=2
OFFSET_MONTHS=(( (CYLC_TASK_PARAM_chunk - 1)*INTERVAL_MONTHS ))
OFFSET=P${OFFSET_MONTHS}M  # e.g. P4M for chunk=3
run-model $(cylc cyclepoint --offset=$OFFSET $INITIAL_POINT)"""
```

The two workflows are shown together in Figure 30. They both achieve the same result, and both can include special tasks at the start, end, or anywhere in between. But as noted earlier the parameterized version has several disadvantages: it must be finite in extent and not too large; the date-time arithmetic has to be done by the user; and the full extent of the workflow will be visible at all times as the suite runs.

Here's a yearly-cycling suite with four parameterized chunks in each cycle point:

```
[cylc]
    [[parameters]]
        chunk = 1..4
[scheduling]
    initial cycle point = 2020-01
    [[dependencies]]
        [[[P1Y]]]
            graph = """model<chunk-1> => model<chunk>
                    model<chunk=4>[-P1Y] => model<chunk=1>"""
```

Note the inter-cycle trigger that connects the first chunk in each cycle point to the last chunk in the previous cycle point. Of course it would be simpler to just use 3-monthly cycling:

```
[scheduling]
    initial cycle point = 2020-01
    [[dependencies]]
        [[[P3M]]]
            graph = "model[-P3M] => model"
```

Here's a possible valid use-case for mixed cycling: consider a portable date-time cycling workflow of model jobs that can each take too long to run on some supported platforms. This could be handled without changing the cycling structure of the suite by splitting the run (at each cycle point) into a variable number of shorter steps, using more steps on less powerful hosts.

9.6.7.1   Cycle Point And Parameter Offsets At Start-Up

In cycling workflows cylc ignores anything earlier than the suite initial cycle point. So this graph:

```
    graph = "model[-P1D] => model"
```

simplifies at the initial cycle point to this:

```
    graph = "model"
```

Similarly, parameter offsets are ignored if they extend beyond the start of the parameter value list. So this graph:

```
    graph = "model<chunk -1> => model<chunk >"
```

simplifies for `chunk=1` to this:

```
    graph = "model_chunk1"
```

Note however that the initial cut-off applies to every parameter list, but only to cycle point sequences that start at the suite initial cycle point. Therefore it may be somewhat easier to use parameterized cycling if you need multiple date-time sequences *with different start points* in the same suite. We plan to allow this sequence-start simplification for any date-time sequence in the future, not just at the suite initial point, but it needs to be optional because delayed-start cycling tasks sometimes need to trigger off earlier cycling tasks.

9.7   Jinja2

This section needs to be revised - the Parameterized Task feature introduced in cylc-6.11.0 (see Section 9.6) provides a cleaner way to auto-generate tasks without coding messy Jinja2 loops.

Cylc has built in support for the Jinja2 template processor in suite definitions. Jinja2 variables, mathematical expressions, loop control structures, conditional logic, etc., are automatically processed to generate the final suite definition seen by cylc.

The need for Jinja2 processing must be declared with a hash-bang comment as the first line of the suite.rc file:

```
#!jinja2
# ...
```

Potential uses for this include automatic generation of repeated groups of similar tasks and dependencies, and inclusion or exclusion of entire suite sections according to the value of a single flag. Consider a large complicated operational suite and several related parallel test suites with slightly different task content and structure (the parallel suites, for instance, might take certain large input files from the operation or the archive rather than downloading them again) - these can now be maintained as a single master suite definition that reconfigures itself according to the value of a flag variable indicating the intended use.

Template processing is the first thing done on parsing a suite definition so Jinja2 expressions can appear anywhere in the file (inside strings and namespace headings, for example).

Jinja2 is well documented at http://jinja.pocoo.org/docs, so here we just provide an example suite that uses it. The meaning of the embedded Jinja2 code should be reasonably self-evident to anyone familiar with standard programming techniques.
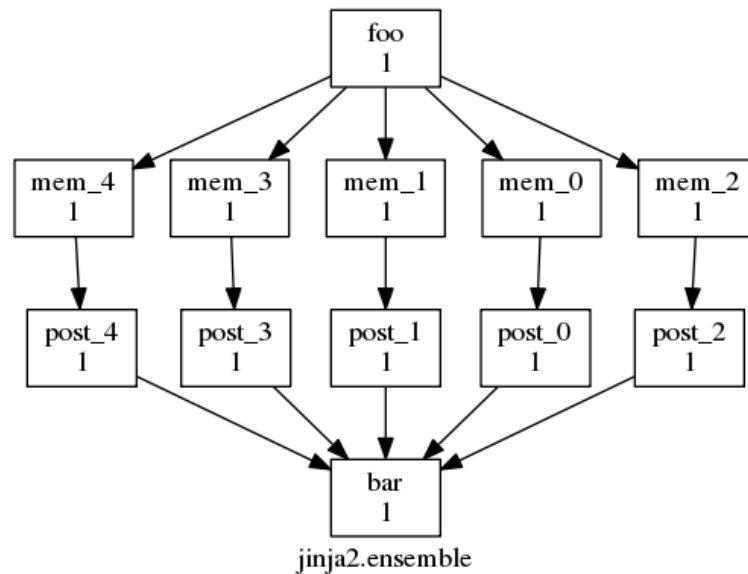
Figure 31: The Jinja2 ensemble example suite graph.

The `jinja2.ensemble` example, graphed in Figure 31, shows an ensemble of similar tasks generated using Jinja2:

```
#!jinja2
{% set N_MEMBERS = 5 %}
[scheduling]
    [[dependencies]]
        graph = """{# generate ensemble dependencies #}
            {% for I in range( 0, N_MEMBERS ) %}
                foo => mem_{{ I }} => post_{{ I }} => bar
            {% endfor %}"""
```

Here is the generated suite definition, after Jinja2 processing:

```
#!jinja2
[scheduling]
    [[dependencies]]
        graph = """
            foo => mem_0 => post_0 => bar
            foo => mem_1 => post_1 => bar
            foo => mem_2 => post_2 => bar
            foo => mem_3 => post_3 => bar
            foo => mem_4 => post_4 => bar
                """
```

And finally, the `jinja2.cities` example uses variables, includes or excludes special cleanup tasks according to the value of a logical flag, and it automatically generates all dependencies and family relationships for a group of tasks that is repeated for each city in the suite. To add a new city and associated tasks and dependencies simply add the city name to list at the top of the file. The suite is graphed, with the New York City task family expanded, in Figure 32.

```
#!Jinja2

title = "Jinja2 city suite example."
description = """
Illustrates use of variables and math expressions, and programmatic
generation of groups of related dependencies and runtime properties."""

{% set HOST = "SuperComputer" %}
{% set CITIES = 'NewYork', 'Philadelphia', 'Newark', 'Houston', 'SantaFe', 'Chicago' %}
{% set CITYJOBS = 'one', 'two', 'three', 'four' %}
```
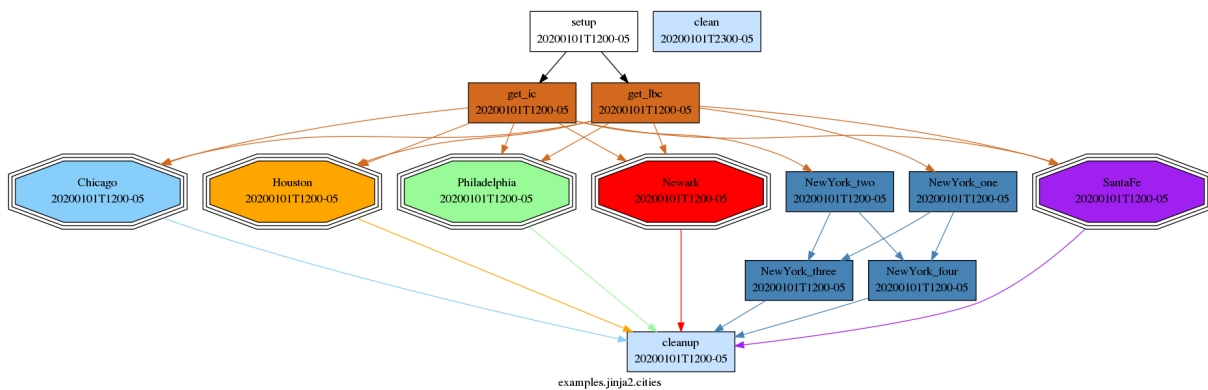
Figure 32: The Jinja2 cities example suite graph, with the New York City task family expanded.

```
{% set LIMIT_MINS = 20 %}

{% set CLEANUP = True %}

[scheduling]
    initial cycle point = 2011-08-08T12
    [[ dependencies ]]
{% if CLEANUP %}
        [[[T23]]]
            graph = "clean"
{% endif %}
        [[[T00,T12]]]
            graph = """
                setup => get_lbc & get_ic # foo
{% for CITY in CITIES %} {# comment #}
                get_lbc => {{ CITY }}_one
                get_ic => {{ CITY }}_two
                {{ CITY }}_one & {{ CITY }}_two => {{ CITY }}_three & {{ CITY }}_four
{% if CLEANUP %}
                {{ CITY }}_three & {{ CITY }}_four => cleanup
{% endif %}
{% endfor %}
                """
[runtime]
    [[on_{{ HOST }} ]]
        [[[remote]]]
            host = {{ HOST }}
            # (remote cylc directory is set in site/user config for this host)
        [[[directives]]]
            wall_clock_limit = "00:{{ LIMIT_MINS|int() + 2 }}:00,00:{{ LIMIT_MINS }}:00"

{% for CITY in CITIES %}
    [[ {{ CITY }} ]]
        inherit = on_{{ HOST }}
{% for JOB in CITYJOBS %}
    [[ {{ CITY }}_{{ JOB }} ]]
        inherit = {{ CITY }}
{% endfor %}
{% endfor %}

[visualization]
    initial cycle point = 2011-08-08T12
    final cycle point = 2011-08-08T23
    [[node groups]]
        cleaning = clean, cleanup
    [[node attributes]]
        cleaning = 'style=filled', 'fillcolor=yellow'
        NewYork = 'style=filled', 'fillcolor=lightblue'
```

9.7.1   Accessing Environment Variables With Jinja2

This functionality is not provided by Jinja2 by default, but cylc automatically imports the user
environment to the template in a dictionary structure called *environ*. A usage example:

```
#!Jinja2
#...
[runtime]
    [[root]]
        [[[environment]]]
            SUITE_OWNER_HOME_DIR_ON_SUITE_HOST = {{environ['HOME']}}
```

This example is emphasizes that *the environment is read on the suite host at the time the suite
definition is parsed* - it is not, for instance, read at task run time on the task host.

9.7.2   Custom Jinja2 Filters

Jinja2 variable values can be modified by "filters", using pipe notation. For example, the built-in
`trim` filter strips leading and trailing white space from a string:

```
{% set MyString = "   dog   " %}
{{ MyString | trim() }}  # "dog"
```

(See official Jinja2 documentation for available built-in filters.)

Cylc also supports custom Jinja2 filters. A custom filter is a single Python function in a source
file with the same name as the function (plus ".py" extension) and stored in one of the following
locations:

- `$CYLC_DIR/lib/Jinja2Filters/`
- `[suite definition directory]/Jinja2Filters/`
- `$HOME/.cylc/Jinja2Filters/`

In the filter function argument list, the first argument is the variable value to be "filtered", and
subsequent arguments can be whatever is needed. Currently there are two custom filters:

9.7.2.1   pad

The "pad" filter is for padding string values to some constant length with a fill character - useful
for generating task names and related values in ensemble suites:

```
{% for i in range(0,100) %}  # 0, 1, ..., 99
    {% set j = i | pad(2,'0') %}
    A_{{j}}             # A_00, A_01, ..., A_99
{% endfor %}
```

9.7.2.2   strftime

The "strftime" filter can be used to format ISO8601 date-time strings using an strftime string.

```
{% set START_CYCLE = '10661004T08+01' %}
{{ START_CYCLE | strftime('%H') }}  # 00
```

Examples:

- `{{START_CYCLE | strftime('%Y')}}` - 1066

- `{{START_CYCLE | strftime('%m')}}` - 10
- `{{START_CYCLE | strftime('%d')}}` - 14
- `{{START_CYCLE | strftime('%H:%M:%S %z')}}` - 08:00:00 +01

It is also possible to parse non-standard date-time strings by passing a strptime string as the second argument.

Examples:

- `{{'12,30,2000' | strftime('%m', '%m,%d,%Y')}}` - 12
- `{{'1066/10/14 08:00:00' | strftime('%Y%m%dT%H', '%Y/%m/%d %H:%M:%S')}}` - 10661014T08

9.7.3   Associative Arrays In Jinja2

Associative arrays (*dicts* in Python) can be very useful. Here's an example, from `$CYLC_DIR/examples/jinja2/dict`:

```
#!Jinja2
{% set obs_types = ['airs', 'iasi'] %}
{% set resource = { 'airs':'ncpus=9', 'iasi':'ncpus=20' } %}

[scheduling]
    [[dependencies]]
        graph = OBS
[runtime]
    [[OBS]]
        [[[job]]]
            batch system = pbs
    {% for i in obs_types %}
    [[ {{i}} ]]
        inherit = OBS
        [[[directives]]]
            -I = {{ resource[i] }}
     {% endfor %}
```

Here's the result:
```
$ cylc get-suite-config -i [runtime][airs]directives SUITE
-I = ncpus=9
```

9.7.4   Jinja2 Default Values And Template Inputs

The values of Jinja2 variables can be passed in from the cylc command line rather than hardwired in the suite definition. Here's an example, from `$CYLC_DIR/examples/jinja2/defaults`:

```
#!Jinja2

title = "Jinja2 example: use of defaults and external input"

description = """
The template variable FIRST_TASK must be given on the cylc command line
using --set or --set-file=FILE; two other variables, LAST_TASK and
N_MEMBERS can be set similarly, but if not they have default values."""

{% set LAST_TASK = LAST_TASK | default( 'baz' ) %}
{% set N_MEMBERS = N_MEMBERS | default( 3 ) | int %}

{# input of FIRST_TASK is required - no default #}

[scheduling]
    initial cycle point = 20100808T00
```

```
    final cycle point   = 20100816T00
    [[dependencies]]
        [[[0]]]
            graph = """{{ FIRST_TASK }} => ENS
                  ENS:succeed-all => {{ LAST_TASK }}"""
[runtime]
    [[ENS]]
{% for I in range( 0, N_MEMBERS ) %}
    [[ mem_{{ I }} ]]
        inherit = ENS
{% endfor %}
```

Here's the result:

```
$ cylc list SUITE
Jinja2 Template Error
'FIRST_TASK' is undefined
cylc-list foo  failed:  1

$ cylc list --set FIRST_TASK=bob foo
bob
baz
mem_2
mem_1
mem_0

$ cylc list --set FIRST_TASK=bob --set LAST_TASK=alice foo
bob
alice
mem_2
mem_1
mem_0

$ cylc list --set FIRST_TASK=bob --set N_MEMBERS=10 foo
mem_9
mem_8
mem_7
mem_6
mem_5
mem_4
mem_3
mem_2
mem_1
mem_0
baz
bob
```

Note also that `cylc view --set FIRST_TASK=bob --jinja2 SUITE` will show the suite with the Jinja2 variables as set.

Note: suites started with template variables set on the command line will *restart* with the same settings. However, you can set them again on the `cylc restart` command line if they need to be overridden.

9.7.5   Jinja2 Variable Scope

Jinja2 variable scoping rules may be surprising. Variables set inside a *for loop* block, for instance, are not accessible outside of the block, so the following will print `# FOO is 0`, not `# FOO is 9`:

```
{% set COUNT = 0 %}
{% for I in range(10) %}
    {% set FOO = I %}
{% endfor %}
# FOO is {{FOO}}
```

9.7.6 Raising Exceptions

Cylc provides two functions for raising exceptions using Jinja2. These exceptions are raised when the suite.rc file is loaded and will prevent a suite from running.

Note: These functions must be contained within `{{` Jinja2 blocks as opposed to `{%` blocks.

9.7.6.1 Raise

The "raise" function will result in an error containing the provided text.

```
{% if not VARIABLE is defined %}
    {{ raise('VARIABLE must be defined for this suite.') }}
{% endif %}
```

9.7.6.2 Assert

The "assert" function will raise an exception containing the text provided in the second argument providing that the first argument evaluates as False. The following example is equivalent to the "raise" example above.

```
{{ assert(VARIABLE is defined, 'VARIABLE must be defined for this suite.') }}
```

9.8 Omitting Tasks At Runtime

It is sometimes convenient to omit certain tasks from the suite at runtime without actually deleting their definitions from the suite.

Defining [runtime] properties for tasks that do not appear in the suite graph results in verbose-mode validation warnings that the tasks are disabled. They cannot be used because the suite graph is what defines their dependencies and valid cycle points. Nevertheless, it is legal to leave these orphaned runtime sections in the suite definition because it allows you to temporarily remove tasks from the suite by simply commenting them out of the graph.

To omit a task from the suite at runtime but still leave it fully defined and available for use (by insertion or `cylc submit`) use one or both of [scheduling][[special task]] lists, *include at start-up* or *exclude at start-up* (documented in A.3.11.6 and A.3.11.5). Then the graph still defines the validity of the tasks and their dependencies, but they are not actually loaded into the suite at start-up. Other tasks that depend on the omitted ones, if any, will have to wait on their insertion at a later time or otherwise be triggered manually.

Finally, with Jinja2 (9.7) you can radically alter suite structure by including or excluding tasks from the [scheduling] and [runtime] sections according to the value of a single logical flag defined at the top of the suite.

9.9 Naked Dummy Tasks And Strict Validation

A *naked dummy task* appears in the suite graph but has no explicit runtime configuration section. Such tasks automatically inherit the default "dummy task" configuration from the root namespace. This is very useful because it allows functional suites to be mocked up quickly

for test and demonstration purposes by simply defining the graph. It is somewhat dangerous, however, because there is no way to distinguish an intentional naked dummy task from one generated by typographic error: misspelling a task name in the graph results in a new naked dummy task replacing the intended task in the affected trigger expression; and misspelling a task name in a runtime section heading results in the intended task becoming a dummy task itself (by divorcing it from its intended runtime config section).

To avoid this problem any dummy task used in a real suite should not be naked - i.e. it should have an explicit entry in under the runtime section of the suite definition, even if the section is empty. This results in exactly the same dummy task behaviour, via implicit inheritance from root, but it allows use of `cylc validate --strict` to catch errors in task names by failing the suite if any naked dummy tasks are detected.

10   Task Implementation

Existing scripts or executables can be used as cylc tasks without any modification so long as they return standard exit status, i.e. zero on success and non-zero for failure, and do not internally spawn detaching processes (see 10.5).

10.1   Task Job Scripts

When a task is ready cylc generates a *task job script* that encapsulates the runtime settings (environment and scripts) defined for it in the suite.rc file. The job script is submitted to run by the *batch system* chosen for the task (see 11). A task job script also exports a set of environment variables prior to executing the task `script` (see 9.4.7).

As shown in the Tutorial (7.11) job scripts are saved to the suite run directory; the commands used to submit them are printed to stdout by cylc in debug mode; and they can be printed with the `cylc cat-log` command or new ones generated and printed with the `cylc jobscript` command. Take a look at one to see exactly how cylc wraps and runs your tasks.

10.2   Inlined Tasks

Tasks that can be performed by simple commands written in `bash` can be entirely implemented within the suite.rc file, as the task *script* items can be multi-line strings.

10.3   Returning Proper Error Status

A task job script in cylc automatically traps ERR (for errors) and EXIT (for unexpected exits). When the trap is triggered, the task job script will send a failed message back to the suite before aborting.

Therefore, commands in user scripts should abort with non-zero exit status if a fatal error occurs (this is just good coding practice anyway). To prevent a command that expects a non-zero return code from triggering a trap, you can use a control statement such as:

```
if cmp FILE1 FILE2; then
    :  # do stuff
else
```

```
    :   # do alternate stuff
fi
```

In addition, the task job script has `set -u` and `set -o pipefail` turned on explicitly to ensure that any variable must be defined on reference (which is very useful for picking typos) and that a command fails if any part of a pipe fails.

10.4   Other Task Messages

General (non-output) messages can also be sent to report progress, warnings, and so on, e.g.:

```
#!/bin/bash
# a warning message (this will be logged by the suite):
cylc task message -p WARNING "oops, something's fishy here"
# information (this will also be logged by the suite):
cylc task message "Hello from task foo"
```

Explanatory messages can be sent before aborting on error:

```
#!/bin/bash
set -e  # abort on error
if ! mkdir /illegal/dir; then
    # (use inline error checking to avoid triggering the above 'set -e')
    cylc task message -p CRITICAL "Failed to create directory /illegal/dir"
    exit 1 # now abort non-zero exit status to trigger the task failed message
fi
```

Or equivalently, with different syntax:

```
#!/bin/bash
set -e
mkdir /illegal/dir || {  # inline error checking using OR operator
    cylc task message -p CRITICAL "Failed to create directory /illegal/dir"
    exit 1
}
```

But not this:

```
#!/bin/bash
set -e
mkdir /illegal/dir  # aborted via ERR trap
if [[ $? != 0 ]]; then  # so this will never be reached.
    cylc task message -p CRITICAL "Failed to create directory /illegal/dir"
    exit 1
fi
```

10.5   Avoid Detaching Processes

If a task script starts background sub-processes and does not wait on them, or internally submits jobs to a batch scheduler and then exits immediately, the detached processes will not be visible to cylc and the task will appear to finish when the top-level script finishes. You will need to modify scripts like this to make them execute all sub-processes in the foreground (or use the shell `wait` command to wait on them before exiting) and to prevent job submission commands from returning before the job completes (e.g. `llsubmit -s` for Loadleveler, `qsub -sync yes` for Sun Grid Engine, and `qsub -W block=true` for PBS).

If this is not possible - perhaps you don't have control over the script or can't work out how to fix it - one alternative approach is to use another task to repeatedly poll for the results of the detached processes:

```
[scheduling]
    [[dependencies]]
        graph = "model => checker => post-proc"
[runtime]
    [[model]]
        # Uh-oh, this script does an internal job submission to run model.exe:
        script = "run-model.sh"
    [[checker]]
        # Fail and retry every minute (for 10 tries at the most) if the model's
        # job.done indicator file does not exist yet.
        retry delays = 10 * PT1M
        script = "[[ ! -f $RUN_DIR/job.done ]] && exit 1"
```

11   Task Job Submission and Management

For the requirements a command, script, or program, must fulfill in order to function as a cylc task, see 10. This section explains how tasks are submitted by the suite daemon when they are ready to run, and how to define new batch system handlers.

When a task is ready cylc generates a job script (see 10.1). The job script is submitted to run by the *batch system* chosen for the task. Different tasks can use different batch systems. Like other runtime properties, you can set a suite default batch system and override it for specific tasks or families:

```
[runtime]
    [[root]] # suite defaults
        [[[job]]]
            batch system = loadleveler
    [[foo]] # just task foo
        [[[job]]]
            batch system = at
```

11.1   Supported Job Submission Methods

Cylc supports a number of commonly used batch systems. See 11.7 for how to add new job submission methods.

11.1.1   background

Runs task job scripts as Unix background processes.

If an execution time limit is specified for a task, its job will be wrapped by the `timeout` command.

11.1.2   at

Submits task job scripts to the rudimentary Unix `at` scheduler. The `atd` daemon must be running.

If an execution time limit is specified for a task, its job will be wrapped by the `timeout` command.

11.1.3   loadleveler

Submits task job scripts to loadleveler by the `llsubmit` command. Loadleveler directives can be provided in the suite.rc file:

```
[runtime]
    [[my_task]]
        [[[job]]]
            batch system = loadleveler
            execution time limit = PT10M
        [[[directives]]]
            foo = bar
            baz = qux
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
# @ foo = bar
# @ baz = qux
# @ wall_clock_limit = 660,600
# @ queue
```

If restart=yes is specified as a directive for loadleveler, the job will automatically trap SIGUSR1, which loadleveler may use to preempt the job. On trapping SIGUSR1, the job will inform the suite that it has been vacated by loadleveler. This will put it back to the submitted state, until it starts running again.

If `execution time limit` is specified, it is used to generate the `wall_clock_limit` directive. The setting is assumed to be the soft limit. The hard limit will be set by adding an extra minute to the soft limit. Do not specify the `wall_clock_limit` directive explicitly if `execution time limit` is specified. Otherwise, the execution time limit known by the suite may be out of sync with what is submitted to the batch system.

11.1.4   lsf

Submits task job scripts to IBM Platform LSF by the `bsub` command. LSF directives can be provided in the suite.rc file:

```
[runtime]
    [[my_task]]
        [[[job]]]
            batch system = lsf
            execution time limit = PT10M
        [[[directives]]]
            -q = foo
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
#BSUB -q = foo
#BSUB -W = 10
```

If `execution time limit` is specified, it is used to generate the `-W` directive. Do not specify the `-W` directive explicitly if `execution time limit` is specified. Otherwise, the execution time limit known by the suite may be out of sync with what is submitted to the batch system.

11.1.5   pbs

Submits task job scripts to PBS (or Torque) by the `qsub` command. PBS directives can be provided in the suite.rc file:

```
[runtime]
    [[my_task]]
        [[[job]]]
            batch system = pbs
            execution time limit = PT1M
        [[[directives]]]
            -V =
            -q = foo
            -l nodes = 1
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
#PBS -V
#PBS -q foo
#PBS -l nodes=1
#PBS -l walltime=60
```

If `execution time limit` is specified, it is used to generate the `-l walltime` directive. Do not specify the `-l walltime` directive explicitly if `execution time limit` is specified. Otherwise, the execution time limit known by the suite may be out of sync with what is submitted to the batch system.

11.1.6   moab

Submits task job scripts to the Moab workload manager by the `msub` command. Moab directives can be provided in the suite.rc file; the syntax is very similar to PBS:

```
[runtime]
    [[my_task]]
        [[[job]]]
            batch system = moab
            execution time limit = PT1M
        [[[directives]]]
            -V =
            -q = foo
            -l nodes = 1
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
#PBS -V
#PBS -q foo
#PBS -l nodes=1
#PBS -l walltime=60
```

(Moab understands `#PBS` directives).

If `execution time limit` is specified, it is used to generate the `-l walltime` directive. Do not specify the `-l walltime` directive explicitly if `execution time limit` is specified. Otherwise, the execution time limit known by the suite may be out of sync with what is submitted to the batch system.

11.1.7   sge

Submits task job scripts to Sun/Oracle Grid Engine by the `qsub` command. SGE directives can be provided in the suite.rc file:

```
[runtime]
    [[my_task]]
        [[[job]]]
            batch system = sge
            execution time limit = P1D
        [[[directives]]]
            -cwd =
            -q = foo
            -l h_data = 1024M
            -l h_rt = 24:00:00
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
#$ -cwd
#$ -q foo
#$ -l h_data=1024M
#$ -l h_rt=24:00:00
```

If `execution time limit` is specified, it is used to generate the `-l h_rt` directive. Do not specify the `-l h_rt` directive explicitly if `execution time limit` is specified. Otherwise, the execution time limit known by the suite may be out of sync with what is submitted to the batch system.

11.1.8   slurm

Submits task job scripts to Simple Linux Utility for Resource Management by the `sbatch` command. SLURM directives can be provided in the suite.rc file (note that since not all SLURM commands have a short form, cylc requires the long form directives):

```
[runtime]
    [[my_task]]
        [[[job]]]
            batch system = slurm
            execution time limit = PT1H
        [[[directives]]]
            --nodes = 5
            --account = QXZ5W2
```

These are written to the top of the task job script like this:

```
#!/bin/bash
#SBATCH --nodes=5
#SBATCH --time=60:00
#SBATCH --account=QXZ5W2
```

If `execution time limit` is specified, it is used to generate the `--time` directive. Do not specify the `--time` directive explicitly if `execution time limit` is specified. Otherwise, the execution time limit known by the suite may be out of sync with what is submitted to the batch system.

11.1.9   Default Directives Provided

For batch systems that use job file directives (PBS, Loadleveler, etc.) default directives are provided to set the job name, stdout and stderr file paths, and the execution time limit (if specified).

Cylc constructs the job name string using a combination of the task ID and the suite name. PBS fails a job submit if the job name in `-N name` is too long. For version 12 or below, this is 15 characters. For version 13, this is 236 characters. The default setting will truncate the job

name string to 15 characters. If you have PBS 13 at your site, you should modify your site's global configuration file to allow the job name to be longer. (See also Section B.9.1.19.5.) For example:

```
[hosts]
    [[myhpc*]]
        [[[batch systems]]]
            [[[[pbs]]]]
                # PBS 13
                job name length maximum = 236
```

11.1.10   Directives Section Quirks (PBS, SGE, ...)

To specify an option with no argument, such as `-V` in PBS or `-cwd` in SGE you must give a null string as the directive value in the suite.rc file.

The left hand side of a setting (i.e. the string before the first equal sign) must be unique. To specify multiple values using an option such as `-l` option in PBS, SGE, etc., either specify all items in a single line:

```
-l=select=28:ncpus=36:mpiprocs=18:ompthreads=2:walltime=12:00:00
```

(Left hand side is `-l`. A second `-l=...` line will override the first.)

Or separate the items (note: no equal sign after `-l`):

```
-l select=28
-l ncpus=36
-l mpiprocs=18
-l ompthreads=2
-l walltime=12:00:00
```

(Left hand sides are now `-l select`, `-l ncpus`, etc.)

11.2   Task stdout And stderr Logs

When a task is ready to run cylc generates a filename root to be used for the task job script and log files. The filename containing the task name, cycle point, and a submit number that increments if the same task is re-triggered multiple times:

```
# task job script:
~/cylc-run/tut/oneoff/basic/log/job/1/hello/01/job
# task stdout:
~/cylc-run/tut/oneoff/basic/log/job/1/hello/01/job.out
# task stderr:
~/cylc-run/tut/oneoff/basic/log/job/1/hello/01/job.err
```

How the stdout and stderr streams are directed into these files depends on the batch system. The `background` method just uses appropriate output redirection on the command line, as shown above. The `loadleveler` method writes appropriate directives to the job script that is submitted to loadleveler.

Cylc obviously has no control over the stdout and stderr output from tasks that do their own internal output management (e.g. tasks that submit internal jobs and direct the associated output to other files). For less internally complex tasks, however, the files referred to here will be complete task job logs.

Some batch systems, such as `pbs`, redirect a job's stdout and stderr streams to a separate cache area while the job is running. The contents are only copied to the normal locations when the job completes. This means that `cylc cat-log` or the gcylc GUI will be unable to find the job's stdout and stderr streams while the job is running. Some sites with these batch systems are known to provide commands for viewing and/or tail-follow a job's stdout and stderr streams that are redirected to these cache areas. If this is the case at your site, you can configure cylc to make use of the provided commands by adding some settings to the global site/user config. E.g.:

```
[hosts]
    [[HOST]]   # <= replace this with a real host name
        [[[batch systems]]]
            [[[[pbs]]]]
                err tailer = qcat -f -e \%(job_id)s
                out tailer = qcat -f -o \%(job_id)s
                err viewer = qcat -e \%(job_id)s
                out viewer = qcat -o \%(job_id)s
```

11.3   Overriding The Job Submission Command

To change the form of the actual command used to submit a job you do not need to define a new batch system handler; just override the `command template` in the relevant job submission sections of your suite.rc file:

```
[runtime]
    [[root]]
        [[[job]]]
            batch system = loadleveler
            # Use '-s' to stop llsubmit returning
            # until all job steps have completed:
            batch submit command template = llsubmit -s %(job)s
```

As explained in A the template's %(job)s will be substituted by the job file path.

11.4   Job Polling

For supported batch systems, one-way polling can be used to determine actual job status: the suite daemon executes a process on the task host, by non-interactive ssh, to interrogate the batch queueing system there, and to read a *status file* that is automatically generated by the task job script as it runs.

Polling may be required to update the suite state correctly after unusual events such as a machine being rebooted with tasks running on it, or network problems that prevent task messages from getting back to the suite host.

Tasks can be polled on demand by right-clicking on them in gcylc or using the `cylc poll` command.

Tasks are polled automatically, once, if they timeout while queueing in a batch scheduler and submission timeout is set. (See A.4.1.15 for how to configure timeouts).

Tasks are polled multiple times, where necessary, when they exceed their execution time limits. These are normally set with some initial delays to allow the batch systems to kill the jobs. (See B.9.1.19.6 for how to configure the polling intervals).

Any tasks recorded in the *submitted* or *running* states at suite restart are automatically polled to determine what happened to them while the suite was down.

Regular polling can also be configured as a health check on tasks submitted to hosts that are known to be flaky, or as the sole method of determining task status on hosts that do not allow task messages to be routed back to the suite host.

To use polling instead of task-to-suite messaging set `task communication method = poll` in cylc site and user global config (see B.9.1.3). The default polling intervals can be overridden for all suites there too (see B.9.1.5 and B.9.1.4), or in specific suite definitions (in which case polling will be done regardless of the task communication method configured for the host; see A.4.1.13.7 and A.4.1.13.8).

Note that regular polling is not as efficient as task messaging in updating task status, and it should be used sparingly in large suites.

Note that for polling to work correctly, the batch queueing system must have a job listing command for listing your jobs, and that the job listing must display job IDs as they are returned by the batch queueing system submit command. For example, for pbs, moab and sge, the `qstat` command should list jobs with their IDs displayed in exactly the same format as they are returned by the `qsub` command.

11.5   Job Killing

For supported batch systems, the suite daemon can execute a process on the task host, by non-interactive ssh, to kill a submitted or running job according to its batch system.

Tasks can be killed on demand by right-clicking on them in gcylc or using the `cylc kill` command.

11.6   Execution Time Limit

You can specify an `execution time limit` for all supported job submission methods. E.g.:

```
[runtime]
    [[task-x]]
        [[[job]]]
            execution time limit = PT1H
```

For tasks running with `background` or `at`, their jobs will be wrapped using the `timeout` command. For all other methods, the relevant time limit directive will be added to their job files.

The `execution time limit` setting will also inform the suite when a a task job should complete by. If a task job has not reported completing within the specified time, the suite will poll the task job. (The default setting is PT1M, PT2M, PT7M. The accumulated times for these intervals will be roughly 1 minute, $1 + 2 = 3$ minutes and $1 + 2 + 7 = 10$ minutes after a task job exceeds its execution time limit.)

11.6.1   Execution Time Limit and Execution Timeout

If you specify an `execution time limit` the `execution timeout event handler` will only be called if the job has not completed after the final poll (by default, 10 min after the time limit). This

should only happen if the submission method you are using is not enforcing wallclock limits (unlikely) or you are unable to contact the machine to confirm the job status.

If you specify an `execution timeout` and not an `execution time limit` then the `execution timeout event handler` will be called as soon as the specified time is reached. The job will also be polled to check its latest status (possibly resulting in an update in its status and the calling of the relevant event handler). This behaviour is deprecated, which users should avoid using.

If you specify an `execution timeout` and an `execution time limit` then the execution timeout setting will be ignored.

11.7   Custom Job Submission Methods

Defining a new batch system handler requires a little Python programming. You can use one of the built-in methods as an example, and read the documentation in the header of the `cylc.batch_sys_manager` module.

11.7.1   An Example

The following user-defined job submission class, called *qsub*, overrides the built-in *pbs* class to change the directive prefix from `#PBS` to `#QSUB`:

```python
#!/usr/bin/env python

from cylc.batch_sys_handlers.pbs import PBSHandler


class QSUBHandler(PBSHandler):
    """A user defined batch system handler."""
    DIRECTIVE_PREFIX = "#QSUB "


BATCH_SYSTEM_HANDLER = QSUBHandler()
```

To check that this works correctly save the new source file to `qsub.py` in one of the allowed locations (see just below), use it in a suite definition:

```
# SUITE.rc
# $HOME/test/suite.rc
[scheduling]
    [[dependencies]]
        graph = "a"
[runtime]
    [[root]]
        [[[job]]]
            batch system = qsub
            execution time limit = PT1M
        [[[directives]]]
            -l nodes = 1
            -q = long
            -V =
```

and generate a job script to see the resulting directives:

```
$ cylc register test $HOME/test
$ cylc jobscript test a.1 | grep QSUB
#QSUB -e /home/oliverh/cylc-run/my.suite/log/job/1/a/01/job.err
#QSUB -l nodes=1
#QSUB -l walltime=60
#QSUB -o /home/oliverh/cylc-run/my.suite/log/job/1/a/01/job.out
#QSUB -N a.1
```

```
#QSUB -q long
#QSUB -V
```

11.7.2   Where To Put New Job Submission Modules

You new job submission class code should be saved to a file with the same name as the class (plus ".py" extension). It can reside in any of the following locations, depending on how generally useful the new method is and whether or not you have write-access to the cylc source tree:

- a `lib/python` sub-directory of your suite definition directory.
- any directory in your `$PYTHONPATH`.
- in the `lib/cylc/batch_sys_handlers` directory of the cylc source tree.

12   Running Suites

This chapter currently features a diverse collection of topics related to running suites. Please also see the Tutorial (7) and command documentation (E), and experiment with plenty of examples.

12.1   Suite Start-up

There are three ways to start a suite running: *cold start* and *warm start*, which start from scratch; and *restart*, which loads a prior suite state. There is no difference between cold and warm start, except that the latter starts from a point beyond the suite initial cycle point.

Once a suite is up and running it is typically a restart that is needed most often (but see also `cylc reload`). Be aware that cold and warm starts wipe out any prior suite state, which prevents returning to a restart if you decide that's what you really intended.

12.1.1   Cold Start

A cold start is the primary way to start a suite run from scratch:

```
$ cylc run SUITE [INITIAL_CYCLE_POINT]
```

The initial cycle point may be specified on the command line or in the suite.rc file. The scheduler starts by loading the first instance of each task at the suite initial cycle point, or at the next valid point for the task.

12.1.2   Restart

A restart starts a suite run from the state recorded at a checkpoint, which is normally the end of a previous run. This allows restarting a suite that was shut down or killed, without rerunning tasks that were already completed, or which were already submitted or running when the suite went down.

```
$ cylc restart SUITE
```

For a restart, the scheduler starts by loading each task in its recorded state. Any tasks recorded as 'submitted' or 'running' will be polled automatically to determine what happened to them while the suite was down.

See Section 12.8 for more detail.

12.1.3   Warm Start

A warm start runs a suite from scratch like a cold start, but from a given cycle point that is later than the suite's initial cycle point. All tasks from the given cycle point will run. It can be considered an inferior alternative to a restart because it may result in some tasks rerunning. A warm start may be required if a restart is not possible because the suite run databases were accidentally deleted (for instance). The warm start cycle point must be given on the command line:

```
$ cylc run --warm SUITE [START_CYCLE_POINT]
```

The original suite initial cycle point is preserved, but all tasks and dependencies before the given start cycle point are ignored.

The scheduler starts by loading a first instance of each task at the warm start cycle point, or at the next valid point for the task. `R1`-type tasks behave exactly the same as other tasks - if their cycle point is at or later than the given start cycle point, they will run; if not, they will be ignored.

12.2   How Tasks Interact With Running Suites

Cylc has three ways of tracking the progress of tasks, configured per task host in the site and user global config files (6). All three methods can be used on different task hosts within the same suite if necessary.

1. **task-to-suite messaging:** cylc job scripts encapsulate task scripting in a wrapper that automatically invokes messaging commands to report progress back to the suite. The messaging commands can be configured to work in two different ways:
   (a) **default:** direct messaging via network sockets using HTTPS.
   (b) **ssh:** for tasks hosts that block access to the network ports required, cylc can use non-interactive ssh to re-invoke task messaging commands on the suite host (where ultimately HTTPS is still used to connect to the server process).
2. **polling:** for task hosts that do not allow return routing to the suite host or ssh, cylc can poll tasks at configurable intervals, using non-interactive ssh.

The remote HTTPS communication method is the default because it is the most direct and efficient; the ssh method inserts an extra step in the process (command re-invocation on the suite host); and task polling is the least efficient because results are checked at predetermined intervals, not when task events actually occur.

12.2.1   Task Polling

Be careful to avoid spamming task hosts with polling commands. Each poll opens (and then closes) a new ssh connection.

Polling intervals are configurable under `[runtime]` because they should may depend on the expected execution time. For instance, a task that typically takes an hour to run might be polled every 10 minutes initially, and then every minute toward the end of its run. Interval values are used in turn until the last value, which is used repeatedly until finished:

```
[runtime]
    [[foo]]
        [[[job]]]
            # poll every minute in the 'submitted' state:
            submission polling intervals = PT1M
            # poll one minute after foo starts running, then every 10
            # minutes for 50 minutes, then every minute until finished:
            execution polling intervals = PT1M, 5*PT10M, PT1M
```

A list of intervals with optional multipliers can be used for both submission and execution polling, although a single value is probably sufficient for submission polling. If these items are not configured default values from site and user global config will be used for the polling task communication method; polling is not done by default under the other task communications methods (but it can still be used if you like).

Polling is also done automatically once on job submission timeout, and multiple times on exceeding the execution time limit, to see if the timed-out task has failed or not; and on suite restarts, to see what happened to any tasks that were orphaned when the suite went down.

12.3   Alternatives To Polling When Routing Is Blocked

If remote ports are blocked and non-interactive ssh doesn't work, but you don't want to use polling from the suite host:

- it has been suggested that network *port forwarding* may provide a solution;
- you may be able to persuade system administrators to provide network routing to one or more dedicated cylc servers;
- it is possible to run cylc itself on HPC login nodes, but depending on what software is installed there this may preclude use of the gcylc GUI and suite visualization tools.

12.4   Task Host Communications Configuration

Here are the default site and user global config items relevant to task state tracking (see these with `cylc get-site-config`):

```
# SITE AND USER CONFIG

# Task messaging settings affect task-to-suite communications.
[task messaging]
    # If a message send fails, retry after this delay:
    retry interval in seconds = 5
    # If send fails after this many tries, give up trying:
    maximum number of tries = 7

    # This timeout is the same as --comms-timeout for user commands. If
    # set to None (no timeout) messages to non-responsive suites
    # (e.g. suspended with Ctrl-Z) could hang indefinitely.
    connection timeout in seconds = 30

# Setup the communication method details. This is required for
# communications between cylc clients and servers (i.e. between
# suite-connecting commands and guis, and running suite server processes).
[communication]
```

```
    # Configure the choice of communication method. Only https is supported
    # at the moment.
# SITE ONLY
    method = https

    # Each suite listens on a dedicated network port. The port to bind to is
    # selected randomly from the allowed range of ports.
# SITE ONLY
    base port = 43001

    # This sets the maximum number of suites that can run at once.
# SITE ONLY
    maximum number of ports = 100

[hosts]
    # The default task host is the suite host, i.e. localhost:
    # Add task host sections if local defaults are not sufficient.
    [[HOST]]
        # Method of communication of task progress back to the suite:
        #    1) default - HTTPS via network ports
        #    2) ssh  - re-invoke messaging commands on suite server
        #    3) poll - the suite polls for status of passive tasks
        # HTTPS comms are still required in all cases *on the suite host*
        # for cylc clients (commands etc.) to communicate with suites.
        task communication method = "default" # or "ssh" or "poll"
        # The "poll" method sets a default interval here to ensure no
        # tasks are accidentally left unpolled. You can override this
        # with run-length appropriate intervals under task [runtime]
        # (however this will also result in routine polling under the
        # default or ssh communications).
        default polling interval in minutes = 1.0
```

12.5 How Commands Interact With Running Suites

User-invoked commands that connect to running suites can also choose between direct communication across network sockets (HTTPS) and re-invocation of commands on the suite host using non-interactive ssh (there is a `--use-ssh` command option for this purpose).

The gcylc GUI requires direct HTTPS connections to its target suite. If that is not possible, run gcylc on the suite host.

12.6 Client Authentication and Passphrases

Suite daemons listen on dedicated network ports for incoming client requests - task messages and user-invoked commands (CLI or GUI). The `cylc scan` command reveals which suites are running on scanned hosts, and what ports they are listening on.

Client programs have to authenticate with the target suite daemon before issuing commands or requesting information. Cylc has two authentication levels: full control via a suite-specific passphrase (see 12.6.1); and configurable free "public" access (see 12.6.2).

12.6.1 Full Control - Suite Passphrases

A file called `passphrase` with owner-only permissions is generated under `.server/` in the suite run directory at registration time. It is loaded by the suite daemon at start-up and used to authenticate connections from client programs. Suite passphrases are used in an encrypted challenge-response scheme; they are never sent raw over the network.

On submission of the first job on another task host cylc will attempt to install the passphrase to the run directory there to enable task jobs to connect to the suite, using non-interactive ssh.

Client programs on other accounts will attempt to read the passphrase via non-interactive SSH and install it to `$HOME/.cylc/auth/OWNER@HOST/SUITE/passphrase`. Alternatively, if the suite owner gives you the passphrase you can install it yourself to the same location.

12.6.2   Public Access - No Passphrase

Possession of a suite passphrase gives full read and control access to the suite. Without the passphrase the amount of information revealed by a suite daemon is determined by the public access privilege level set in global site/user config (B.15) and optionally overidden in suites (A.2.16):

- *identity* - only suite and owner names revealed
- *description* - identity plus suite title and description
- *state-totals* - identity, description, and task state totals
- *full-read* - full read-only access for monitor and GUI
- *shutdown* - full read access plus shutdown, but no other control.

The default public access level is *state-totals*. The `cylc scan` command can print descriptions and task state totals in addition to basic suite identity, if you have the right passphrases or if the requested information is revealed publicly.

12.7   How Tasks Get Access To Cylc

Running tasks need access to cylc via `$PATH`, principally for the task messaging commands. To allow this, the first thing a task job script does is set `$CYLC_VERSION` to the cylc version number of the running suite. If you need to run several suites at once under different incompatible versions of cylc, check that your site is using the cylc version wrapper (see `INSTALL` and `admin/cylc-wrapper` in a cylc installation) then set `$CYLC_VERSION` to the desired version. In the case of developers wishing to run their own copy of cylc rather than a centrally installed one, set `$CYLC_HOME` to point to your cylc copy.

Access to the cylc executable for different hosts can be configured using the site and user global configuration files. If the environment for running the cylc executable is only set up correctly in a login shell for a given host, you can set `[hosts][HOST]use login shell = True` for the relevant host (this is the default behaviour). If the environment is already correct without the login shell, but the cylc executable is not in `$PATH`, then `[hosts][HOST]cylc executable` can be used to specify the path to the cylc executable.

To customize the environment more generally for cylc on jobs hosts, use of `job-init-env.sh` is described in 7.1.1.

12.8   Restarting Suites

A restarted suite (see `cylc restart --help`) is initialized from a previous recorded checkpoint, which is normally the end of a previous run, so that it can carry on from wherever it got to before being shut down or killed.

12.8.1   Restart From Latest

A normal restart is easy. Simply invoke the `cylc restart` command with the suite name:
```
$ cylc restart SUITE
```

It will restart the suite from the latest checkpoint.

12.8.2   Restart From Checkpoint

You can use the `cylc ls-checkpoints` command to identify the checkpoint to use to restart a suite. (See also `cylc ls-checkpoints --help`.)

The checkpoint ID 0 (zero) is always used for latest state of the suite, which is updated continuously as the suite progresses. The checkpoint IDs of earlier states are positive integers starting from 1, and incremented each time a new checkpoint is stored. Currently suites automatically store checkpoints before and after reloads, and on restarts (using the latest checkpoints before the restarts).

Once you have identified the checkpoint to restart from, invoke the `cylc restart` command with the suite name and the `--checkpoint=CHECKPOINT` option:
```
$ cylc restart --checkpoint=CHECKPOINT-ID SUITE
```

12.8.3   Manual Checkpoints

You can use the `cylc checkpoint` command to tell a running suite to record a checkpoint at any time:
```
$ cylc checkpoint SUITE CHECKPOINT-NAME
```

The 2nd argument is a name you give to the checkpoint so you can easily identify it when you need to use it.

(See also `cylc checkpoint --help`.)

12.8.4   Behaviour of Tasks on Restart

All tasks are reloaded in exactly their former states as recorded in checkpoint. Failed tasks, for instance, are not automatically resubmitted at restart in case the underlying problem has not been addressed yet.

Tasks recorded in the submitted or running states are automatically polled on restart, to see if they are still waiting in a batch queue, still running, or if they succeeded or failed while the suite was down.

12.9   Task States

As a suite runs its task proxies may pass through the following states:

- **waiting** - prerequisites not satisfied yet (note that clock-trigger tasks also wait on their trigger time).

- **queued** - ready to run (prerequisites satisfied) but temporarily held back by an *internal cylc queue* (see 12.13).
- **held** - will not be submitted even if ready to run.
- **ready** - ready to run (prerequisites satisfied) and handed to cylc's job submission subsystem.
- **submitted** - submitted to run, but not executing yet (could be waiting in an external batch scheduler queue).
- **submit-failed** - job submission failed *or* submitted job killed (cancelled) before commencing execution.
- **submit-retrying** - job submission failed, but a submission retry was configured. Will only enter the *submit-failed* state if all configured submission retries are exhausted.
- **running** - currently executing (a *task started* message was received, or the task polled as running).
- **succeeded** - finished executing successfully (a *task succeeded* message was received, or the task polled as succeeded).
- **failed** - aborted execution due to some error condition (a *task failed* message was received, or the task polled as failed).
- **retrying** - job execution failed, but an execution retry was configured. Will only enter the *failed* state if all configured execution retries are exhausted.

Note that greyed-out "base graph nodes" in the gcylc graph view do not represent task states; they are displayed to fill out the graph structure where corresponding task proxies do not currently exist in the live task pool.

For manual task state reset purposes **ready** is a pseudo-state that means *waiting* with all prerequisites satisfied.

12.10   Remote Control - Passphrases and Network Ports

Connecting to a running suite requires knowing the *network port* it is listening on, and the *suite passphrase* to authenticate with once a connection is made to the port.

A suite writes its contact information to `$HOME/cylc-run/<SUITE>/.service/contact` at start-up. Client commands that connect to the suite read this file to obtain the necessary information on how to connect to the suite.

To connect to a suite running on another account you must install the suite passphrase (12.6.1), and configure non-interactive ssh so that the contact information can be retrieved from the remote contact file. Then use the `--user` and `--host` command options to connect:

```
$ cylc monitor --user=USER --host=HOST SUITE
```

Alternatively, you can determine suite contact information using `cylc scan`, and use them explicitly on the command line:

```
$ cylc monitor --user=USER --host=HOST --port=PORT SUITE
```

Possession of a suite passphrase gives full control over the suite, and ssh access to the contact file also implies full access to the suite host account, so it is recommended that this only be used to interact with suites running on accounts that you already have full access. (See also 12.6.2.)

12.11   Network Connection Timeouts

A connection timeout can be set in site and user global config files (see 6) so that messaging commands cannot hang indefinitely if the suite is not responding (this can be caused by suspending a suite with Ctrl-Z) thereby preventing the task from completing. The same can be done on the command line for other suite-connecting user commands, with the `--comms-timeout` option.

12.12   Runahead Limiting

Runahead limiting prevents the fastest tasks in a suite from getting too far ahead of the slowest ones. Newly spawned tasks are released to the task pool only when they fall below the runahead limit. A low runhead limit can prevent cylc from interleaving cycles, but it will not stall a suite unless it fails to extend out past a future trigger (see 9.3.5.11). A high runahead limit may allow fast tasks that are not constrained by dependencies or clock-triggers to spawn far ahead of the pack, which could have performance implications for the suite daemon when running very large suites. Succeeded and failed tasks are ignored when computing the runahead limit.

The preferred runahead limiting mechanism restricts the number of consecutive active cycle points. The default value is three active cycle points; see A.3.8. Alternatively the interval between the slowest and fastest tasks can be specified as hard limit; see A.3.7.

12.13   Limiting Active Tasks With Internal Queues

Large suites can potentially overwhelm task hosts by submitting too many tasks at once. You can prevent this with *internal queues*, which limit the number of tasks that can be active (submitted or running) at the some time.

A queue is defined by a *name*; a *limit*, which is the maximum number of active tasks allowed for the queue; and a list of *members*, assigned by task or family name.

Queue configuration is done under the [scheduling] section of the suite.rc file (like dependencies, internal queues constrain *when* a task runs).

By default every task is assigned to the *default* queue, which by default has a zero limit (interpreted by cylc as no limit). To use a single queue for the whole suite just set the default queue limit:

```
[scheduling]
    [[ queues]]
        # limit the entire suite to 5 active tasks at once
        [[[default]]]
            limit = 5
```

To use additional queues just name each one, set their limits, and assign members:

```
[scheduling]
    [[ queues]]
        [[[q_foo]]]
            limit = 5
            members = foo, bar, baz
```

Any tasks not assigned to a particular queue will remain in the default queue. The *queues* example suite illustrates how queues work by running two task trees side by side (as seen in the graph GUI) each limited to 2 and 3 tasks respectively:

```
title = demonstrates internal queueing
description = """
Two trees of tasks: the first uses the default queue set to a limit of
two active tasks at once; the second uses another queue limited to three
active tasks at once. Run via the graph control GUI for a clear view.
              """
[scheduling]
    [[queues]]
        [[[default]]]
            limit = 2
        [[[foo]]]
            limit = 3
            members = n, o, p, FAM2, u, v, w, x, y, z
    [[dependencies]]
        graph = """
            a => b & c => FAM1
            n => o & p => FAM2
            FAM1:succeed-all => h & i & j & k & l & m
            FAM2:succeed-all => u & v & w & x & y & z
              """
[runtime]
    [[FAM1, FAM2]]
    [[d,e,f,g]]
        inherit = FAM1
    [[q,r,s,t]]
        inherit = FAM2
```

12.14   Automatic Task Retry On Failure

See also A.4.1.13.6 in the *Suite.rc Reference.*

Tasks can be configured with a list of "retry delay" periods, as ISO 8601 durations, such that
if a task fails it will go into a temporary *retrying* state and then automatically resubmit itself
after the next specified delay period expires. A usage example is shown in the suite listed below
under 12.15.

12.15   Suite And Task Event Handling

See also A.2.13 and A.4.1.15 in the *Suite.rc Reference.*

Cylc can call nominated event handlers when certain suite or task events occur. This is intended
to facilitate centralized alerting and automated handling of critical events. Event handlers can
be used to send a message, call a pager, and so on; or intervene in the operation of their own
suite using cylc commands.

To send an email, you can use the built-in setting `[[[events]]]mail events` to specify a list of
events for which notifications should be sent. E.g. to send an email on (submission) failed and
retry:

```
[runtime]
    [[foo]]
        retry delays = PT0S, PT30S
        script = "test ${CYLC_TASK_TRY_NUMBER} -eq 3"
        [[[events]]]
            mail events = submission failed, submission retry, failed, retry
```

By default, the emails will be sent to the current user with:

- `to:` set as `$USER`
- `from:` set as `notifications@$(hostname)`

- SMTP server at `localhost:25`

These can be configured using the settings:

- `[[[events]]]mail to` (list of email addresses),
- `[[[events]]]mail from`
- `[[[events]]]mail smtp`.

By default, a cylc suite will send you no more than one task event email every 5 minutes - this is to prevent your inbox from being flooded by emails should a large group of tasks all fail at similar time. See A.2.8 for details.

(It is worth noting that while mail events are accumulating the succeeded task proxies do not get cleaned up - which would be bad in a large, active suite with a long mail event interval. If you do configure mail notifications for succeeded tasks, you should ensure that you don't have an excessively long task mail event interval.)

Event handler commands can be located in the suite `bin/` directory, otherwise it is up to you to ensure their location is in `$PATH` (in the shell in which cylc runs, on the suite host). The commands should require very little resource to run and should return quickly. (Each event handler is invoked by a child process in a finite process pool that is also used to submit, poll and kill jobs. The child process will wait for the event handler to complete before moving on to the next item in the queue. If the process pool is saturated with long running event handlers, the suite will appear to hang.)

Task event handlers can be specified using the `[[[events]]]<event> handler` settings, where `<event>` is one of:

- 'submitted' - the job submit command was successful
- 'submission failed' - the job submit command failed
- 'submission timeout' - task job submission timed out
- 'submission retry' - task job submission failed, but will retry after a configured delay
- 'started' - the task reported commencement of execution
- 'succeeded' - the task reported successful completion
- 'warning' - the task reported a warning message
- 'failed' - the task failed
- 'retry' - the task failed but will retry
- 'execution timeout' - task execution timed out

The value of each setting should be a list of command lines or command line templates (see below).

Alternatively, task event handlers can be specified using the `[[[events]]]handlers` and the `[[[events]]]handler even` settings, where the former is a list of command lines or command line templates (see below) and the latter is a list of events for which these commands should be invoked.

A command line template may have any or all of these patterns which will be substituted with actual values:

- %(event)s: event name
- %(suite)s: suite name
- %(point)s: cycle point
- %(name)s: task name

- %(submit_num)s: submit number
- %(id)s: task ID (i.e. %(name)s.%(point)s)
- %(message)s: event message, if any

Otherwise, the command line will be called with the following command line arguments:

```
<task-event-handler> %(event)s %(suite)s %(id)s %(message)s
```

For an explanation of the substitution syntax, see String Formatting Operations in the Python documentation.

The retry event occurs if a task fails and has any remaining retries configured (see 12.14). The event handler will be called as soon as the task fails, not after the retry delay period when it is resubmitted.

Note that event handlers are called by cylc itself, not by the running tasks so if you wish to pass them additional information via the environment you must use [cylc] →[[environment]], not task runtime environments.

The following 2 `suite.rc` snippets are examples on how to specify event handlers using the alternate methods:

```
[runtime]
    [[foo]]
        retry delays = PT0S, PT30S
        script = "test ${CYLC_TASK_TRY_NUMBER} -eq 2"
        [[[events]]]
            retry handler = "echo '!!!!!EVENT!!!!!' "
            failed handler = "echo '!!!!!EVENT!!!!!' "

[runtime]
    [[foo]]
        retry delays = PT0S, PT30S
        script = "test ${CYLC_TASK_TRY_NUMBER} -eq 2"
        [[[events]]]
            handlers = "echo '!!!!!EVENT!!!!!' "
            handler events = retry, failed
```

Note: The handler command is called like this:

```
echo '!!!!!EVENT!!!!!' %(event)s %(suite)s %(id)s %(message)s
```

12.16   Reloading The Suite Definition At Runtime

The `cylc reload` command reloads the suite definition at run time. This allows: (a) changing task config items such as script or environment; (b) adding tasks to, or removing them from, the suite definition, at run time - without shutting the suite down and restarting it. (It is easy to shut down and restart cylc suites, but reloading may be useful if you don't want to wait for long-running tasks to finish first).

Note that *defined tasks* can be already be added to or removed from a running suite with the `cylc insert` and `cylc remove` commands; the reload command allows addition and removal of *task definitions*. If a new task is definition is added (and used in the graph) you will still need to manually insert an instance of it (with a particular cycle point) into the running suite. If a task definition (and its use in the graph) is deleted, existing task proxies of the of the deleted type will run their course after the reload but new instances will not be spawned. Changes to a task definition will only take effect when the next task instance is spawned (existing instances will not be affected).

12.17 Handling Job Preemption

Some HPC facilities allow job preemption: the resource manager can kill or suspend running low priority jobs in order to make way for high priority jobs. The preempted jobs may then be automatically restarted by the resource manager, from the same point (if suspended) or requeued to run again from the start (if killed). If a running cylc task gets suspended or hard-killed (`kill -9 <PID>` is not a trappable signal so cylc cannot detect task failure in this case) and then later restarted, it will just appear to cylc as if it takes longer than normal to run. If the job is soft-killed the signal will be trapped by the task job script and a failure message sent, resulting in cylc putting the task into the failed state. When the preempted task restarts and sends its started message cylc would normally treat this as an error condition (a dead task is not supposed to be sending messages) - a warning will be logged and the task will remain in the failed state. However, if you know that preemption is possible on your system you can tell cylc that affected tasks should be resurrected from the dead, to carry on as normal if progress messages start coming in again after a failure:

```
# ...
[runtime]
    [[HPC]]
        enable resurrection = True
    [[TaskFoo]]
        inherit = HPC
# ...
```

To test this in any suite, manually kill a running task then, after cylc registers the task failed, resubmit the killed job manually by cutting-and-pasting the original job submission command from the suite stdout stream.

12.18 Manual Task Triggering and Edit-Run

Any task proxy currently present in the suite can be manually triggered at any time using the `cylc trigger` command, or from the right-click task menu in gcylc. If the task belongs to a limited internal queue (see 12.13), this will queue it; if not, or if it is already queued, it will submit immediately.

With `cylc trigger --edit` (also in the gcylc right-click task menu) you can edit the generated task job script to make one-off changes before the task submits.

12.19 Cylc Broadcast

The `cylc broadcast` command overrides `[runtime]` settings in a running suite. This can be used to communicate information to downstream tasks by broadcasting environment variables (communication of information from one task to another normally takes place via the filesystem, i.e. the input/output file relationships embodied in inter-task dependencies). Variables (and any other runtime settings) may be broadcast to all subsequent tasks, or targeted specifically at a specific task, all subsequent tasks with a given name, or all tasks with a given cycle point; see broadcast command help for details.

Broadcast settings targeted at a specific task ID or cycle point expire and are forgotten as the suite moves on. Un-targeted variables and those targeted at a task name persist throughout the suite run, even across restarts, unless manually cleared using the broadcast command - and so should be used sparingly.

12.20   The Meaning And Use Of Initial Cycle Point

When a suite is started with the `cylc run` command (cold or warm start) the cycle point at which it starts can be given on the command line or hardwired into the suite.rc file:

`cylc run foo 20120808T06Z`

or:

```
[scheduling]
    initial cycle point = 20100808T06Z
```

An initial cycle given on the command line will override one in the suite.rc file.

12.20.1   The Environment Variable CYLC_SUITE_INITIAL_CYCLE_POINT

In the case of a *cold start only* the initial cycle point is passed through to task execution environments as `$CYLC_SUITE_INITIAL_CYCLE_POINT`. The value is then stored in suite database files and persists across restarts, but it does get wiped out (set to `None`) after a warm start, because a warm start is really an implicit restart in which all state information is lost (except that the previous cycle is assumed to have completed).

The `$CYLC_SUITE_INITIAL_CYCLE_POINT` variable allows tasks to determine if they are running in the initial cold-start cycle point, when different behaviour may be required, or in a normal mid-run cycle point. Note however that an initial `R1` graph section is now the preferred way to get different behaviour at suite start-up.

12.21   Simulating Suite Behaviour

Several suite run modes allow you to simulate suite behaviour quickly without running the suite's real jobs - which may be long-running and resource-hungry:

- *dummy mode* - runs dummy tasks as background jobs on configured job hosts.
  - simulates scheduling, job host connectivity, and generates all job files on suite and job hosts.
- *dummy-local mode* - runs real dummy tasks as background jobs on the suite host, which allows dummy-running suites from other sites.
  - simulates scheduling and generates all job files on the suite host.
- *simulation mode* - does not run any real tasks.
  - simulates scheduling without generating any job files.

Set the run mode (default *live*) in the GUI suite start dialog box, or on the command line:

```
$ cylc run --mode=dummy SUITE
$ cylc restart --mode=dummy SUITE
```

You can get specified tasks to fail in these modes, for more flexible suite testing. See Section A.4.1.21 for simulation configuration.

12.21.1   Proportional Simulated Run Length

If task `[job]execution time limit` is set, Cylc divides it by `[simulation]speedup factor` (default `10.0`) to compute simulated task run lengths (default 10 seconds).

118

12.21.2   Limitations Of Suite Simulation

Dummy mode ignores batch scheduler settings because Cylc does not know which job resource directives (requested memory, number of compute nodes, etc.) would need to be changed for the dummy jobs. If you need to dummy-run jobs on a batch scheduler manually comment out `script` items and modify directives in your live suite, or else use a custom live mode test suite.

Note that the dummy modes ignore all configured task `script` items including `init-script`. If your `init-script` is required to run even dummy tasks on a job host, note that host environment setup should be done elsewhere - see 3.3.3.

12.21.3   Restarting Suites With A Different Run Mode?

The run mode is recorded in the suite run database files. Cylc will not let you *restart* a non-live mode suite in live mode, or vice versa. To test a live suite in simulation mode just take a quick copy of it and run the the copy in simulation mode.

12.22   Automated Reference Test Suites

Reference tests are finite-duration suite runs that abort with non-zero exit status if any of the following conditions occur (by default):

- cylc fails
- any task fails
- the suite times out (e.g. a task dies without reporting failure)
- a nominated shutdown event handler exits with error status

The default shutdown event handler for reference tests is `cylc hook check-triggering` which compares task triggering information (what triggers off what at run time) in the test run suite log to that from an earlier reference run, disregarding the timing and order of events - which can vary according to the external queueing conditions, runahead limit, and so on.

To prepare a reference log for a suite, run it with the `--reference-log` option, and manually verify the correctness of the reference run.

To reference test a suite, just run it (in dummy mode for the most comprehensive test without running real tasks) with the `--reference-test` option.

A battery of automated reference tests is used to test cylc before posting a new release version. Reference tests can also be used to check that a cylc upgrade will not break your own complex suites - the triggering check will catch any bug that causes a task to run when it shouldn't, for instance; even in a dummy mode reference test the full task job script (sans `script` items) executes on the proper task host by the proper batch system.

Reference tests can be configured with the following settings:

```
[cylc]
    [[reference test]]
        suite shutdown event handler = cylc check-triggering
        required run mode = dummy
        allow task failures = False
        live mode suite timeout = PT5M
        dummy mode suite timeout = PT2M
        simulation mode suite timeout = PT2M
```

12.22.1   Roll-your-own Reference Tests

If the default reference test is not sufficient for your needs, firstly note that you can override the default shutdown event handler, and secondly that the `--reference-test` option is merely a short cut to the following suite.rc settings which can also be set manually if you wish:

```
[cylc]
    abort if any task fails = True
    [[events]]
        shutdown handler = cylc check-triggering
        timeout = PT5M
        abort if shutdown handler fails = True
        abort on timeout = True
```

12.23   Triggering Off Of Tasks In Other Suites

The `cylc suite-state` command interrogates suite run databases. It has a polling mode that waits for a given task in the target suite to achieve a given state. This can be used to make task scripting wait for a remote task to succeed (for example). The suite graph notation also provides a way to define automatic suite-state polling tasks, which use the same polling command under the hood. Note that cylc suite-state can only trigger off task *states* in remote suites and does not support triggering off task messages.

Here's how to trigger a task `bar` off a task `foo` in a remote suite called `other.suite`:

```
[scheduling]
    [[dependencies]]
        [[[T00, T12]]]
            graph = "my-foo<other.suite::foo> => bar"
```

Local task `my-foo` will poll for the success of `foo` in suite `other.suite`, at the same cycle point, succeeding only when or if it succeeds. Other task states can also be polled:

```
    graph = "my-foo<other.suite::foo:fail> => bar"
```

The default polling parameters (e.g. maximum number of polls and the interval between them) are printed by `cylc suite-state --help` and can be configured if necessary under the local polling task runtime section:

```
[scheduling]
    [[ dependencies]]
        [[[T00,T12]]]
            graph = "my-foo<other.suite::foo> => bar"
[runtime]
    [[my-foo]]
        [[[suite state polling]]]
            max-polls = 100
            interval = PT10S
```

For suites owned by others, or those with run databases in non-standard locations, use the `--run-dir` option, or in-suite:

```
[runtime]
    [[my-foo]]
        [[[suite state polling]]]
            run-dir = /path/to/top/level/cylc/run-directory
```

If the remote task has a different cycling sequence, just arrange for the local polling task to be on the same sequence as the remote task that it represents. For instance, if local task `cat` cycles 6-hourly at `0,6,12,18` but needs to trigger off a remote task `dog` at `3,9,15,21`:

```
[scheduling]
    [[dependencies]]
        [[[T03,T09,T15,T21]]]
            graph = "my-dog<other.suite::dog>"
        [[[T00,T06,T12,T18]]]
            graph = "my-dog[-PT3H] => cat"
```

For suite-state polling the cycle point of the target task is treated as a literal string so the polling command has to be told if the remote suite has a different cycle point format. Use the `--template` option for this, or in-suite:

```
[runtime]
    [[my-foo]]
        [[[suite state polling]]]
            template = %Y-%m-%dT%H
```

Note that the remote suite does not have to be running when polling commences because the command interrogates the suite run database, not the suite server process.

12.24   Suite Server Logs

Each suite maintains its own log of time-stamped events under the *suite server log directory*:

```
$HOME/cylc-run/<SUITE-NAME>/log/suite/
```

By way of example, we will show the complete server log generated (at cylc-7.2.0) by a small suite that runs two 30-second dummy tasks `foo` and `bar` for a single cycle point `2017-01-01T00Z` before shutting down:

```
[cylc]
    cycle point format = %Y-%m-%dT%HZ
[scheduling]
    initial cycle point = 2017-01-01T00Z
    final cycle point = 2017-01-01T00Z
    [[dependencies]]
        graph = "foo => bar"
[runtime]
    [[foo]]
        script = sleep 30; /bin/false
    [[bar]]
        script = sleep 30; /bin/true
```

By the task scripting defined above, this suite will stall when `foo` fails. Then, the suite owner *vagrant@cylon* manually resets the failed task's state to *succeeded*, allowing `bar` to trigger and the suite to finish and shut down. Here's the complete suite log for this run:

```
$ cylc log SUITE-NAME
2017-03-30T09:46:10Z INFO - Suite starting: server=localhost:43086 pid=3483
2017-03-30T09:46:10Z INFO - Run mode: live
2017-03-30T09:46:10Z INFO - Initial point: 2017-01-01T00Z
2017-03-30T09:46:10Z INFO - Final point: 2017-01-01T00Z
2017-03-30T09:46:10Z INFO - Cold Start 2017-01-01T00Z
2017-03-30T09:46:11Z INFO - [foo.2017-01-01T00Z] -submit_method_id=3507
2017-03-30T09:46:11Z INFO - [foo.2017-01-01T00Z] -submission succeeded
2017-03-30T09:46:11Z INFO - [foo.2017-01-01T00Z] -(current:submitted)> started
    at 2017-03-30T09:46:10Z
2017-03-30T09:46:41Z CRITICAL - [foo.2017-01-01T00Z] -(current:running)> Task
    job script received signal EXIT at 2017-03-30T09:46:40Z
2017-03-30T09:46:41Z CRITICAL - [foo.2017-01-01T00Z] -(current:running)> failed
    at 2017-03-30T09:46:40Z
2017-03-30T09:46:42Z WARNING - suite stalled
2017-03-30T09:46:42Z WARNING - Unmet prerequisites for bar.2017-01-01T00Z:
2017-03-30T09:46:42Z WARNING -  * foo.2017-01-01T00Z succeeded
2017-03-30T09:47:58Z INFO - [client-command] reset_task_states vagrant@cylon:
    cylc-reset 1e0d8e9f-2833-4dc9-a0c8-9cf263c4c8c3
```

```
2017-03-30T09:47:58Z INFO - [foo.2017-01-01T00Z] -resetting state to succeeded
2017-03-30T09:47:58Z INFO - Command succeeded: reset_task_states([u'foo.2017'],
    state=succeeded)
2017-03-30T09:47:59Z INFO - [bar.2017-01-01T00Z] -submit_method_id=3565
2017-03-30T09:47:59Z INFO - [bar.2017-01-01T00Z] -submission succeeded
2017-03-30T09:47:59Z INFO - [bar.2017-01-01T00Z] -(current:submitted)> started
    at 2017-03-30T09:47:58Z
2017-03-30T09:48:29Z INFO - [bar.2017-01-01T00Z] -(current:running)> succeeded
    at 2017-03-30T09:48:28Z
2017-03-30T09:48:30Z INFO - Waiting for the command process pool to empty for
    shutdown
2017-03-30T09:48:30Z INFO - Suite shutting down - AUTOMATIC
```

The information logged here includes:

- event timestamps, at the start of each line
- suite server host, port and process ID
- suite initial and final cycle points
- suite start type (cold start in this case)
- task events (task started, succeeded, failed, etc.)
- suite stalled warning (in this suite nothing else can run when `foo` fails)
- the client command issued by *vagrant@cylon* to reset `foo` to *succeeded*
- job IDs - in this case process IDs for background jobs (or PBS job IDs etc.)
- state changes due to incoming task progress message ("started at ..." etc.) suite shutdown time and reasons (AUTOMATIC means "all tasks finished and nothing else to do")

Note that suite log files are primarily intended for human eyes. If you need to have an external system to monitor suite events automatically, interrogate the sqlite *suite run database* (see 12.25) rather than parse the log files.

12.25   Suite Run Databases

Suites maintain `sqlite` databases to record run history and restart checkpoints etc.:

```
$HOME/cylc-run/<SUITE-NAME>/log/db
```

You can use any standard sqlite interface to read these databases, such as the Python standard library `sqlite3` module, and a command line tool of the same name:

```
$ sqlite3 ~/cylc-run/foo/log/db << _END_
> .headers on
> select * from task_events where name is "foo";
> _END_
name|cycle|time|submit_num|event|message
foo|1|2017-03-12T11:06:08Z|1|incrementing submit number|
foo|1|2017-03-12T11:06:09Z|1|submission succeeded|
foo|1|2017-03-12T11:06:09Z|1|output completed|started
foo|1|2017-03-12T11:06:09Z|1|started|
foo|1|2017-03-12T11:06:19Z|1|output completed|succeeded
foo|1|2017-03-12T11:06:19Z|1|succeeded|
```

13   Suite Storage, Discovery, Revision Control, and Deployment

Small groups of cylc users can of course share suites by manual copying, and generic revision control tools can be used on cylc suites as for any collection of files. Beyond this cylc does not have a built-in solution for suite storage and discovery, revision control, and deployment, on a

122

network. That is not cylc's core purpose, and large sites may have preferred revision control systems and suite meta-data requirements that are difficult to anticipate. We can, however, recommend the use of *Rose* to do all of this very easily and elegantly with cylc suites.

13.1   Rose

Rose is *a framework for managing and running suites of scientific applications*, developed at the UK Met Office for use with cylc. It is available under the open source GPL license.

- Rose documentation: `http://metomi.github.io/rose/doc/rose.html`
- Rose source repository: `https://github.com/metomi/rose`

A Suite.rc Reference

This appendix defines all legal suite definition config items. Embedded Jinja2 code (see 9.7) must process to a valid raw suite.rc file. See also 9.2 for a descriptive overview of suite.rc files, including syntax (9.2.1).

A.1 Top Level Items

The only top level configuration items at present are the suite title and description.

A.1.1 title

A single line description of the suite. It is displayed in the db viewer window and can be retrieved at run time with the `cylc show` command.

- *type:* single line string
- *default:* (none)

A.1.2 description

A multi-line description of the suite. It can be retrieved by the db viewer right-click menu, or at run time with the `cylc show` command.

- *type:* multi-line string
- *default:* (none)

A.1.3 group

A single line group name for a suite. It is displayed in the gscan window and can be used as a way of grouping related suites together. `cylc show` command.

- *type:* single line string
- *default:* (none)

A.1.4 URL

A web URL to suite documentation. If present it can be browsed with the `cylc doc` command, or from the gcylc Suite menu. The variable `${CYLC_SUITE_NAME}` will be replaced with the actual suite name (note this is not a shell environment variable in this context). See also task URLs (A.4.1.4).

- *type:* string (URL)
- *default:* (none)
- *example:* `http://suites/${CYLC_SUITE_NAME}/index.html`

A.2 [cylc]

This section is for configuration that is not specifically task-related.

A.2.1 [cylc] →required run mode

If this item is set cylc will abort if the suite is not started in the specified mode. This can be used for demo suites that have to be run in simulation mode, for example, because they have been taken out of their normal operational context; or to prevent accidental submission of expensive real tasks during suite development.

- *type:* string
- *legal values:* live, dummy, dummy-local, simulation
- *default:* None

A.2.2 [cylc] →UTC mode

Cylc runs off the suite host's system clock by default. This item allows you to run the suite in UTC even if the system clock is set to local time. Clock-trigger tasks will trigger when the current UTC time is equal to their cycle point date-time plus offset; other time values used, reported, or logged by the suite daemon will usually also be in UTC. The default for this can be set at the site level (see B.14.1).

- *type:* boolean
- *default:* False, unless overridden at site level.

A.2.3 [cylc] →cycle point format

To just alter the timezone used in the date/time cycle point format, see A.2.5. To just alter the number of expanded year digits (for years below 0 or above 9999), see A.2.4.

Cylc usually uses a `CCYYMMDDThhmmZ` (`Z` in the special case of UTC) or `CCYYMMDDThhmm+hhmm` format (`+` standing for `+` or `-` here) for writing down date/time cycle points, which follows one of the basic formats outlined in the ISO 8601 standard. For example, a cycle point on the 3rd of February 2001 at 4:50 in the morning, UTC (+0000 timezone), would be written `20010203T0450Z`. Similarly, for the the 3rd of February 2001 at 4:50 in the morning, +1300 timezone, cylc would write `20010203T0450+1300`.

You may use the isodatetime library's syntax to write dates and times in ISO 8601 formats - `CC` for century, `YY` for decade and decadal year, `+X` for expanded year digits and their positive or negative sign, thereafter following the ISO 8601 standard example notation except for fractional digits, which are represented as `,ii` for `hh`, `,nn` for `mm`, etc. For example, to write date/times as week dates with fractional hours, set cycle point format to `CCYYWwwDThh,iiZ` e.g. `1987W041T08,5Z` for 08:30 UTC on Monday on the fourth ISO week of 1987.

You can also use a subset of the strptime/strftime POSIX standard - supported tokens are `\%F`, `\%H`, `\%M`, `\%S`, `\%Y`, `\%d`, `\%j`, `\%m`, `\%s`, `\%z`.

To use the old cylc date-time format (e.g. `2014020106` for 06:00 on the 1st of February 2014), set cycle point format to `\%Y\%m\%d\%H`.

Please note that using characters like "/" is not allowed, as it will break task output files ("/" is a reserved character in POSIX file and directory naming). Using ":" is also not allowed, as it is likely to interfere with usage of commands like "rsync" when applied to task output files.

A.2.4   [cylc] →cycle point num expanded year digits

For years below 0 or above 9999, the ISO 8601 standard specifies that an extra number of year digits and a sign should be used. This extra number needs to be written down somewhere (here).

For example, if this extra number is set to 2, 00Z on the 1st of January in the year 10040 will be represented as `+0100400101T0000Z` (2 extra year digits used). With this number set to 3, 06Z on the 4th of May 1985 would be written as `+00019850504T0600Z`.

This number defaults to 0 (no sign or extra digits used).

A.2.5   [cylc] →cycle point time zone

If you set UTC mode to True (A.2.2) then this will default to `Z`. If you use a custom cycle point format (A.2.3), you should specify the timezone choice (or null timezone choice) here as well.

You may set your own time zone choice here, which will be used for all date/time cycle point dumping. Time zones should be expressed as ISO 8601 time zone offsets from UTC, such as `+13`, `+1300`, `-0500` or `+0645`, with `Z` representing the special `+0000` case. Cycle points will be converted to the time zone you give and will be represented with this string at the end.

Cycle points that are input without time zones (e.g. as an initial cycle point setting) will use this time zone if set. If this isn't set (and UTC mode is also not set), then they will default to the current local time zone.

Note that the ISO standard also allows writing the hour and minute separated by a ":" (e.g. `+13:00`) - however, this is not recommended, given that the time zone is used as part of task output filenames.

A.2.6   [cylc] →abort if any task fails

Cylc does not normally abort if tasks fail, but if this item is turned on it will abort with exit status 1 if any task fails.

- *type:* boolean
- *default:* False

A.2.7   [cylc] →health check interval

Specify the time interval on which a running cylc suite will check that its run directory exists and that its contact file contains the expected information. If not, the suite will shut itself down automatically.

- *type:* ISO 8601 duration/interval representation (e.g. `PT5M`, 5 minutes (note: by contrast, `P5M` means 5 months, so remember the `T`!)).

- *default:* PT10M

A.2.8   [cylc] →task event mail interval

Group together all the task event mail notifications into a single email within a given interval. This is useful to prevent flooding users' mail boxes when many task events occur within a short period of time.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default: PT5M*

A.2.9   [cylc] →disable automatic shutdown

This has the same effect as the `--no-auto-shutdown` flag for the suite run commands: it prevents the suite daemon from shutting down normally when all tasks have finished (a suite timeout can still be used to stop the daemon after a period of inactivity, however). This option can make it easier to re-trigger tasks manually near the end of a suite run, during suite development and debugging.

- *type:* boolean
- *default:* False

A.2.10   [cylc] →log resolved dependencies

If this is turned on cylc will write the resolved dependencies of each task to the suite log as it becomes ready to run (a list of the IDs of the tasks that actually satisfied its prerequisites at run time). Mainly used for cylc testing and development.

- *type:* boolean
- *default:* False

A.2.11   [cylc] →parameters

Define parameter values here for use in expanding *parameterized tasks* - see Section 9.6.

- *type:* list of strings, or an integer range `LOWER..UPPER` (two dots, inclusive bounds)
- *default:* (none)
- *examples:*
  - `run = control, test1, test2`
  - `mem = 1..5` (equivalent to `1, 2, 3, 4, 5`).

A.2.12   [cylc] →parameter templates

Parameterized task names (see previous item, and Section 9.6) are expanded, for each parameter value, using string templates. You can assign templates to parameter names here, to override the default templates.

- *type:* a Python-style string template

- *default* for an integer-valued parameter `p`: `_p%(p)s`
  e.g. `foo<run>` becomes `foo_run3` for `run` value `3`.
- *default* for a non integer-valued parameter `p`: `_%(p)s`
  e.g. `foo<run>` becomes `foo_top` for `run` value `top`.
- *example:* `run = -R%(run)s`
  e.g. `foo<run>` becomes `foo-R3` for `run` value `3`.

Note that the values of a parameter named `p` are substituted for `%(p)s`. The `s` indicates a string value and should be used even for integer-valued parameters, because cylc converts integer parameter values to strings, with (depending on size) zero-padding. In `_run%(run)s` the first "run" is a string literal, and the second gets substituted with each value of the parameter.

A.2.13   [cylc] →[[events]]

Cylc has internal "hooks" to which you can attach handlers that are called by the suite daemon whenever certain events occur. This section configures suite event hooks; see A.4.1.15 for task event hooks.

Event handler commands can send an email or an SMS, call a pager, intervene in the operation of their own suite, or whatever. They can be held in the suite bin directory, otherwise it is up to you to ensure their location is in `$PATH` (in the shell in which cylc runs, on the suite host). The commands should require very little resource to run and should return quickly.

Each event handler can be specified as a list of command lines or command line templates.

A command line template may have any or all of these patterns which will be substituted with actual values:

- %(event)s: event name (see below)
- %(suite)s: suite name
- %(message)s: event message, if any

Otherwise, the command line will be called with the following command line arguments:

```
<suite-event-handler> %(event)s %(suite)s '%(message)s'
```

Additional information can be passed to event handlers via [cylc] →[[environment]].

A.2.13.1   [cylc] →[[events]] →EVENT handler

A comma-separated list of one or more event handlers to call when one of the following EVENTs occurs:

- **startup** - the suite has started running
- **shutdown** - the suite is shutting down
- **timeout** - the suite has timed out
- **stalled** - the suite has stalled
- **inactivity** - the suite is inactive

Default values for these can be set at the site level via the siterc file (see B.14.4).

Item details:

- *type:* string (event handler script name)
- *default:* None, unless defined at the site level.
- *example:* `startup handler = my-handler.sh`

A.2.13.2   [cylc] →[[[events]]] →handlers

Specify the general event handlers as a list of command lines or command line templates.

- *type:* Comma-separated list of strings (event handler command line or command line templates).
- *default:* (none)
- *example:* `handlers = my-handler.sh`

A.2.13.3   [cylc] →[[events]] →handler events

Specify the events for which the general event handlers should be invoked.

- *type:* Comma-separated list of events
- *default:* (none)
- *example:* `handler events = timeout, shutdown`

A.2.13.4   [cylc] →[[events]] →mail events

Specify the suite events for which notification emails should be sent.

- *type:* Comma-separated list of events
- *default:* (none)
- *example:* `handler = startup, shutdown, timeout`

A.2.13.5   [cylc] →[[events]] →mail footer

Specify a string or string template to insert to footers of notification emails for both suite events and task events.

A template string may have any or all of these patterns which will be substituted with actual values:

- %(host)s: suite host name
- %(port)s: suite port number
- %(owner)s: suite owner name
- %(suite)s: suite name

- *type:*
- *default:* (none)
- *example:* `mail footer = see: http://localhost/%(owner)s/notes-on/%(suite)s/`

A.2.13.6   [cylc] →[[events]] →mail from

Specify an alternate `from:` email address for suite event notifications.

- *type:* string
- *default:* None, (notifications@HOSTNAME)
- *example:* `mail from = no-reply@your-org`

A.2.13.7   [cylc] →[[events]] →mail smtp

Specify the SMTP server for sending suite event email notifications.

- *type:* string
- *default:* None, (localhost:25)
- *example:* `mail smtp = smtp.yourorg`

A.2.13.8   [cylc] →[[events]] →mail to

A list of email addresses to send suite event notifications. The list can be anything accepted by the `mail` command.

- *type:* string
- *default:* None, (USER@HOSTNAME)
- *example:* `mail to = your.colleague`

A.2.13.9   [cylc] →[[events]] →timeout

If a timeout is set and the timeout event is handled, the timeout event handler(s) will be called if the suite stays in a stalled state for some period of time. The timer is set initially at suite start up. It is possible to set a default for this at the site level (see B.14.4).

- *type:* ISO 8601 duration/interval representation (e.g. `PT5S`, 5 seconds, `PT1S`, 1 second) - minimum 0 seconds.
- *default:* (none), unless set at the site level.

A.2.13.10   [cylc] →[[events]] →inactivity

If inactivity is set and the inactivity event is handled, the inactivity event handler(s) will be called if there is no activity in the suite for some period of time. The timer is set initially at suite start up. It is possible to set a default for this at the site level (see B.14.4).

- *type:* ISO 8601 duration/interval representation (e.g. `PT5S`, 5 seconds, `PT1S`, 1 second) - minimum 0 seconds.
- *default:* (none), unless set at the site level.

A.2.13.11   [cylc] →[[events]] →reset timer

If `True` (the default) the suite timer will continually reset after any task changes state, so you can time out after some interval since the last activity occured rather than on absolute suite execution time.

- *type:* boolean

- *default:* True

A.2.13.12   [cylc] →[[events]] →abort on stalled

If this is set to True it will cause the suite to abort with error status if it stalls. A suite is considered "stalled" if there are no active, queued or submitting tasks or tasks waiting for clock triggers to be met. It is possible to set a default for this at the site level (see B.14.4).

- *type:* boolean
- *default:* False, unless set at the site level.

A.2.13.13   [cylc] →[[events]] →abort on timeout

If a suite timer is set (above) this will cause the suite to abort with error status if the suite times out while still running. It is possible to set a default for this at the site level (see B.14.4).

- *type:* boolean
- *default:* False, unless set at the site level.

A.2.13.14   [cylc] →[[events]] →abort on inactivity

If a suite inactivity timer is set (above) this will cause the suite to abort with error status if the suite is inactive for some period while still running. It is possible to set a default for this at the site level (see B.14.4).

- *type:* boolean
- *default:* False, unless set at the site level.

A.2.13.15   [cylc] →[[events]] →abort if EVENT handler fails

Cylc does not normally care whether an event handler succeeds or fails, but if this is turned on the EVENT handler will be executed in the foreground (which will block the suite while it is running) and the suite will abort if the handler fails.

- *type:* boolean
- *default:* False

A.2.14   [cylc] →[[environment]]

Environment variables defined in this section are passed to suite and task event handlers.

- These variables are not passed to tasks - use task runtime variables for that. Similarly, task runtime variables are not available to event handlers - which are executed by the suite daemon, (not by running tasks) in response to task events.
- Cylc-defined environment variables such as `$CYLC_SUITE_RUN_DIR` are not passed to task event handlers by default, but you can make them available by extracting them to the cylc environment like this:

```
[cylc]
    [[environment]]
        CYLC_SUITE_RUN_DIR = $CYLC_SUITE_RUN_DIR
```

- These variables - unlike task execution environment variables which are written to job scripts and interpreted by the shell at task run time - are not interpreted by the shell prior to use so shell variable expansion expressions cannot be used here.

A.2.14.1   [cylc] →[[environment]] →___VARIABLE___

Replace ___VARIABLE___ with any number of environment variable assignment expressions. Values may refer to other local environment variables (order of definition is preserved) and are not evaluated or manipulated by cylc, so any variable assignment expression that is legal in the shell in which cylc is running can be used (but see the warning above on variable expansions, which will not be evaluated). White space around the '=' is allowed (as far as cylc's file parser is concerned these are just suite configuration items).

- *type:* string
- *default:* (none)
- *examples:*
  - `FOO = $HOME/foo`

A.2.15   [cylc] →[[reference test]]

Reference tests are finite-duration suite runs that abort with non-zero exit status if cylc fails, if any task fails, if the suite times out, or if a shutdown event handler that (by default) compares the test run with a reference run reports failure. See 12.22.

A.2.15.1   [cylc] →[[reference test]] →suite shutdown event handler

A shutdown event handler that should compare the test run with the reference run, exiting with zero exit status only if the test run verifies.

- *type:* string (event handler command name or path)
- *default:* `cylc hook check-triggering`

As for any event handler, the full path can be ommited if the script is located somewhere in `$PATH` or in the suite bin directory.

A.2.15.2   [cylc] →[[reference test]] →required run mode

If your reference test is only valid for a particular run mode, this setting will cause cylc to abort if a reference test is attempted in another run mode.

- *type:* string
- *legal values:* live, dummy, dummy-local, simulation
- *default:* None

A.2.15.3   [cylc] →[[reference test]] →allow task failures

A reference test run will abort immediately if any task fails, unless this item is set, or a list of
expected task failures is provided (below).

- *type:* boolean
- *default:* False

A.2.15.4   [cylc] →[[reference test]] →expected task failures

A reference test run will abort immediately if any task fails, unless *allow task failures* is set
(above) or the failed task is found in a list IDs of tasks that are expected to fail.

- *type:* Comma-separated list of strings (task IDs: `name.cycle_point`).
- *default:* (none)
- *example:* `foo.20120808, bar.20120908`

A.2.15.5   [cylc] →[[reference test]] →live mode suite timeout

The timeout value, expressed as an ISO 8601 duration/interval, after which the test run should
be aborted if it has not finished, in live mode. Test runs cannot be done in live mode unless
you define a value for this item, because it is not possible to arrive at a sensible default for all
suites.

- *type:* ISO 8601 duration/interval representation, e.g. `PT5M` is 5 minutes (note: by contrast
  `P5M` means 5 months, so remember the `T`!).
- *default:* PT1M (1 minute)

A.2.15.6   [cylc] →[[reference test]] →simulation mode suite timeout

The timeout value in minutes after which the test run should be aborted if it has not finished,
in simulation mode. Test runs cannot be done in simulation mode unless you define a value for
this item, because it is not possible to arrive at a sensible default for all suites.

- *type:* ISO 8601 duration/interval representation (e.g. `PT5M`, 5 minutes (note: by contrast,
  `P5M` means 5 months, so remember the `T`!)).
- *default:* PT1M (1 minute)

A.2.15.7   [cylc] →[[reference test]] →dummy mode suite timeout

The timeout value, expressed as an ISO 8601 duration/interval, after which the test run should
be aborted if it has not finished, in dummy mode. Test runs cannot be done in dummy mode
unless you define a value for this item, because it is not possible to arrive at a sensible default
for all suites.

- *type:* ISO 8601 duration/interval representation (e.g. `PT5M`, 5 minutes (note: by contrast,
  `P5M` means 5 months, so remember the `T`!)).
- *default:* PT1M (1 minute)

A.2.16   [cylc] →[[authentication]]

Authentication of client programs with suite daemons can be set in the global site/user config files and overridden here if necessary. See B.15 for more information.

A.2.16.1   [cylc] →[[authentication]] →public

The client privilege level granted for public access - i.e. no suite passphrase required. See B.15 for legal values.

A.2.17   [cylc] →[[simulation]]

Suite-level configuration for the *simulation* and *dummy* run modes described in Section 12.21.

A.2.17.1   [cylc] →[[simulation]] →disable suite event handlers

If this is set to `True` configured suite event handlers will not be called in simulation or dummy modes.

- *type:* boolean
- *default:* `True`

A.3   [scheduling]

This section allows cylc to determine when tasks are ready to run.

A.3.1   [scheduling] →cycling mode

Cylc runs using the proleptic Gregorian calendar by default. This item allows you to either run the suite using the 360 day calendar (12 months of 30 days in a year) or using integer cycling.

- *type:* string
- *legal values:* gregorian, 360day, integer
- *default:* gregorian

A.3.2   [scheduling] →initial cycle point

In a cold start each cycling task (unless specifically excluded under [special tasks]) will be loaded into the suite with this cycle point, or with the closest subsequent valid cycle point for the task. This item can be overridden on the command line or in the gcylc suite start panel.

In date-time cycling, if you do not provide time zone information for this, it will be assumed to be local time, or in UTC if A.2.2 is set, or in the time zone determined by A.2.5 if that is set.

- *type:* ISO 8601 date/time point representation (e.g. `CCYYMMDDThhmm`, 19951231T0630) or "now".
- *default:* (none)

The string "now" converts to the current date-time on the suite host (adjusted to UTC if the suite is in UTC mode but the host is not) to minute resolution. Minutes (or hours, etc.) may be ignored depending on your cycle point format (A.2.3).

A.3.3   [scheduling] →final cycle point

Cycling tasks are held once they pass the final cycle point, if one is specified. Once all tasks have achieved this state the suite will shut down. If this item is provided you can override it on the command line or in the gcylc suite start panel.

In date-time cycling, if you do not provide time zone information for this, it will be assumed to be local time, or in UTC if A.2.2 is set, or in the A.2.5 if that is set.

- *type:* ISO 8601 date/time point representation (e.g. `CCYYMMDDThhmm`, 19951231T1230) or ISO 8601 date/time offset (e.g. +P1D+PT6H)
- *default:* (none)

A.3.4   [scheduling] →initial cycle point constraints

In a cycling suite it is possible to restrict the initial cycle point by defining a list of truncated time points under the initial cycle point constraints.

- *type:* Comma-separated list of ISO 8601 truncated time point representations (e.g. T00, T06, T-30).
- *default:* (none)

A.3.5   [scheduling] →final cycle point constraints

In a cycling suite it is possible to restrict the final cycle point by defining a list of truncated time points under the final cycle point constraints.

- *type:* Comma-separated list of ISO 8601 truncated time point representations (e.g. T00, T06, T-30).
- *default:* (none)

A.3.6   [scheduling] →hold after point

Cycling tasks are held once they pass the hold after cycle point, if one is specified. Unlike the final cycle point suite will not shut down once all tasks have passed this point. If this item is provided you can override it on the command line or in the gcylc suite start panel.

A.3.7   [scheduling] →runahead limit

Runahead limiting prevents the fastest tasks in a suite from getting too far ahead of the slowest ones, as documented in 12.12.

This config item specifies a hard limit as a cycle interval between the slowest and fastest tasks. It is deprecated in favour of the newer default limiting by `max active cycle points` (A.3.8).

- *type:* Cycle interval string e.g. `PT12H` for a 12 hour limit under ISO 8601 cycling.
- *default:* (none)

A.3.8 [scheduling] →max active cycle points

Runahead limiting prevents the fastest tasks in a suite from getting too far ahead of the slowest ones, as documented in 12.12.

This config item supersedes the deprecated hard `runahead limit` (A.3.7). It allows up to `N` (default 3) consecutive cycle points to be active at any time, adjusted up if necessary for any future triggering.

- *type:* integer
- *default:* 3

A.3.9 [scheduling] →spawn to max active cycle points

Allows tasks to spawn out to `max active cycle points` (A.3.8), removing restriction that a task has to have submitted before its successor can be spawned.

Important: This should be used with care given the potential impact of additional task proxies both in terms of memory and cpu for the cylc daemon as well as overheads in rendering all the additional tasks in gcylc. Also, use of the setting may highlight any issues with suite design relying on the default behaviour where downstream tasks would otherwise be waiting on ones upstream submitting and the suite would have stalled e.g. a housekeeping task at a later cycle deleting an earlier cycle's data before that cycle has had chance to run where previously the task would not have been spawned until its predecessor had been submitted.

- *type:* boolean
- *default:* False

A.3.10 [scheduling] →[[queues]]

Configuration of internal queues, by which the number of simultaneously active tasks (submitted or running) can be limited, per queue. By default a single queue called *default* is defined, with all tasks assigned to it and no limit. To use a single queue for the whole suite just set the limit on the *default* queue as required. See also 12.13.

A.3.10.1 [scheduling] →[[queues]] →[[[___QUEUE___]]]

Section heading for configuration of a single queue. Replace ___QUEUE___ with a queue name, and repeat the section as required.

- *type:* string
- *default:* "default"

A.3.10.2 [scheduling] →[[queues]] →[[[___QUEUE___]]] →limit

The maximum number of active tasks allowed at any one time, for this queue.

- *type:* integer
- *default:* 0 (i.e. no limit)

A.3.10.3 [scheduling] →[[queues]] →[[[___QUEUE___]]] →members

A list of member tasks, or task family names, to assign to this queue (assigned tasks will automatically be removed from the default queue).

- *type:* Comma-separated list of strings (task or family names).
- *default:* none for user-defined queues; all tasks for the "default" queue

A.3.11 [scheduling] →[[special tasks]]

This section is used to identify tasks with special behaviour. Family names can be used in special task lists as shorthand for listing all member tasks.

A.3.11.1 [scheduling] →[[special tasks]] →clock-trigger

Clock-trigger tasks (see 9.3.5.14) wait on a wall clock time specified as an offset from their own cycle point.

- *type:* Comma-separated list of task or family names with associated date-time offsets expressed as ISO8601 interval strings, positive or negative, e.g. PT1H for 1 hour. The offset specification may be omitted to trigger right on the cycle point.
- *default:* (none)
- *example:*
  ```
  clock-trigger = foo(PT1H30M), bar(PT1.5H), baz
  ```

A.3.11.2 [scheduling] →[[special tasks]] →clock-expire

Clock-expire tasks enter the *expired* state and skip job submission if too far behind the wall clock when they become ready to run. The expiry time is specified as an offset from wall-clock time; typically it should be negative - see 9.3.5.15.

- *type:* Comma-separated list of task or family names with associated date-time offsets expressed as ISO8601 interval strings, positive or negative, e.g. PT1H for 1 hour. The offset may be omitted if it is zero.
- *default:* (none)
- *example:*
  ```
  clock-expire = foo(-P1D)
  ```

A.3.11.3   [scheduling] →[[special tasks]] →external-trigger

Externally triggered tasks (see 9.3.5.16) wait on external events reported via the `cylc ext-trigger` command. To constrain triggers to a specific cycle point, include `$CYLC_TASK_CYCLE_POINT` in the trigger message string and pass the cycle point to the `cylc ext-trigger` command.

- *type:* Comma-separated list of task names with associated external trigger message strings.
- *default:* (none)
- *example:* (note the comma and line-continuation character)

```
external-trigger = get-satx("new sat-X data ready"), \
                   get-saty("new sat-Y data ready for
                       $CYLC_TASK_CYCLE_POINT")
```

A.3.11.4   [scheduling] →[[special tasks]] →sequential

Sequential tasks are automatically given dependence on their own predecessor. This is equivalent to use of explicit inter-cycle triggers in the graph, except that the automatic version does not show in suite graph visualization. For more on sequential tasks see 9.3.5.12 and **??**.

- *type:* Comma-separated list of task or family names.
- *default:* (none)
- *example:* `sequential = foo, bar`

A.3.11.5   [scheduling] →[[special tasks]] →exclude at start-up

Any task listed here will be excluded from the initial task pool (this goes for suite restarts too). If an *inclusion* list is also specified, the initial pool will contain only included tasks that have not been excluded. Excluded tasks can still be inserted at run time. Other tasks may still depend on excluded tasks if they have not been removed from the suite dependency graph, in which case some manual triggering, or insertion of excluded tasks, may be required.

- *type:* Comma-separated list of task or family names.
- *default:* (none)

A.3.11.6   [scheduling] →[[special tasks]] →include at start-up

If this list is not empty, any task *not* listed in it will be excluded from the initial task pool (this goes for suite restarts too). If an *exclusion* list is also specified, the initial pool will contain only included tasks that have not been excluded. Excluded tasks can still be inserted at run time. Other tasks may still depend on excluded tasks if they have not been removed from the suite dependency graph, in which case some manual triggering, or insertion of excluded tasks, may be required.

- *type:* Comma-separated list of task or family names.
- *default:* (none)

A.3.12   [scheduling] →[[dependencies]]

The suite dependency graph is defined under this section. You can plot the dependency graph as you work on it, with `cylc graph` or by right clicking on the suite in the db viewer. See also 9.3.

A.3.12.1   [scheduling] →[[dependencies]] →graph

The dependency graph for a completely non-cycling suites can go here. See also A.3.12.2.1 below and 9.3, for graph string syntax.

- *type:* string
- *example:* (see A.3.12.2.1 below)

A.3.12.2   [scheduling] →[[dependencies]] →[[[___RECURRENCE___]]]

___RECURRENCE___ section headings define the sequence of cycle points for which the subsequent graph section is valid. These should be specified in our ISO 8601 derived sequence syntax, or similar for integer cycling:

- *examples:*
  - date-time cycling: `[[[T00,T06,T12,T18]]]` or `[[[PT6H]]]`
  - integer cycling (stepped by 2): `[[[P2]]]`
- *default:* (none)

See 9.3.3 for more on recurrence expressions, and how multiple graph sections combine.

A.3.12.2.1   [scheduling] →[[dependencies]] →[[[___RECURRENCE___]]] →graph

The dependency graph for a given recurrence section goes here. Syntax examples follow; see also 9.3 and 9.3.5.

- *type:* string
- *examples:*
  ```
  graph = """
      foo => bar => baz & waz    # baz and waz both trigger off bar
      foo[-P1D-PT6H] => bar      # bar triggers off foo[-P1D-PT6H]
      baz:out1 => faz            # faz triggers off a message output of baz
      X:start => Y               # Y triggers if X starts executing
      X:fail => Y                # Y triggers if X fails
      foo[-PT6H]:fail => bar     # bar triggers if foo[-PT6H] fails
      X => !Y                    # Y suicides if X succeeds
      X | X:fail => Z            # Z triggers if X succeeds or fails
      X:finish => Z              # Z triggers if X succeeds or fails
      (A | B & C ) | D => foo    # general conditional triggers
      foo:submit => bar          # bar triggers if foo is successfully
          submitted
      foo:submit-fail => bar     # bar triggers if submission of foo fails
      # comment
      """
  ```

- *default:* (none)

A.4 [runtime]

This section is used to specify how, where, and what to execute when tasks are ready to run. Common configuration can be factored out in a multiple-inheritance hierarchy of runtime namespaces that culminates in the tasks of the suite. Order of precedence is determined by the C3 linearization algorithm as used to find the *method resolution order* in Python language class hiearchies. For details and examples see 9.4.

A.4.1 [runtime] →[[___NAME___]]

Replace ___NAME___ with a namespace name, or a comma-separated list of names, and repeat as needed to define all tasks in the suite. Names may contain letters, digits, underscores, and hyphens. A namespace represents a group or family of tasks if other namespaces inherit from it, or a task if no others inherit from it.

- *legal values:*
  - `[[foo]]`
  - `[[foo, bar, baz]]`

If multiple names are listed the subsequent settings apply to each.

All namespaces inherit initially from *root*, which can be explicitly configured to provide or override default settings for all tasks in the suite.

A.4.1.1 [runtime] →[[___NAME___]] →inherit

A list of the immediate parent(s) this namespace inherits from. If no parents are listed `root` is assumed.

- *type:* Comma-separated list of strings (parent namespace names).
- *default:* `root`

A.4.1.2 [runtime] →[[___NAME___]] →title

A single line description of this namespace. It is displayed by the `cylc list` command and can be retrieved from running tasks with the `cylc show` command.

- *type:* single line string
- *root default:* (none)

A.4.1.3 [runtime] →[[___NAME___]] →description

A multi-line description of this namespace, retrievable from running tasks with the `cylc show` command.

- *type:* multi-line string
- *root default:* (none)

A.4.1.4   [runtime] →[[___NAME___]] →URL

A web URL to task documentation for this suite. If present it can be browsed with the `cylc doc` command, or by right-clicking on the task in gcylc. The variables `${CYLC_SUITE_NAME}` and `${CYLC_TASK_NAME}` will be replaced with the actual suite and task names (note that these are not environment variables in this context). See also suite URLs (A.1.4).

- *type:* string (URL)
- *default:* (none)
- *example:* you can set URLs to all tasks in a suite by putting something like the following in the root namespace:

```
[runtime]
    [[root]]
        URL = http://suites/${CYLC_SUITE_NAME}/${CYLC_TASK_NAME}.html
```

(Note that URLs containing the suite comment delimiter `#` must be protected by quotes).

A.4.1.5   [runtime] →[[___NAME___]] →init-script

This is invoked by the task job script before the task execution environment is configured, so it does not have access to any suite or task environment variables. It can be a single command or multiple lines of scripting. The original intention was to allow remote tasks to source login scripts to configure their access to cylc, but this should no longer be necessary (see 12.7). See also `env-script`, `err-script`, `pre-script`, `script`, and `post-script`.

- *type:* string
- *default:* (none)
- *example:* `init-script = "echo Hello World"`

A.4.1.6   [runtime] →[[___NAME___]] →env-script

This is invoked by the task job script between the cylc-defined environment (suite and task identity, etc.) and the user-defined task runtime environment - i.e. it has access to the cylc environment, and the task environment has access to variables defined by this scripting. It can be a single command or multiple lines of scripting. See also `init-script`, `err-script`, `pre-script`, `script`, and `post-script`.

- *type:* string
- *default:* (none)
- *example:* `env-script = "echo Hello World"`

A.4.1.7   [runtime] →[[___NAME___]] →err-script

This is any custom script to be invoked at the end of the error trap, (if the error trap is triggered due to failure of a command in the task job). The output of this will always be sent to STDERR and `$1` is set to the name of the signal caught by the error trap. The script should be fast and use very little system resource to ensure that the error trap can return quickly. It can be a single command or multiple lines of scripting. See also `init-script`, `env-script`, `pre-script`, `script`, and `post-script`.

- *type:* string
- *default:* (none)
- *example:* `err-script = "printenv FOO"`

A.4.1.8   [runtime] →[[___NAME___]] →pre-script

This is invoked by the task job script immediately before the `script` item (just below). It can be a single command or multiple lines of scripting. See also `init-script`, `env-script`, `err-script`, `script`, and `post-script`.

- *type:* string
- *default:* (none)
- *example:*
    ```
    pre-script = """
      . $HOME/.profile
      echo Hello from suite ${CYLC_SUITE_NAME}!"""
    ```


A.4.1.9   [runtime] →[[___NAME___]] →script

The is the main user-defined scripting to run when the task is ready. It can be a single command or multiple lines of scripting. See also `init-script`, `env-script`, `err-script`, `pre-script`, and `post-script`.

- *type:* string
- *root default:* (none)

A.4.1.10   [runtime] →[[___NAME___]] →post-script

This is invoked by the task job script immediately after the `script` item (just above). It can be a single command or multiple lines of scripting. See also `init-script`, `env-script`, `err-script`, `pre-script`, and `script`.

- *type:* string
- *default:* (none)

A.4.1.11   [runtime] →[[___NAME___]] →work sub-directory

Task job scripts are executed from within *work directories* created automatically under the suite run directory. A task can get its own work directory from `$CYLC_TASK_WORK_DIR` (or simply `$PWD` if it does not `cd` elsewhere at runtime). The default directory path contains task name and cycle point, to provide a unique workspace for every instance of every task. If several tasks need to exchange files and simply read and write from their from current working directory, this item can be used to override the default to make them all use the same workspace.

The top level share and work directory location can be changed (e.g. to a large data area) by a global config setting (see B.9.1.2).

- *type:* string (directory path, can contain environment variables)
- *default:* `$CYLC_TASK_CYCLE_POINT/$CYLC_TASK_NAME`

- *example:* `$CYLC_TASK_CYCLE_POINT/shared/`

Note that if you omit cycle point from the work sub-directory path successive instances of the task will share the same workspace. Consider the effect on cycle point offset housekeeping of work directories before doing this.

A.4.1.12   [runtime] →[[___NAME___]] →enable resurrection

If a message is received from a failed task cylc will normally treat this as an error condition, issue a warning, and leave the task in the "failed" state. But if "enable resurrection" is switched on failed tasks can come back from the dead: if the same task job script is executed again cylc will put the task back into the running state and continue as normal when the started message is received. This can be used to handle HPC-style job preemption wherein a resource manager may kill a running task and reschedule it to run again later, to make way for a job with higher immediate priority. See also 12.17

- *type:* boolean
- *default:* False

A.4.1.13   [runtime] →[[___NAME___]] →[[[job]]]

This section configures the means by which cylc submits task job scripts to run.

A.4.1.13.1   [runtime] →[[___NAME___]] →[[[job]]] →batch system

See 11 for how job submission works, and how to define new handlers for different batch systems. Cylc has a number of built in batch system handlers:

- *type:* string
- *legal values:*
    - `background` - invoke a child process
    - `at` - the rudimentary Unix `at` scheduler
    - `loadleveler` - IBM LoadLeveler `llsubmit`, with directives defined in the suite.rc file
    - `lsf` - IBM Platform LSF `bsub`, with directives defined in the suite.rc file
    - `pbs` - PBS `qsub`, with directives defined in the suite.rc file
    - `sge` - Sun Grid Engine `qsub`, with directives defined in the suite.rc file
    - `slurm` - Simple Linux Utility for Resource Management `sbatch`, with directives defined in the suite.rc file
    - `moab` - Moab workload manager `msub`, with directives defined in the suite.rc file
- *default:* `background`

A.4.1.13.2   [runtime] →[[___NAME___]] →[[[job]]] →execution time limit

Specify the execution wall clock limit for a job of the task. For `background` and `at`, the job script will be invoked using the `timeout` command. For other batch systems, the specified time will be automatically translated into the equivalent directive for wall clock limit.

- *type:* ISO 8601 duration/interval representation

- *example:* `PT5M`, 5 minutes, `PT1H`, 1 hour
- *default:* (none)

A.4.1.13.3 [runtime] →[[___NAME___]] →[[[job]]] →batch submit command template

This allows you to override the actual command used by the chosen batch system. The template's %(job)s will be substituted by the job file path.

- *type:* string
- *legal values:* a string template
- *example:* `llsubmit \%(job)s`

A.4.1.13.4 [runtime] →[[___NAME___]] →[[[job]]] →shell

Location of the command used to interpret the job script submitted by the suite daemon when a task is ready to run. This can be set to the location of `bash` in the job host if the shell is not installed in the standard location. *Note: It has no bearing on any sub-shells that may be called by the job script.*

Setting this to the path of a ksh93 interpreter is deprecated. Support of which will be withdrawn in a future cylc release. Setting this to any other shell is not supported.

- *type:* string
- *root default:* `/bin/bash`

A.4.1.13.5 [runtime] →[[___NAME___]] →[[[job]]] →submission retry delays

A list of duration (in ISO 8601 syntax), after which to resubmit if job submission fails.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *example:* `PT1M,3*PT1H`, `P1D` is equivalent to `PT1M, PT1H, PT1H, PT1H, P1D` - 1 minute, 1 hour, 1 hour, 1 hour, 1 day.
- *default:* (none)

A.4.1.13.6 [runtime] →[[___NAME___]] →[[[job]]] →execution retry delays

A list of ISO 8601 time duration/intervals after which to resubmit the task if it fails. The variable `$CYLC_TASK_TRY_NUMBER` in the task execution environment is incremented each time, starting from 1 for the first try - this can be used to vary task behaviour by try number.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *example:* `PT1.5M,3*PT10M` is equivalent to `PT1.5M, PT10M, PT10M, PT10M` - 1.5 minutes, 10 minutes, 10 minutes, 10 minutes.
- *default:* (none)

A.4.1.13.7   [runtime] →[[___NAME___]] →[[[job]]] →submission polling intervals

A list of intervals, expressed as ISO 8601 duration/intervals, with optional multipliers, after which cylc will poll for status while the task is in the submitted state.

For the polling task communication method this overrides the default submission polling interval in the site/user config files (6). For default and ssh task communications, polling is not done by default but it can still be configured here as a regular check on the health of submitted tasks.

Each list value is used in turn until the last, which is used repeatedly until finished.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *example:* `PT1M,3*PT1H, PT1M` is equivalent to `PT1M, PT1H, PT1H, PT1H, PT1M` - 1 minute, 1 hour, 1 hour, 1 hour, 1 minute.
- *default:* (none)

A single interval value is probably appropriate for submission polling.

A.4.1.13.8   [runtime] →[[___NAME___]] →[[[job]]] →execution polling intervals

A list of intervals, expressed as ISO 8601 duration/intervals, with optional multipliers, after which cylc will poll for status while the task is in the running state.

For the polling task communication method this overrides the default execution polling interval in the site/user config files (6). For default and ssh task communications, polling is not done by default but it can still be configured here as a regular check on the health of submitted tasks.

Each list value is used in turn until the last, which is used repeatedly until finished.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *example:* `PT1M,3*PT1H, PT1M` is equivalent to `PT1M, PT1H, PT1H, PT1H, PT1M` - 1 minute, 1 hour, 1 hour, 1 hour, 1 minute.
- *default:* (none)

A.4.1.14   [runtime] →[[___NAME___]] →[[[remote]]]

Configure host and username, for tasks that do not run on the suite host account. Non-interactive ssh is used to submit the task by the configured batch system, so you must distribute your ssh key to allow this. Cylc must be installed on remote task hosts, but no external software dependencies are required there.

A.4.1.14.1   [runtime] →[[___NAME___]] →[[[remote]]] →host

The remote host for this namespace. This can be a static hostname, an environment variable that holds a hostname, or a command that prints a hostname to stdout. Host selection commands are executed just prior to job submission. The host (static or dynamic) may have an entry in the cylc site or user config file to specify parameters such as the location of cylc on the remote machine; if not, the corresponding local settings (on the suite host) will be assumed to apply on the remote host.

- *type:* string (a valid hostname on the network)
- *default:* (none)
- *examples:*
  - static host name: `host = foo`
  - fully qualified: `host = foo.bar.baz`
  - dynamic host selection:
    * shell command (1): `host = $(host-selector.sh)`
    * shell command (2): `host = ‘host-selector.sh)‘`
    * environment variable: `host = $MY_HOST`

A.4.1.14.2   [runtime] →[[___NAME___]] →[[[remote]]] →owner

The username of the task host account. This is (only) used in the non-interactive ssh command invoked by the suite daemon to submit the remote task (consequently it may be defined using local environment variables (i.e. the shell in which cylc runs, and [cylc] →[[environment]]).

If you use dynamic host selection and have different usernames on the different selectable hosts, you can configure your `$HOME/.ssh/config` to handle username translation.

- *type:* string (a valid username on the remote host)
- *default:* (none)

A.4.1.14.3   [runtime] →[[___NAME___]] →[[[remote]]] →retrieve job logs

Remote task job logs are saved to the suite run directory on the task host, not on the suite host. If you want the job logs pulled back to the suite host automatically, you can set this item to `True`. The suite will then attempt to `rsync` the job logs once from the remote host each time a task job completes. E.g. if the job file is `~/cylc-run/tut.oneoff.remote/log/job/1/hello/01/job`, anything under `~/cylc-run/tut.oneoff.remote/log/job/1/hello/01/` will be retrieved.

- *type:* boolean
- *default:* False

A.4.1.14.4   [runtime] →[[___NAME___]] →[[[remote]]] →retrieve job logs max size

If the disk space of the suite host is limited, you may want to set the maximum sizes of the job log files to retrieve. The value can be anything that is accepted by the `--max-size=SIZE` option of the `rsync` command.

- *type:* string
- *default:* None

A.4.1.14.5   [runtime] →[[___NAME___]] →[[[remote]]] →retrieve job logs retry delays

Some batch systems have considerable delays between the time when the job completes and when it writes the job logs in its normal location. If this is the case, you can configure an initial delay and some retry delays between subsequent attempts. The default behaviour is to attempt once without any delay.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *default:* (none)
- *example:* `handler = PT10S, PT1M, PT5M`

A.4.1.14.6   [runtime] →[[___NAME___]] →[[[remote]]] →suite definition directory

The path to the suite definition directory on the remote host, needed if remote tasks require access to files stored there (via `$CYLC_SUITE_DEF_PATH`) or in the suite bin directory (via `$PATH`). If this item is not defined, the local suite definition directory path will be assumed, with the suite owner's home directory, if present, replaced by `'$HOME'` for interpretation on the remote host.

- *type:* string (a valid directory path on the remote host)
- *default:* (local suite definition path with `$HOME` replaced)

A.4.1.15   [runtime] →[[___NAME___]] →[[[events]]]

Cylc can call nominated event handlers when certain task events occur. This section configures specific task event handlers; see A.2.13 for suite event hooks.

Event handler commands can be located in the suite `bin/` directory, otherwise it is up to you to ensure their location is in `$PATH` (in the shell in which cylc runs, on the suite host). The commands should require very little resource to run and should return quickly.

Each task event handler can be specified as a list of command lines or command line templates.

A command line template may have any or all of these patterns which will be substituted with actual values:

- %(event)s: event name
- %(suite)s: suite name
- %(point)s: cycle point
- %(name)s: task name
- %(submit_num)s: submit number
- %(id)s: task ID (i.e. %(name)s.%(point)s)
- %(message)s: event message, if any

Otherwise, the command line will be called with the following arguments:

```
<task-event-handler> %(event)s %(suite)s %(id)s %(message)s
```

For an explanation of the substitution syntax, see String Formatting Operations in the Python documentation: https://docs.python.org/2/library/stdtypes.html#string-formatting.

Additional information can be passed to event handlers via the [cylc] →[[environment]] (but not via task runtime environments - event handlers are not called by tasks).

A.4.1.15.1   [runtime] →[[___NAME___]] →[[[events]]] →EVENT handler

A list of one or more event handlers to call when one of the following EVENTs occurs:

- **submitted** - the job submit command was successful

- **submission failed** - the job submit command failed, or the submitted job was killed before it started executing
- **submission retry** - job submit failed, but cylc will resubmit it after a configured delay
- **submission timeout** - the submitted job timed out without commencing execution
- **started** - the task reported commencement of execution
- **succeeded** - the task reported that it completed successfully
- **failed** - the task reported that if tailed to complete successfully
- **retry** - the task failed, but cylc will resubmit it after a configured delay
- **execution timeout** - the task timed out after execution commenced
- **warning** - the task reported a warning priority message

Item details:

- *type:* Comma-separated list of strings (event handler scripts).
- *default:* None
- *example:* `failed handler = my-failed-handler.sh`

A.4.1.15.2   [runtime] →[[___NAME___]] →[[[events]]] →submission timeout

If a task has not started after the specified ISO 8601 duration/interval, the *submission timeout* event handler(s) will be called.

- *type:* ISO 8601 duration/interval representation (e.g. `PT30M`, 30 minutes or `P1D`, 1 day).
- *default:* (none)

A.4.1.15.3   [runtime] →[[___NAME___]] →[[[events]]] →execution timeout

If a task has not finished after the specified ISO 8601 duration/interval, the *execution timeout* event handler(s) will be called.

- *type:* ISO 8601 duration/interval representation (e.g. `PT4H`, 4 hours or `P1D`, 1 day).
- *default:* (none)

A.4.1.15.4   [runtime] →[[___NAME___]] →[[[events]]] →reset timer

If you set an execution timeout the timer can be reset to zero every time a message is received from the running task (which indicates the task is still alive). Otherwise, the task will timeout if it does not finish in the alotted time regardless of incoming messages.

- *type:* boolean
- *default:* False

A.4.1.15.5   [runtime] →[[___NAME___]] →[[[events]]] →handlers

Specify a list of command lines or command line templates as task event handlers.

- *type:* Comma-separated list of strings (event handler command line or command line templates).
- *default:* (none)

- *example:* `handlers = my-handler.sh`

A.4.1.15.6   [runtime] →[[___NAME___]] →[[[events]]] →handler events

Specify the events for which the general task event handlers should be invoked.

- *type:* Comma-separated list of events
- *default:* (none)
- *example:* `handler events = submission failed, failed`

A.4.1.15.7   [runtime] →[[___NAME___]] →[[[events]]] →handler retry delays

Specify an initial delay before running an event handler command and any retry delays in case the command returns a non-zero code. The default behaviour is to run an event handler command once without any delay.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *default:* (none)
- *example:* `handler = PT10S, PT1M, PT5M`

A.4.1.15.8   [runtime] →[[___NAME___]] →[[[events]]] →mail events

Specify the events for which notification emails should be sent.

- *type:* Comma-separated list of events
- *default:* (none)
- *example:* `handler = submission failed, failed`

A.4.1.15.9   [runtime] →[[___NAME___]] →[[[events]]] →mail from

Specify an alternate `from:` email address for event notifications.

- *type:* string
- *default:* None, (notifications@HOSTNAME)
- *example:* `mail from = no-reply@your-org`

A.4.1.15.10   [runtime] →[[___NAME___]] →[[[events]]] →mail retry delays

Specify an initial delay before running the mail notification command and any retry delays in case the command returns a non-zero code. The default behaviour is to run the mail notification command once without any delay.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *default:* (none)
- *example:* `handler = PT10S, PT1M, PT5M`

A.4.1.15.11   [runtime] →[[___NAME___]] →[[[events]]] →mail smtp

Specify the SMTP server for sending email notifications.

- *type:* string
- *default:* None, (localhost:25)
- *example:* `mail smtp = smtp.yourorg`

A.4.1.15.12   [runtime] →[[___NAME___]] →[[[events]]] →mail to

A list of email addresses to send task event notifications. The list can be anything accepted by
the `mail` command.

- *type:* string
- *default:* None, (USER@HOSTNAME)
- *example:* `mail to = your.colleague`

A.4.1.16   [runtime] →[[___NAME___]] →[[[environment]]]

The user defined task execution environment. Variables defined here can refer to cylc suite and
task identity variables, which are exported earlier in the task job script, and variable assignment
expressions can use cylc utility commands because access to cylc is also configured earlier in
the script. See also 9.4.7.

A.4.1.16.1   [runtime] →[[___NAME___]] →[[[environment]]] →___VARIABLE___

Replace ___VARIABLE___ with any number of environment variable assignment expressions.
Order of definition is preserved so values can refer to previously defined variables. Values are
passed through to the task job script without evaluation or manipulation by cylc, so any variable
assignment expression that is legal in the job submission shell can be used. White space around
the '=' is allowed (as far as cylc's suite.rc parser is concerned these are just normal configuration
items).

- *type:* string
- *default:* (none)
- *legal values:* depends to some extent on the task job submission shell (A.4.1.13.4).
- *examples*, for the bash shell:
    - `FOO = $HOME/bar/baz`
    - `BAR = ${FOO}$GLOBALVAR`
    - `BAZ = $( echo "hello world")`
    - `WAZ = ${FOO%.jpg}.png`
    - `NEXT_CYCLE = $( cylc cycle-point --offset=PT6H )`
    - `PREV_CYCLE = `cylc cycle-point --offset=-PT6H``
    - `ZAZ = "${FOO#bar}"#<-- QUOTED to escape the suite.rc comment character`

A.4.1.17   [runtime] →[[___NAME___]] →[[[environment filter]]]

This section contains environment variable inclusion and exclusion lists that can be used to filter the inherited environment. *This is not intended as an alternative to a well-designed inheritance hierarchy that provides each task with just the variables it needs.* Filters can, however, improve suites with tasks that inherit a lot of environment they don't need, by making it clear which tasks use which variables. They can optionally be used routinely as explicit "task environment interfaces" too, at some cost to brevity, because they guarantee that variables filtered out of the inherited task environment are not used.

Note that environment filtering is done after inheritance is completely worked out, not at each level on the way, so filter lists in higher-level namespaces only have an effect if they are not overridden by descendants.

A.4.1.17.1   [runtime] →[[___NAME___]] →[[[environment filter]]] →include

If given, only variables named in this list will be included from the inherited environment, others will be filtered out. Variables may also be explicitly excluded by an `exclude` list.

- *type:* Comma-separated list of strings (variable names).
- *default:* (none)

A.4.1.17.2   [runtime] →[[___NAME___]] →[[[environment filter]]] →exclude

Variables named in this list will be filtered out of the inherited environment. Variables may also be implicitly excluded by omission from an `include` list.

- *type:* Comma-separated list of strings (variable names).
- *default:* (none)

A.4.1.18   [runtime] →[[___NAME___]] →[[[directives]]]

Batch queue scheduler directives. Whether or not these are used depends on the batch system. For the built-in methods that support directives (`loadleveler`, `lsf`, `pbs`, `sge`, `slurm`, `moab`), directives are written to the top of the task job script in the correct format for the method. Specifying directives individually like this allows use of default directives that can be individually overridden at lower levels of the runtime namespace hierarchy.

A.4.1.18.1   [runtime] →[[___NAME___]] →[[[directives]]] →___DIRECTIVE___

Replace ___DIRECTIVE___ with each directive assignment, e.g. `class = parallel`

- *type:* string
- *default:* (none)

Example directives for the built-in batch system handlers are shown in 11.1.

A.4.1.19   [runtime] →[[___NAME___]] →[[[outputs]]]

Register custom task outputs for use in message triggering in this section (9.3.5.5)

A.4.1.19.1   [runtime] →[[___NAME___]] →[[[outputs]]] →___OUTPUT___

Replace ___OUTPUT___ with one or more custom task output messages (9.3.5.5). The item
name is used to select the custom output message in graph trigger notation.

- *type:* string
- *default:* (none)
- *examples:*

```
out1 = "sea state products ready"
out2 = "NWP restart files completed"
```

A.4.1.20   [runtime] →[[___NAME___]] →[[[suite state polling]]]

Configure automatic suite polling tasks as described in 12.23. The items in this section reflect
the options and defaults of the `cylc suite-state` command, except that the target suite name
and the `--task`, `--cycle`, and `--status` options are taken from the graph notation.

A.4.1.20.1   [runtime] →[[___NAME___]] →[[[suite state polling]]] →run-dir

For your own suites the run database location is determined by your site/user config. For other
suites, e.g. those owned by others, or mirrored suite databases, use this item to specify the
location of the top level cylc run directory (the database should be a suite-name sub-directory
of this location).

- *type:* string (a directory path on the target suite host)
- *default:* as configured by site/user config (for your own suites)

A.4.1.20.2   [runtime] →[[___NAME___]] →[[[suite state polling]]] →template

Cycle point template of the target suite, if different from that of the polling suite.

- *type:* string
- *default:* cycle point format of the polling suite
- *example:* `\%Y-\%m-\%dT\%H`

A.4.1.20.3   [runtime] →[[___NAME___]] →[[[suite state polling]]] →interval

Polling interval expressed as an ISO 8601 duration/interval.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:* PT1M

A.4.1.20.4   [runtime] →[[___NAME___]] →[[[suite state polling]]] →max-polls

The maximum number of polls before timing out and entering the 'failed' state.

- *type:* integer
- *default:* 10

A.4.1.20.5   [runtime] →[[___NAME___]] →[[[suite state polling]]] →user

Username of an account on the suite host to which you have access. The polling `cylc suite-state`
command will be invoked on the remote account.

- *type:* string (username)
- *default:* (none)

A.4.1.20.6   [runtime] →[[___NAME___]] →[[[suite state polling]]] →host

The hostname of the target suite. The polling `cylc suite-state` command will be invoked on the
remote account.

- *type:* string (hostname)
- *default:* (none)

A.4.1.20.7   [runtime] →[[___NAME___]] →[[[suite state polling]]] →verbose

Run the polling `cylc suite-state` command in verbose output mode.

- *type:* boolean
- *default:* False

A.4.1.21   [runtime] →[[___NAME___]] →[[[simulation]]]

Task configuration for the suite *simulation* and *dummy* run modes described in Section 12.21.

A.4.1.21.1   [runtime] →[[___NAME___]] →[[[simulation]]] →default run length

The default simulated job run length, if `[job]execution time limit` and `[simulation]speedup factor`
are not set.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:* `PT10S`

A.4.1.21.2   [runtime] →[[___NAME___]] →[[[simulation]]] →speedup factor

If `[job]execution time limit` is set, the task simulated run length is computed by dividing it by
this factor.

- *type:* float
- *default:* (none) - i.e. do not use proportional run length

- *example:* `10.0`

A.4.1.21.3   [runtime] →[[___NAME___]] →[[[simulation]]] →time limit buffer

For dummy jobs, a new `[job]execution time limit` is set to the simulated task run length plus this buffer interval, to avoid job kill due to exceeding the time limit.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:* PT10S

A.4.1.21.4   [runtime] →[[___NAME___]] →[[[simulation]]] →fail cycle points

Configure simulated or dummy jobs to fail at certain cycle points.

- *type:* list of strings (cycle points), or *all*
- *default:* (none) - no instances of the task will fail
- *examples:*
  - `all` - all instance of the task will fail
  - `2017-08-12T06, 2017-08-12T18` - these instances of the task will fail

A.4.1.21.5   [runtime] →[[___NAME___]] →[[[simulation]]] →fail try 1 only

If this is set to `True` only the first run of the task instance will fail, otherwise retries will fail too.

- *type:* boolean
- *default:* `True`

A.4.1.21.6   [runtime] →[[___NAME___]] →[[[simulation]]] →disable task event handlers

If this is set to `True` configured task event handlers will not be called in simulation or dummy modes.

- *type:* boolean
- *default:* `True`

A.5   [visualization]

Configuration of suite graphing for the `cylc graph` command (graph extent, styling, and initial family-collapsed state) and the gcylc graph view (initial family-collapsed state). Graphviz documentation of node shapes and so on can be found at http://www.graphviz.org/Documentation.php.

A.5.1   [visualization] →initial cycle point

The initial cycle point for graph plotting.

- *type:* ISO 8601 date/time representation (e.g. CCYYMMDDThhmm)
- *default:* the suite initial cycle point

The visualization initial cycle point gets adjusted up if necessary to the suite initial cycling point.

A.5.2   [visualization] →final cycle point

An explicit final cycle point for graph plotting. If used, this overrides the preferred *number of cycle points* (below).

- *type:* ISO 8601 date/time representation (e.g. CCYYMMDDThhmm)
- *default:* (none)

The visualization final cycle point gets adjusted down if necessary to the suite final cycle point.

A.5.3   [visualization] →number of cycle points

The number of cycle points to graph starting from the visualization initial cycle point. This is the preferred way of defining the graph end point, but it can be overridden by an explicit *final cycle point* (above).

- *type:* integer
- *default:* 3

A.5.4   [visualization] →collapsed families

A list of family (namespace) names to be shown in the collapsed state (i.e. the family members will be replaced by a single family node) when the suite is first plotted in the graph viewer or the gcylc graph view. If this item is not set, the default is to collapse all families at first. Interactive GUI controls can then be used to group and ungroup family nodes at will.

- *type:* Comma-separated list of family names.
- *default:* (none)

A.5.5   [visualization] →use node color for edges

Plot graph edges (dependency arrows) with the same color as the upstream node, otherwise default to black.

- *type:* boolean
- *default:* False

A.5.6   [visualization] →use node fillcolor for edges

Plot graph edges (i.e. dependency arrows) with the same fillcolor as the upstream node, if it is filled, otherwise default to black.

- *type:* boolean
- *default:* False

A.5.7   [visualization] →node penwidth

Line width of node shape borders.

- *type:* integer
- *default:* 2

A.5.8   [visualization] →edge penwidth

Line width of graph edges (dependency arrows).

- *type:* integer
- *default:* 2

A.5.9   [visualization] →use node color for labels

Graph node labels can be printed in the same color as the node outline.

- *type:* boolean
- *default:* False

A.5.10   [visualization] →default node attributes

Set the default attributes (color and style etc.) of graph nodes (tasks and families). Attribute pairs must be quoted to hide the internal `=` character.

- *type:* Comma-separated list of quoted `'attribute=value'` pairs.
- *legal values:* see graphviz or pygraphviz documentation
- *default:* `'style=filled'`, `'fillcolor=yellow'`, `'shape=box'`

A.5.11   [visualization] →default edge attributes

Set the default attributes (color and style etc.) of graph edges (dependency arrows). Attribute pairs must be quoted to hide the internal `=` character.

- *type:* Comma-separated list of quoted `'attribute=value'` pairs.
- *legal values:* see graphviz or pygraphviz documentation
- *default:* `'color=black'`

A.5.12   [visualization] →[[node groups]]

Define named groups of graph nodes (tasks and families) which can styled en masse, by name, in [visualization] →[[node attributes]]. Node groups are automatically defined for all task families, including root, so you can style family and member nodes at once by family name.

A.5.12.1   [visualization] →[[node groups]] →___GROUP___

Replace ___GROUP___ with each named group of tasks or families.

- *type:* Comma-separated list of task or family names.
- *default:* (none)
- *example:*

```
PreProc = foo, bar
PostProc = baz, waz
```

A.5.13   [visualization] →[[node attributes]]

Here you can assign graph node attributes to specific nodes, or to all members of named groups defined in [visualization] →[[node groups]]. Task families are automatically node groups. Styling of a family node applies to all member nodes (tasks and sub-families), but precedence is determined by ordering in the suite definition. For example, if you style a family red and then one of its members green, cylc will plot a red family with one green member; but if you style one member green and then the family red, the red family styling will override the earlier green styling of the member.

A.5.13.1   [visualization] →[[node attributes]] →___NAME___

Replace ___NAME___ with each node or node group for style attribute assignment.

- *type:* Comma-separated list of quoted `'attribute=value'` pairs.
- *legal values:* see graphviz or pygraphviz documentation
- *default:* (none)
- *example:* (with reference to the node groups defined above)

```
PreProc = 'style=filled', 'fillcolor=orange'
PostProc = 'color=red'
foo = 'style=filled'
```

B   Global (Site, User) Config File Reference

This section defines all legal items and values for cylc site and user config files. See *Site And User Config Files* (Section 6) for file locations, intended usage, and how to generate the files using the `cylc get-site-config` command.

As for suite definitions, Jinja2 expressions can be embedded in site and user config files to generate the final result parsed by cylc. Use of Jinja2 in suite definitions is documented in Section 9.7.

B.1   Top Level Items

B.1.1   temporary directory

A temporary directory is needed by a few cylc commands, and is cleaned automatically on exit. Leave unset for the default (usually `$TMPDIR`).

- *type:* string (directory path)
- *default:* (none)
- *example:* `temporary directory = /tmp/$USER/cylc`

B.1.2   process pool size

Number of process pool worker processes used to execute shell commands (job submission, event handlers, job poll and kill commands).

- *type:* integer
- *default:* None (number of processor cores on the suite host)

B.1.3   disable interactive command prompts

Commands that intervene in running suites can be made to ask for confirmation before acting. Some find this annoying and ineffective as a safety measure, however, so command prompts are disabled by default.

- *type:* boolean
- *default:* True

B.1.4   enable run directory housekeeping

The suite run directory tree is created anew with every suite start (not restart) but output from the most recent previous runs can be retained in a rolling archive. Set length to 0 to keep no backups. **This is incompatible with current Rose suite housekeeping** (see Section 13 for more on Rose) so it is disabled by default, in which case new suite run files will overwrite existing ones in the same run directory tree. Rarely, this can result in incorrect polling results due to the presence of old task status files.

- *type:* boolean

- *default:* False

B.1.5   run directory rolling archive length

The number of old run directory trees to retain if run directory housekeeping is enabled.

- *type:* integer
- *default:* 2

B.1.6   task host select command timeout

When a task host in a suite is a shell command string, cylc calls the shell to determine the task host. This call is invoked by the main process, and may cause the suite to hang while waiting for the command to finish. This setting sets a timeout for such a command to ensure that the suite can continue.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default: PT10S*

B.2   [task messaging]

This section contains configuration items that affect task-to-suite communications.

B.2.1   [task messaging] →retry interval

If a send fails, the messaging code will retry after a configured delay interval.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:* PT5S

B.2.2   [task messaging] →maximum number of tries

If successive sends fail, the messaging code will give up after a configured number of tries.

- *type:* integer
- *minimum:* 1
- *default:* 7

B.2.3   [task messaging] →connection timeout

This is the same as the `--comms-timeout` option in cylc commands. Without a timeout remote connections to unresponsive suites can hang indefinitely (suites suspended with Ctrl-Z for instance).

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:* PT30S

B.3   [suite logging]

The suite event log, held under the suite run directory, is maintained as a rolling archive. Logs are rolled over (backed up and started anew) when they reach a configurable limit size.

B.3.1   [suite logging] →roll over at start-up

If True, a new suite log will be started for a new suite run.

- *type:* boolean
- *default:* True

B.3.2   [suite logging] →rolling archive length

How many rolled logs to retain in the archive.

- *type:* integer
- *minimum:* 1
- *default:* 5

B.3.3   [suite logging] →maximum size in bytes

Suite event logs are rolled over when they reach this file size.

- *type:* integer
- *default:* 1000000

B.4   [documentation]

Documentation locations for the `cylc doc` command and gcylc Help menus.

B.4.1   [documentation] →[[files]]

File locations of documentation held locally on the cylc host server.

B.4.1.1   [documentation] →[[files]] →html index

File location of the main cylc documentation index.

- *type:* string
- *default:* `$CYLC_DIR/doc/index.html`

B.4.1.2 [documentation] →[[files]] →pdf user guide

File location of the cylc User Guide, PDF version.

- *type:* string
- *default:* `$CYLC_DIR/doc/cug-pdf.pdf`

B.4.1.3 [documentation] →[[files]] →multi-page html user guide

File location of the cylc User Guide, multi-page HTML version.

- *type:* string
- *default:* `$CYLC_DIR/doc/html/multi/cug-html.html`

B.4.1.4 [documentation] →[[files]] →single-page html user guide

File location of the cylc User Guide, single-page HTML version.

- *type:* string
- *default:* `$CYLC_DIR/doc/html/single/cug-html.html`

B.4.2 [documentation] →[[urls]]

Online documentation URLs.

B.4.2.1 [documentation] →[[urls]] →internet homepage

URL of the cylc internet homepage, with links to documentation for the latest official release.

- *type:* string
- *default:* http://cylc.github.com/cylc/

B.4.2.2 [documentation] →[[urls]] →local index

Local intranet URL of the main cylc documentation index.

- *type:* string
- *default:* (none)

B.5 [document viewers]

PDF and HTML viewers can be launched by cylc to view the documentation.

B.5.1 [document viewers] →pdf

Your preferred PDF viewer program.

- *type:* string

- *default:* evince

B.5.2   [document viewers] →html

Your preferred web browser.

- *type:* string
- *default:* firefox

B.6   [editors]

Choose your favourite text editor for editing suite definitions.

B.6.1   [editors] →terminal

The editor to be invoked by the cylc command line interface.

- *type:* string
- *default:* `vim`
- *examples:*
    - `terminal = emacs -nw` (emacs non-GUI)
    - `terminal = emacs` (emacs GUI)
    - `terminal = gvim -f` (vim GUI)

B.6.2   [editors] →gui

The editor to be invoked by the cylc GUI.

- *type:* string
- *default:* `gvim -f`
- *examples:*
    - `gui = emacs`
    - `gui = xterm -e vim`

B.7   [communication]

This section covers options for network communication between cylc clients (suite-connecting commands and guis) servers (running suites). Each suite listens on a dedicated network port, binding on the first available starting at the configured base port.

By default, the communication method is HTTPS secured with HTTP Digest Authentication. If the system does not support SSL, it will fall back to HTTP.

B.7.1   [communication] →method

The choice of client-server communication method - currently only HTTPS is supported, although others could be developed and plugged in.

- *type:* string
- *options:*
    - **https**
- *default:* https

B.7.2   [communication] →base port

The first port that cylc is allowed to use.

- *type:* integer
- *default:* 43001

B.7.3   [communication] →maximum number of ports

This determines the maximum number of suites that can run at once on the suite host.

- *type:* integer
- *default:* 100

B.7.4   [communication] →proxies on

Enable or disable proxy servers for HTTPS - disabled by default.

- *type:* boolean
- *localhost default:* False

B.7.5   [communication] →options

Option flags for the communication method. Currently only 'SHA1' is supported for HTTPS, which alters HTTP Digest Auth to use the SHA1 hash algorithm rather than the standard MD5. This is more secure but is also less well supported by third party web clients including web browsers. You may need to add the 'SHA1' option if you are running on platforms where MD5 is discouraged (e.g. under FIPS).

- *type:* string_list
- *default:* `[]`
- *options:*
    - **SHA1**

B.8   [monitor]

Configurable settings for the command line `cylc monitor` tool.

B.8.1   [monitor] →sort order

The sort order for tasks in the monitor view.

- *type:* string
- *options:*
  - **alphanumeric**
  - **definition** - the order that tasks appear under `[runtime]` in the suite definition.
- *default:* definition

B.9   [hosts]

The [hosts] section configures some important host-specific settings for the suite host ('local-host') and remote task hosts. Note that *remote task behaviour is determined by the site/user config on the suite host, not on the task host.* Suites can specify task hosts that are not listed here, in which case local settings will be assumed, with the local home directory path, if present, replaced by `$HOME` in items that configure directory locations.

B.9.1   [hosts] →[[HOST]]

The default task host is the suite host, **localhost**, with default values as listed below. Use an explicit `[hosts][[localhost]]` section if you need to override the defaults. Localhost settings are then also used as defaults for other hosts, with the local home directory path replaced as described above. This applies to items omitted from an explicit host section, and to hosts that are not listed at all in the site and user config files. Explicit host sections are only needed if the automatically modified local defaults are not sufficient.

Host section headings can also be *regular expressions* to match multiple hostnames. Note that the general regular expression wildcard is '`.*`' (zero or more of any character), not '`*`'. Hostname matching regular expressions are used as-is in the Python `re.match()` function. As such they match from the beginning of the hostname string (as specified in the suite definition) and they do not have to match through to the end of the string (use the string-end matching character '`$`' in the expression to force this).

A hierachy of host match expressions from specific to general can be used because config items are processed in the order specified in the file.

- *type:* string (hostname or regular expression)
- *examples:*
  - `server1.niwa.co.nz` - explicit host name
  - `server\d.niwa.co.nz` - regular expression

B.9.1.1   [hosts] →[[HOST]] →run directory

The top level of the directory tree that holds suite-specific output logs, run database, etc.

- *type:* string (directory path)
- *default:* `$HOME/cylc-run`

B.9.1.2   [hosts] →[[HOST]] →work directory

The top level for suite work and share directories.

- *type:* string (directory path)
- *localhost default:* `$HOME/cylc-run`

B.9.1.3   [hosts] →[[HOST]] →task communication method

The means by which task progress messages are reported back to the running suite. See above for default polling intervals for the poll method.

- *type:* string (must be one of the following three options)
- *options:*
    - **default** - direct client-server communication via network ports
    - **ssh** - use ssh to re-invoke the messaging commands on the suite server
    - **poll** - the suite polls for the status of tasks (no task messaging)
- *localhost default:* default

B.9.1.4   [hosts] →[[HOST]] →execution polling intervals

Cylc can poll running jobs to catch problems that prevent task messages from being sent back to the suite, such as hard job kills, network outages, or unplanned task host shutdown. Routine polling is done only for the polling *task communication method* (below) unless suite-specific polling is configured in the suite definition. A list of interval values can be specified, with the last value used repeatedly until the task is finished - this allows more frequent polling near the beginning and end of the anticipated task run time. Multipliers can be used as shorthand as in the example below.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:*
- *example:* `execution polling intervals = 5*PT1M, 10*PT5M, 5*PT1M`

B.9.1.5   [hosts] →[[HOST]] →submission polling intervals

Cylc can also poll submitted jobs to catch problems that prevent the submitted job from executing at all, such as deletion from an external batch scheduler queue. Routine polling is done only for the polling *task communication method* (above) unless suite-specific polling is configured in the suite definition. A list of interval values can be specified as for execution polling (above) but a single value is probably sufficient for job submission polling.

- *type:* ISO 8601 duration/interval representation (e.g. `PT10S`, 10 seconds, or `PT1M`, 1 minute).
- *default:*
- *example:* (see the execution polling example above)

B.9.1.6   [hosts] →[[HOST]] →remote copy template

A string for the command used to copy files to a remote host. This is not used on the suite host unless you run local tasks under another user account. The value is assumed to be `scp` with some initial options or a command that implements a similar interface to `scp`.

- *type:* string
- *localhost default:* `scp -oBatchMode=yes -oConnectTimeout=10`

B.9.1.7   [hosts] →[[HOST]] →remote shell template

A string for the command used to invoke commands on this host. This is not used on the suite host unless you run local tasks under another user account. The value is assumed to be `ssh` with some initial options or a command that implements a similar interface to `ssh`.

- *type:* string
- *localhost default:* `ssh -oBatchMode=yes -oConnectTimeout=10`

B.9.1.8   [hosts] →[[HOST]] →use login shell

Whether to use a login shell or not for remote command invocation. By default cylc runs remote ssh commands using a login shell:

```
ssh user@host 'bash --login cylc ...'
```

which will source `/etc/profile` and `~/.profile` to set up the user environment. However, for security reasons some institutions do not allow unattended commands to start login shells, so you can turn off this behaviour to get:

```
ssh user@host 'cylc ...'
```

which will use the default shell on the remote machine, sourcing `~/.bashrc` (or `~/.cshrc`) to set up the environment.

- *type:* boolean
- *localhost default:* True

B.9.1.9   [hosts] →[[HOST]] →cylc executable

The `cylc` executable on a remote host. Note this should point to the cylc multi-version wrapper (see 7.2) on the host, not `bin/cylc` for a specific installed version. Specify a full path if `cylc` is not in `\$PATH` when it is invoked via `ssh` on this host.

- *type:* string
- *localhost default:* `cylc`

B.9.1.10   [hosts] →[[HOST]] →global init-script

If specified, the value of this setting will be inserted to just before the `init-script` section of all job scripts that are to be submitted to the specified remote host.

- *type:* string
- *localhost default:* `""`

B.9.1.11   [hosts] →[[HOST]] →copyable environment variables

A list containing the names of the environment variables that can and/or need to be copied from the suite daemon to a job.

- *type:* string_list
- *localhost default:* `[]`

B.9.1.12   [hosts] →[[HOST]] →retrieve job logs

Global default for the A.4.1.14.3 setting for the specified host.

B.9.1.13   [hosts] →[[HOST]] →retrieve job logs command

If `rsync -a` is unavailable or insufficient to retrieve job logs from a remote host, you can use this setting to specify a suitable command.

- *type:* string
- *default:* rsync -a

B.9.1.14   [hosts] →[[HOST]] →retrieve job logs max size

Global default for the A.4.1.14.4 setting for the specified host.

B.9.1.15   [hosts] →[[HOST]] →retrieve job logs retry delays

Global default for the A.4.1.14.5 setting for the specified host.

B.9.1.16   [hosts] →[[HOST]] →task event handler retry delays

Host specific default for the A.4.1.15.7 setting.

B.9.1.17   [hosts] →[[HOST]] →local tail command template

A template (with `%(filename)s` substitution) for the command used to tail-follow local job logs, used by the gcylc log viewer and `cylc cat-log --tail`. You are unlikely to need to override this.

- *type:* string
- *default:* `tail -n +1 -F %(filename)s`

B.9.1.18 [hosts] →[[HOST]] →remote tail command template

A template (with `%(filename)s` substitution) for the command used to tail-follow remote job logs, used by the gcylc log viewer and `cylc cat-log --tail`. The remote tail command needs to be told to die when its parent process exits. You may need to override this command for task hosts where the default `tail` or `ps` commands are not equivalent to the Gnu Linux versions.

- *type:* string
- *default:* `tail --pid=$(ps h -o ppid $$ | sed -e 's/[[:space:]]//g')-n +1 -F %(filename)s`
- *example:* for AIX hosts:
  `/gnu/tail --pid=$(ps -o ppid= -p $$ | sed -e 's/[[:space:]]//g')-n +1 -F %(filename)s`

B.9.1.19 [hosts] →[[HOST]] →[[[batch systems]]]

Settings for particular batch systems on HOST. In the subsections below, SYSTEM should be replaced with the cylc batch system handler name that represents the batch system (see A.4.1.13.1).

B.9.1.19.1 [hosts] →[[HOST]] →[[[batch systems]]] →[[[[SYSTEM]]]] →err tailer

A command template (with `%(job_id)s` substitution) that can be used to tail-follow the stderr stream of a running job if SYSTEM does not use the normal log file location while the job is running. This setting overrides B.9.1.17 and B.9.1.18 above.

- *type:* string
- *default:* (none)
- *example:* For PBS:
```
[hosts]
    [[ myhpc*]]
        [[[batch systems]]]
            [[[[pbs]]]]
                err tailer = qcat -f -e %(job_id)s
                out tailer = qcat -f -o %(job_id)s
                err viewer = qcat -e %(job_id)s
                out viewer = qcat -o %(job_id)s
```

B.9.1.19.2 [hosts] →[[HOST]] →[[[batch systems]]] →[[[[SYSTEM]]]] →out tailer

A command template (with `%(job_id)s` substitution) that can be used to tail-follow the stdout stream of a running job if SYSTEM does not use the normal log file location while the job is running. This setting overrides B.9.1.17 and B.9.1.18 above.

- *type:* string
- *default:* (none)
- *example:* see B.9.1.19.1

B.9.1.19.3 [hosts] →[[HOST]] →[[[batch systems]]] →[[[[SYSTEM]]]] →err viewer

A command template (with `%(job_id)s` substitution) that can be used to view the stderr stream of a running job if SYSTEM does not use the normal log file location while the job is running.

- *type:* string
- *default:* (none)
- *example:* see B.9.1.19.1

B.9.1.19.4   [hosts] →[[HOST]] →[[[batch systems]]] →[[[[SYSTEM]]]] →out viewer

A command template (with `%(job_id)s` substitution) that can be used to view the stdout stream of a running job if SYSTEM does not use the normal log file location while the job is running.

- *type:* string
- *default:* (none)
- *example:* see B.9.1.19.1

B.9.1.19.5   [hosts] →[[HOST]] →[[[batch systems]]] →[[[[SYSTEM]]]] →job name length maximum

The maximum length for job name acceptable by a batch system on a given host. Currently, this setting is only meaningful for PBS jobs. For example, PBS 12 or older will fail a job submit if the job name has more than 15 characters, which is the default setting. If you have PBS 13 or above, you may want to modify this setting to a larger value.

- *type:* integer
- *default:* (none)
- *example:* For PBS:

```
[hosts]
    [[myhpc*]]
        [[[batch systems]]]
            [[[[pbs]]]]
                # PBS 13
                job name length maximum = 236
```


B.9.1.19.6   [hosts] →[[HOST]] →[[[batch systems]]] →[[[[SYSTEM]]]] →execution time limit polling intervals

The intervals between polling after a task job (submitted to the relevant batch system on the relevant host) exceeds its execution time limit. The default setting is PT1M, PT2M, PT7M. The accumulated times (in minutes) for these intervals will be roughly 1, $1 + 2 = 3$ and $1 + 2 + 7 = 10$ after a task job exceeds its execution time limit.

- *type:* Comma-separated list of ISO 8601 duration/interval representations, optionally *preceded* by multipliers.
- *default:* PT1M, PT2M, PT7M
- *example:*

```
[hosts]
    [[myhpc*]]
        [[[batch systems]]]
            [[[[pbs]]]]
                execution time limit polling intervals = 5*PT2M
```

B.10   [suite host self-identification]

The suite host's identity must be determined locally by cylc and passed to running tasks (via `$CYLC_SUITE_HOST`) so that task messages can target the right suite on the right host.

B.10.1   [suite host self-identification] →method

This item determines how cylc finds the identity of the suite host. For the default *name* method cylc asks the suite host for its host name. This should resolve on remote task hosts to the IP address of the suite host; if it doesn't, adjust network settings or use one of the other methods. For the *address* method, cylc attempts to use a special external "target address" to determine the IP address of the suite host as seen by remote task hosts (in-source documentation in `$CYLC_DIR/lib/cylc/suite_host.py` explains how this works). And finally, as a last resort, you can choose the *hardwired* method and manually specify the host name or IP address of the suite host.

- *type:* string
- *options:*
    - name - self-identified host name
    - address - automatically determined IP address (requires *target*, below)
    - hardwired - manually specified host name or IP address (requires *host*, below)
- *default:* name

B.10.2   [suite host self-identification] →target

This item is required for the *address* self-identification method. If your suite host sees the internet, a common address such as `google.com` will do; otherwise choose a host visible on your intranet.

- *type:* string (an inter- or intranet URL visible from the suite host)
- *default:* `google.com`

B.10.3   [suite host self-identification] →host

Use this item to explicitly set the name or IP address of the suite host if you have to use the *hardwired* self-identification method.

- *type:* string (host name or IP address)
- *default:* (none)

B.11   [suite host scanning]

Utilities such as `cylc gsummary` need to scan hosts for running suites.

B.11.1   [suite host scanning] →hosts

A list of hosts to scan for running suites.

- *type:* comma-separated list of host names or IP addresses.
- *default:* localhost

B.12   [task events]

Global site/user defaults for A.4.1.15.

B.13   [test battery]

Settings for the automated development tests.

B.13.1   [test battery] →remote host with shared fs

The name of a remote host with shared HOME file system as the host running the test battery.

B.13.2   [test battery] →remote host

The name of a remote host without shared HOME file system as the host running the test battery.

B.13.3   [test battery] →[[batch systems]]

Settings for testing supported batch systems (job submission methods). The tests for a batch system are only performed if the batch system is available on the test host or a remote host accessible via SSH from the test host.

B.13.3.1   [test battery] →[[batch systems]] →[[[SYSTEM]]]

SYSTEM is the name of a supported batch system with automated tests. This can currently be "loadleveler", "lsf", "pbs", "sge" and/or "slurm".

B.13.3.1.1   [test battery] →[[batch systems]] →[[[SYSTEM]]] →host

The name of a host where commands for this batch system is available. Use "localhost" if the batch system is available on the host running the test battery. Any specified remote host should be accessible via SSH from the host running the test battery.

B.13.3.1.2   [test battery] →[[batch systems]] →[[[SYSTEM]]] →err viewer

The command template (with `\%(job_id)s` substitution) for testing the run time stderr viewer functionality for this batch system.

B.13.3.1.3   [test battery] →[[batch systems]] →[[[SYSTEM]]] →out viewer

The command template (with `\%(job_id)s` substitution) for testing the run time stdout viewer functionality for this batch system.

B.13.3.1.4   [test battery] →[[batch systems]] →[[[SYSTEM]]] →[[[[directives]]]]

The minimum set of directives that must be supplied to the batch system on the site to initiate jobs for the tests.

B.14   [cylc]

Default values for entries in the suite.rc [cylc] section.

B.14.1   [cylc] →UTC mode

Allows you to set a default value for UTC mode in a suite at the site level. See A.2.2 for details.

B.14.2   [cylc] →health check interval

Site default suite health check interval. See A.2.7 for details.

B.14.3   [cylc] →task event mail interval

Site default task event mail interval. See A.2.8 for details.

B.14.4   [cylc] →[[events]]

You can define site defaults for each of the following options, details of which can be found under A.2.13:

B.14.4.0.5   [cylc] →[[events]] →**handlers**

B.14.4.0.6   [cylc] →[[events]] →**handler events**

B.14.4.0.7   [cylc] →[[events]] →**startup handler**

B.14.4.0.8   [cylc] →[[events]] →**shutdown handler**

B.14.4.0.9   [cylc] →[[events]] →**mail events**

B.14.4.0.10   [cylc] →[[events]] →**mail footer**

B.14.4.0.11   [cylc] →[[events]] →**mail from**

B.14.4.0.12   [cylc] →[[events]] →**mail smtp**

B.14.4.0.13   [cylc] →[[events]] →**mail to**

B.14.4.0.14   [cylc] →[[events]] →**timeout handler**

B.14.4.0.15   [cylc] →[[events]] →**timeout**

B.14.4.0.16   [cylc] →[[events]] →**abort on timeout**

B.14.4.0.17   [cylc] →[[events]] →**stalled handler**

B.14.4.0.18   [cylc] →[[events]] →**abort on stalled**

B.14.4.0.19   [cylc] →[[events]] →**inactivity handler**

B.14.4.0.20   [cylc] →[[events]] →**inactivity**

B.14.4.0.21   [cylc] →[[events]] →**abort on inactivity**

B.15   [authentication]

Authentication of client programs with suite daemons can be configured here, and overridden in suites if necessary (see A.2.16).

The suite-specific passphrase must be installed on a user's account to authorize full control privileges (see 7.5 and 12.6). In the future we plan to move to a more traditional user account model so that each authorized user can have their own password.

B.15.1   [authentication] →public

This sets the client privilege level for public access - i.e. no suite passphrase required.

- *type:* string (must be one of the following options)
- *options:*

- – *identity* - only suite and owner names revealed
- – *description* - identity plus suite title and description
- – *state-totals* - identity, description, and task state totals
- – *full-read* - full read-only access for monitor and GUI
- – *shutdown* - full read access plus shutdown, but no other control.
- *default:* state-totals

C Gcylc Config File Reference

This section defines all legal items and values for the gcylc user config file, which should be located in `$HOME/.cylc/gcylc.rc`. Current settings can be printed with the `cylc get-gui-config` command.

C.1 Top Level Items

C.1.1 dot icon size

Set the size of the task state dot icons displayed in the text and dot views.

- *type:* string
- *legal values:* "small" (10px), "medium" (14px), "large" (20px), "extra large (30px)"
- *default:* "medium"

C.1.2 initial side-by-side views

Set the suite view panels initial orientation when the GUI starts. This can be changed later using the "View" menu "Toggle views side-by-side" option.

- *type:* boolean (False or True)
- *default:* "False"

C.1.3 initial views

Set the suite view panel(s) displayed initially, when the GUI starts. This can be changed later using the tool bar.

- *type:* string (a list of one or two view names)
- *legal values:* "text", "dot", "graph"
- *default:* "text"
- *example:* `initial views = graph, dot`

C.1.4 sort by definition order

If this is not turned off the default sort order for task names and families in the dot and text views will the order they appear in the suite definition. Clicking on the task name column in the treeview will toggle to alphanumeric sort, and a View menu item does the same for the dot view. If turned off, the default sort order is alphanumeric and definition order is not available at all.

- *type:* boolean
- *default:* True

C.1.5   sort column

If "text" is in `initial views` then `sort column` sets the column that will be sorted initially when the GUI launches. Sorting can be changed later by clicking on the column headers.

- *type:* string
- *legal values:* "task", "state", "host", "job system", "job ID", "T-submit", "T-start", "T-finish", "dT-mean", "latest message", "none"
- *default:* "none"
- *example:* `sort column = T-start`

C.1.6   sort column ascending

For use in combination with `sort column`, sets whether the column will be sorted using ascending or descending order.

- *type:* boolean
- *default:* "True"
- *example:* `sort column ascending = False`

C.1.7   task filter highlight color

The color used to highlight active task filters in gcylc. It must be a name from the X11 rgb.txt file, e.g. `SteelBlue`; or a *quoted* hexadecimal color code, e.g. `"#ff0000"` for red (quotes are required to prevent the hex code being interpreted as a comment).

- *type:* string
- *default:* `PowderBlue`

C.1.8   task states to filter out

Set the initial filtering options when the GUI starts. Later this can be changed by using the "View" menu "Task Filtering" option.

- *type:* string list
- *legal values:* waiting, held, queued, ready, expired, submitted, submit-failed, submit-retrying, running, succeeded, failed, retrying, runahead
- *default:* runahead

C.1.9   transpose dot

Transposes the content in dot view so that it displays from left to right rather than from top to bottom. Can be changed later using the options submenu available via the view menu.

- *type:* boolean
- *default:* "False"
- *example:* `transpose dot = True`

C.1.10 transpose graph

Transposes the content in graph view so that it displays from left to right rather than from top to bottom. Can be changed later using the options submenu via the view menu.

- *type:* boolean
- *default:* "False"
- *example:* `transpose graph = True`

C.1.11 ungrouped views

List suite views, if any, that should be displayed initially in an ungrouped state. Namespace family grouping can be changed later using the tool bar.

- *type:* string (a list of zero or more view names)
- *legal values:* "text", "dot", "graph"
- *default:* (none)
- *example:* `ungrouped views = text, dot`

C.1.12 use theme

Set the task state color theme, common to all views, to use initially. The color theme can be changed later using the tool bar. See `gcylc.rc.eg` and `themes.rc` in `$CYLC_DIR/conf/gcylcrc/` for how to modify existing color themes or define your own. Use `cylc get-gui-config` to list your available themes.

- *type:* string (theme name)
- *legal values:* "default", "solid", "high-contrast", "color-blind", and any custom or user-modified themes.
- *default:* "default"

C.1.13 window size

Sets the size (in pixels) of the cylc GUI at startup.

- *type:* integer list (x, y)
- *legal values:* positive integers
- *default:* 800, 500
- *example:* `window size = 1000, 700`

C.2 [themes]

This section may contain task state color theme definitions.

C.2.1 [themes] →[[THEME]]

The name of the task state color-theme to be defined in this section.

- *type:* string

C.2.1.1   [themes] →[[THEME]] →inherit

You can inherit from another theme in order to avoid defining all states.

- *type:* string (parent theme name)
- *default:* "default"

C.2.1.2   [themes] →[[THEME]] →defaults

Set default icon attributes for all state icons in this theme.

- *type:* string list (icon attributes)
- *legal values:* `"color=COLOR"`, `"style=STYLE"`, `"fontcolor=FONTCOLOR"`
- *default:* (none)

For the attribute values, COLOR and FONTCOLOR can be color names from the X11 rgb.txt file, e.g. `SteelBlue`; or hexadecimal color codes, e.g. `#ff0000` for red; and STYLE can be "filled" or "unfilled". See `gcylc.rc.eg` and `themes.rc` in `$CYLC_DIR/conf/gcylcrc/` for examples.

C.2.1.3   [themes] →[[THEME]] →STATE

Set icon attributes for all task states in THEME, or for a subset of them if you have used theme inheritance and/or defaults. Legal values of STATE are any of the cylc task proxy states: *waiting, runahead, held, queued, ready, submitted, submit-failed, running, succeeded, failed, retrying, submit-retrying.*

- *type:* string list (icon attributes)
- *legal values:* `"color=COLOR"`, `"style=STYLE"`, `"fontcolor=FONTCOLOR"`
- *default:* (none)

For the attribute values, COLOR and FONTCOLOR can be color names from the X11 rgb.txt file, e.g. `SteelBlue`; or hexadecimal color codes, e.g. `#ff0000` for red; and STYLE can be "filled" or "unfilled". See `gcylc.rc.eg` and `themes.rc` in `$CYLC_DIR/conf/gcylcrc/` for examples.

D Cylc Gscan Config File Reference

This section defines all legal items and values for the gscan config file which should be located in `$HOME/.cylc/gscan.rc`.

D.1 Top Level Items

D.1.1 activate on startup

Set whether `cylc gpanel` will activate automatically when the gui is loaded or not.

- *type:* boolean (True or False)
- *legal values:* "True", "False"
- *default:* "False"
- *example:* `activate on startup = True`

D.1.2 columns

Set the data fields displayed initially when the `cylc gscan` GUI starts. This can be changed later using the right click context menu.
Note that the order in which the fields are specified does not affect the order in which they are displayed.

- *type:* string (a list of one or more view names)
- *legal values:* "host", "owner", "status", "suite", "title", "updated"
- *default:* "status", "suite"
- *example:* `columns = suite, title, status`

E   Command Reference

Cylc ("silk") is a suite engine and metascheduler that specializes in cycling weather and climate forecasting suites and related processing (but it can also be used for one-off workflows of non-cycling tasks). For detailed documentation see the Cylc User Guide (cylc doc --help).

Version 7.3.0

The graphical user interface for cylc is "gcylc" (a.k.a. "cylc gui").

```
USAGE:
  % cylc -v,--version                 # print cylc version
  % cylc version                      # (ditto, by command)
  % cylc help,--help,-h,?             # print this help page

  % cylc help CATEGORY                # print help by category
  % cylc CATEGORY help                # (ditto)

  % cylc help [CATEGORY] COMMAND      # print command help
  % cylc [CATEGORY] COMMAND help,--help # (ditto)

  % cylc [CATEGORY] COMMAND [options] SUITE [arguments]
  % cylc [CATEGORY] COMMAND [options] SUITE TASK [arguments]

Commands and categories can both be abbreviated. Use of categories is
optional, but they organize help and disambiguate abbreviated commands:
  % cylc control trigger SUITE TASK   # trigger TASK in SUITE
  % cylc trigger SUITE TASK           # ditto
  % cylc con trig SUITE TASK          # ditto
  % cylc c t SUITE TASK               # ditto

TASK IDENTIFICATION IN CYLC SUITES
  Tasks are identified by NAME.CYCLE_POINT where POINT is either a
  date-time or an integer.
  Date-time cycle points are in an ISO 8601 date-time format, typically
  CCYYMMDDThhmm followed by a time zone - e.g. 20101225T0600Z.
  Integer cycle points (including those for one-off suites) are integers
  - just '1' for one-off suites.

HOW TO DRILL DOWN TO COMMAND USAGE HELP:
  % cylc help          # list all available categories (this page)
  % cylc help prep     # list commands in category 'preparation'
  % cylc help prep edit # command usage help for 'cylc [prep] edit'

Command CATEGORIES:
  all .......... The complete command set.
  preparation ... Suite editing, validation, visualization, etc.
  information ... Interrogate suite definitions and running suites.
  discovery ..... Detect running suites.
  control ....... Suite start up, monitoring, and control.
  utility ....... Cycle arithmetic and templating, etc.
  task .......... The task messaging interface.
  hook .......... Suite and task event hook scripts.
  admin ........ Cylc installation, testing, and example suites.
  license|GPL ... Software licensing information (GPL v3.0).
```

E.1   Command Categories

E.1.1   admin

CATEGORY: admin - Cylc installation, testing, and example suites.

HELP: cylc [admin] COMMAND help,--help
  You can abbreviate admin and COMMAND.
  The category admin may be omitted.

COMMANDS:

```
    check-software .... Check required software is installed.
    import-examples ... Import example suites your suite run directory
    profile-battery ... Run a battery of profiling tests
    test-battery ...... Run a battery of self-diagnosing test suites
    upgrade-run-dir ... Upgrade a pre-cylc-6 suite run directory
```

E.1.2   all

CATEGORY: all - The complete command set.

HELP: cylc [all] COMMAND help,--help
    You can abbreviate all and COMMAND.
    The category all may be omitted.

COMMANDS:
```
    5to6 ....................................... Improve the cylc 6
        compatibility of a cylc 5 suite file
    broadcast|bcast ............................ Change suite [runtime]
        settings on the fly
    cat-log|log ................................ Print various suite and task
        log files
    cat-state .................................. Print the state of tasks from
        the state dump
    check-software ............................. Check required software is
        installed.
    check-triggering ........................... A suite shutdown event hook
        for cylc testing
    check-versions ............................. Compare cylc versions on task
        host accounts
    checkpoint ................................. Tell suite to checkpoint its
        current state
    conditions ................................. Print the GNU General Public
        License v3.0
    cycle-point|cyclepoint|datetime|cycletime ..... Cycle point arithmetic and
        filename templating
    diff|compare ............................... Compare two suite definitions
        and print differences
    documentation|browse ....................... Display cylc documentation (
        User Guide etc.)
    dump ....................................... Print the state of tasks in a
        running suite
    edit ....................................... Edit suite definitions,
        optionally inlined
    email-suite ................................ A suite event hook script
        that sends email alerts
    email-task ................................. A task event hook script that
        sends email alerts
    ext-trigger|external-trigger ............... Report an external trigger
        event to a suite
    get-directory .............................. Retrieve suite source
        directory paths
    get-gui-config ............................. Print gcylc configuration
        items
    get-site-config|get-global-config ............ Print site/user configuration
        items
    get-suite-config|get-config ................ Print suite configuration
        items
    get-suite-contact|get-contact|print-contact ... Print the contact information
        of a suite daemon
    get-suite-version|get-cylc-version ........... Print the cylc version of a
        suite daemon
    gpanel ..................................... Internal interface for GNOME
        2 panel applet
    graph ...................................... Plot suite dependency graphs
        and runtime hierarchies
    graph-diff ................................. Compare two suite
        dependencies or runtime hierarchies
    gscan|gsummary ............................. Scan GUI for monitoring
        multiple suites
    gui ........................................ (a.k.a. gcylc) cylc GUI for
        suite control etc.
```

```
hold ........................................ Hold (pause) suites or
    individual tasks
import-examples ............................. Import example suites your
    suite run directory
insert ...................................... Insert tasks into a running
    suite
job-logs-retrieve ........................... (Internal) Retrieve logs from
    a remote host for a task job
job-submit .................................. (Internal) Submit a job
jobs-kill ................................... (Internal) Kill task jobs
jobs-poll ................................... (Internal) Retrieve status
    for task jobs
jobs-submit ................................. (Internal) Submit task jobs
jobscript ................................... Generate a task job script
    and print it to stdout
kill ........................................ Kill submitted or running
    tasks
list|ls ..................................... List suite tasks and family
    namespaces
ls-checkpoints .............................. Display task pool etc at
    given events
message|task-message ........................ (task messaging) Report task
    messages
monitor ..................................... An in-terminal suite monitor
    (see also gcylc)
nudge ....................................... Cause the cylc task
    processing loop to be invoked
ping ........................................ Check that a suite is running
poll ........................................ Poll submitted or running
    tasks
print ....................................... Print registered suites
profile-battery ............................. Run a battery of profiling
    tests
register .................................... Register a suite for use
release|unhold .............................. Release (unpause) suites or
    individual tasks
reload ...................................... Reload the suite definition
    at run time
remove ...................................... Remove tasks from a running
    suite
reset ....................................... Force one or more tasks to
    change state.
restart ..................................... Restart a suite from a
    previous state
run|start ................................... Start a suite at a given
    cycle point
scan ........................................ Scan a host for running
    suites
scp-transfer ................................ Scp-based file transfer for
    cylc suites
search|grep ................................. Search in suite definitions
set-runahead ................................ Change the runahead limit in
    a running suite.
set-verbosity ............................... Change a running suite's
    logging verbosity
show ........................................ Print task state (
    prerequisites and outputs etc.)
spawn ....................................... Force one or more tasks to
    spawn their successors.
stop|shutdown ............................... Shut down running suites
submit|single ............................... Run a single task just as its
    parent suite would
suite-state ................................. Query the task states in a
    suite
test-battery ................................ Run a battery of self-
    diagnosing test suites
trigger ..................................... Manually trigger or re-
    trigger a task
upgrade-run-dir ............................. Upgrade a pre-cylc-6 suite
    run directory
validate .................................... Parse and validate suite
    definitions
```

```
version ........................................ Print the cylc release
    version
view .......................................... View suite definitions,
    inlined and Jinja2 processed
warranty ...................................... Print the GPLv3 disclaimer of
    warranty
```

E.1.3   control

```
CATEGORY: control - Suite start up, monitoring, and control.

HELP: cylc [control] COMMAND help,--help
    You can abbreviate control and COMMAND.
    The category control may be omitted.

COMMANDS:
    broadcast|bcast ................ Change suite [runtime] settings on the fly
    checkpoint ..................... Tell suite to checkpoint its current state
    ext-trigger|external-trigger ... Report an external trigger event to a suite
    gui ............................ (a.k.a. gcylc) cylc GUI for suite control
        etc.
    hold ........................... Hold (pause) suites or individual tasks
    insert ......................... Insert tasks into a running suite
    kill ........................... Kill submitted or running tasks
    nudge .......................... Cause the cylc task processing loop to be
        invoked
    poll ........................... Poll submitted or running tasks
    release|unhold ................. Release (unpause) suites or individual tasks
    reload ......................... Reload the suite definition at run time
    remove ......................... Remove tasks from a running suite
    reset .......................... Force one or more tasks to change state.
    restart ........................ Restart a suite from a previous state
    run|start ...................... Start a suite at a given cycle point
    set-runahead ................... Change the runahead limit in a running suite
        .
    set-verbosity .................. Change a running suite's logging verbosity
    spawn .......................... Force one or more tasks to spawn their
        successors.
    stop|shutdown .................. Shut down running suites
    trigger ........................ Manually trigger or re-trigger a task
```

E.1.4   discovery

```
CATEGORY: discovery - Detect running suites.

HELP: cylc [discovery] COMMAND help,--help
    You can abbreviate discovery and COMMAND.
    The category discovery may be omitted.

COMMANDS:
    check-versions ... Compare cylc versions on task host accounts
    ping ............. Check that a suite is running
    scan ............. Scan a host for running suites
```

E.1.5   hook

```
CATEGORY: hook - Suite and task event hook scripts.

HELP: cylc [hook] COMMAND help,--help
    You can abbreviate hook and COMMAND.
    The category hook may be omitted.

COMMANDS:
    check-triggering .... A suite shutdown event hook for cylc testing
    email-suite ......... A suite event hook script that sends email alerts
    email-task .......... A task event hook script that sends email alerts
    job-logs-retrieve ... (Internal) Retrieve logs from a remote host for a task
        job
```

E.1.6   information

CATEGORY: information - Interrogate suite definitions and running suites.

HELP: cylc [information] COMMAND help,--help
   You can abbreviate information and COMMAND.
   The category information may be omitted.

COMMANDS:
   cat-log|log .... Print various suite and task
       log files
   cat-state .... Print the state of tasks from
        the state dump
   documentation|browse .... Display cylc documentation (
       User Guide etc.)
   dump .... Print the state of tasks in a
        running suite
   get-gui-config .... Print gcylc configuration
       items
   get-site-config|get-global-config .... Print site/user configuration
        items
   get-suite-config|get-config .... Print suite configuration
       items
   get-suite-contact|get-contact|print-contact ... Print the contact information
       of a suite daemon
   get-suite-version|get-cylc-version .... Print the cylc version of a
       suite daemon
   gpanel .... Internal interface for GNOME
       2 panel applet
   gscan|gsummary .... Scan GUI for monitoring
       multiple suites
   gui|gcylc .... (a.k.a. gcylc) cylc GUI for
       suite control etc.
   list|ls .... List suite tasks and family
       namespaces
   monitor .... An in-terminal suite monitor
       (see also gcylc)
   show .... Print task state (
       prerequisites and outputs etc.)
   version .... Print the cylc release
       version

E.1.7   license

CATEGORY: license|GPL - Software licensing information (GPL v3.0).

HELP: cylc [license|GPL] COMMAND help,--help
   You can abbreviate license|GPL and COMMAND.
   The category license|GPL may be omitted.

COMMANDS:
   conditions ... Print the GNU General Public License v3.0
   warranty ..... Print the GPLv3 disclaimer of warranty

E.1.8   preparation

CATEGORY: preparation - Suite editing, validation, visualization, etc.

HELP: cylc [preparation] COMMAND help,--help
   You can abbreviate preparation and COMMAND.
   The category preparation may be omitted.

COMMANDS:
   5to6 .... Improve the cylc 6 compatibility of a cylc 5 suite file
   diff|compare .... Compare two suite definitions and print differences
   edit .... Edit suite definitions, optionally inlined
   get-directory ... Retrieve suite source directory paths
   graph .... Plot suite dependency graphs and runtime hierarchies
   graph-diff ...... Compare two suite dependencies or runtime hierarchies
   jobscript .... Generate a task job script and print it to stdout

```
    list|ls ......... List suite tasks and family namespaces
    print ........... Print registered suites
    register ....... Register a suite for use
    search|grep ..... Search in suite definitions
    validate ....... Parse and validate suite definitions
    view ........... View suite definitions, inlined and Jinja2 processed
```

E.1.9   task

CATEGORY: task - The task messaging interface.

HELP: cylc [task] COMMAND help,--help
    You can abbreviate task and COMMAND.
    The category task may be omitted.

COMMANDS:
```
    job-submit ............. (Internal) Submit a job
    jobs-kill .............. (Internal) Kill task jobs
    jobs-poll .............. (Internal) Retrieve status for task jobs
    jobs-submit ............ (Internal) Submit task jobs
    message|task-message ... (task messaging) Report task messages
    submit|single ......... Run a single task just as its parent suite would
```

E.1.10   utility

CATEGORY: utility - Cycle arithmetic and templating, etc.

HELP: cylc [utility] COMMAND help,--help
    You can abbreviate utility and COMMAND.
    The category utility may be omitted.

COMMANDS:
```
    cycle-point|cyclepoint|datetime|cycletime ... Cycle point arithmetic and
        filename templating
    ls-checkpoints .............................. Display task pool etc at given
        events
    scp-transfer ................................ Scp-based file transfer for
        cylc suites
    suite-state ................................. Query the task states in a
        suite
```

E.2   Commands

E.2.1   5to6

Usage: cylc [prep] 5to6 FILE

Suggest changes to a cylc 5 suite file to make it more cylc 6 compatible.
This may be a suite.rc file, an include file, or a suite.rc.processed file.

By default, print the changed file to stdout. Lines that have been changed
are marked with '# UPGRADE'. These marker comments are purely for your own
information and should not be included in any changes you make. In
particular, they may break continuation lines.

Lines with '# UPGRADE CHANGE' have been altered.
Lines with '# UPGRADE ... INFO' indicate that manual change is needed.

As of cylc 7, 'cylc validate' will no longer print out automatic dependency
section translations. At cylc 6 versions of cylc, 'cylc validate' will show
start-up/mixed async replacement R1* section(s). The validity of these can
be highly dependent on the initial cycle point choice (e.g. whether it is
T00 or T12).

This command works best for hour-based cycling - it will always convert
e.g. 'foo[T-6]' to 'foo[-PT6H]', even where this is in a monthly or yearly
cycling section graph.

This command is an aid, and is not an auto-upgrader or a substitute for
reading the documentation. The suggested changes must be understood and
checked by hand.

Example usage:

```
# Print out a file path (FILE) with suggested changes to stdout.
cylc 5to6 FILE

# Replace the file with the suggested changes file.
cylc 5to6 FILE > FILE

# Save a copy of the changed file.
cylc 5to6 FILE > FILE.5to6

# Show the diff of the changed file vs the original file.
diff - <(cylc 5to6 FILE) <FILE
```

Options:
  -h, --help   Print this help message and exit.

E.2.2   broadcast

Usage: cylc [control] broadcast|bcast [OPTIONS] REG

Override [runtime] config in targeted namespaces in a running suite.

Uses for broadcast include making temporary changes to task behaviour,
and task-to-downstream-task communication via environment variables.

A broadcast can target any [runtime] namespace for all cycles or for a
specific cycle. If a task is affected by specific-cycle and all-cycle
broadcasts at once, the specific takes precedence. If a task is affected
by broadcasts to multiple ancestor namespaces, the result is determined
by normal [runtime] inheritance. In other words, it follows this order:

all:root -> all:FAM -> all:task -> tag:root -> tag:FAM -> tag:task

Broadcasts persist, even across suite restarts, until they expire when
their target cycle point is older than the oldest current in the suite,
or until they are explicitly cancelled with this command. All-cycle
broadcasts do not expire.

For each task the final effect of all broadcasts to all namespaces is
computed on the fly just prior to job submission. The --cancel and
--clear options simply cancel (remove) active broadcasts, they do not
act directly on the final task-level result. Consequently, for example,
you cannot broadcast to "all cycles except Tn" with an all-cycle
broadcast followed by a cancel to Tn (there is no direct broadcast to Tn
to cancel); and you cannot broadcast to "all members of FAMILY except
member_n" with a general broadcast to FAMILY followed by a cancel to
member_n (there is no direct broadcast to member_n to cancel).

To broadcast a variable to all tasks (quote items with internal spaces):
  % cylc broadcast -s "[environment]VERSE = the quick brown fox" REG
To do the same with a file:
  % cat >'broadcast.rc' <<'__RC__'
  % [environment]
  %   VERSE = the quick brown fox
  % __RC__
  % cylc broadcast -F 'broadcast.rc' REG
To cancel the same broadcast:
  % cylc broadcast --cancel "[environment]VERSE" REG
If -F FILE was used, the same file can be used to cancel the broadcast:
  % cylc broadcast -G 'broadcast.rc' REG

Use -d/--display to see active broadcasts. Multiple --cancel options or
multiple --set and --set-file options can be used on the same command line.
Multiple --set and --set-file options are cumulative.

The --set-file=FILE option can be used when broadcasting multiple values, or

when the value contains newline or other metacharacters. If FILE is "-", read
from standard input.

Broadcast cannot change [runtime] inheritance.

See also 'cylc reload' - reload a modified suite definition at run time.

Arguments:
    REG   Suite name

Options:
  -h, --help   show this help message and exit
  -p CYCLE_POINT, --point=CYCLE_POINT
                        Target cycle point. More than one can be added.
                        Defaults to '*' with --set and --cancel, and nothing
                        with --clear.
  -n NAME, --namespace=NAME
                        Target namespace. Defaults to 'root' with --set and
                        --cancel, and nothing with --clear.
  -s [SEC]ITEM=VALUE, --set=[SEC]ITEM=VALUE
                        A [runtime] config item and value to broadcast.
  -F FILE, --set-file=FILE, --file=FILE
                        File with config to broadcast. Can be used multiple
                        times.
  -c [SEC]ITEM, --cancel=[SEC]ITEM
                        An item-specific broadcast to cancel.
  -G FILE, --cancel-file=FILE
                        File with broadcasts to cancel. Can be used multiple
                        times.
  -C, --clear   Cancel all broadcasts, or with -p/--point,
                        -n/--namespace, cancel all broadcasts to targeted
                        namespaces and/or cycle points. Use "-C -p '*'" to
                        cancel all all-cycle broadcasts without canceling all
                        specific-cycle broadcasts.
  -e CYCLE_POINT, --expire=CYCLE_POINT
                        Cancel any broadcasts that target cycle points earlier
                        than, but not inclusive of, CYCLE_POINT.
  -d, --display   Display active broadcasts.
  -k TASKID, --display-task=TASKID
                        Print active broadcasts for a given task
                        (NAME.CYCLE_POINT).
  -b, --box   Use unicode box characters with -d, -k.
  -r, --raw   With -d/--display or -k/--display-task, write out the
                        broadcast config structure in raw Python form.
  --user=USER   Other user account name. This results in command
                        reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                        on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                        tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                        retrieved automatically if non-interactive ssh is
                        configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                        The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                        Set a timeout for network connections to the running
                        suite. The default is no timeout. For task messaging
                        connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
  -f, --force   Do not ask for confirmation before acting. Note that
                        it is not necessary to use this option if interactive
                        command prompts have been disabled in the site/user
                        config files.

E.2.3   cat-log

`Usage: cylc [info] cat-log|log [OPTIONS] REG [TASK-ID]`

Print the location or content of any suite or task log file, or a listing of a
task log directory on the suite or task host. By default the suite event log
or task job script is printed. For task logs you must use the same cycle
point format as the suite (list the log directory to see what it is).

Arguments:
```
   REG                      Suite name
   [TASK-ID]                Task ID
```

Options:
```
  -h, --help               show this help message and exit
  -l, --location           Print location of the log file, exit 0 if it exists,
                           exit 1 otherwise
  -o, --stdout             Suite log: out, task job log: job.out
  -e, --stderr             Suite log: err, task job log: job.err
  -r INT, --rotation=INT
                           Suite logs log rotation number
  -a, --activity           Task job log only: Short for --filename=job-
                           activity.log
  -d, --diff               Task job log only: Short for --filename=job-edit.diff
                           (file present after an edit-run).
  -u, --status             Task job log only: Short for --filename=job.status
  -f FILENAME, -c FILENAME, --filename=FILENAME, --custom=FILENAME
                           Name of log file (e.g. 'job.stats').
  --tail                   Tail the job log, if the task is running.
  -s INT, -t INT, --submit-number=INT, --try-number=INT
                           Task job log only: submit number (default=NN).
  -x, --list-local         List a log directory on the suite host
  -y, --list-remote        Task job log only: List log directory on the job host
  --user=USER              Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST              Other host name. This results in command reinvocation
                           on the remote account.
  -v, --verbose            Verbose output mode.
  --debug                  Run suites in non-daemon mode, and show exception
                           tracebacks.
```

E.2.4   cat-state

`Usage: cylc [info] cat-state [OPTIONS] REG`

Print the suite state in the old state dump file format to stdout.
This command is deprecated; use "cylc ls-checkpoints" instead.

Arguments:
```
   REG                   Suite name
```

Options:
```
  -h, --help       show this help message and exit
  -d, --dump       Use the same display format as the 'cylc dump' command.
  --user=USER      Other user account name. This results in command reinvocation
                   on the remote account.
  --host=HOST      Other host name. This results in command reinvocation on the
                   remote account.
  -v, --verbose    Verbose output mode.
  --debug          Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.5   check-software

`Usage: cylc [admin] check-software`

Check that the external software required by cylc is installed.

Minimum versions are not checked, except in the case of Python.

Options:

    -h, --help   Print this help message and exit.

E.2.6   check-triggering

```
cylc [hook] check-triggering ARGS
```

This is a cylc shutdown event handler that compares the newly generated
suite log with a previously generated reference log "reference.log"
stored in the suite definition directory. Currently it just compares
runtime triggering information, disregarding event order and timing, and
fails the suite if there is any difference. This should be sufficient to
verify correct scheduling of any suite that is not affected by different
run-to-run conditional triggering.

1) run your suite with "cylc run --generate-reference-log" to generate
the reference log with resolved triggering information. Check manually
that the reference run was correct.
2) run reference tests with "cylc run --reference-test" - this
automatically sets the shutdown event handler along with a suite timeout
and "abort if shutdown handler fails", "abort on timeout", and "abort if
any task fails".

Reference tests can use any run mode:
 * simulation mode - tests that scheduling is equivalent to the reference
 * dummy mode - also tests that task hosting, job submission, job script
   evaluation, and cylc messaging are not broken.
 * live mode - tests everything (but takes longer with real tasks!)

 If any task fails, or if cylc itself fails, or if triggering is not
 equivalent to the reference run, the test will abort with non-zero exit
 status - so reference tests can be used as automated tests to check
 that changes to cylc have not broken your suites.

E.2.7   check-versions

Usage: cylc [discovery] check-versions [OPTIONS] SUITE

Check the version of cylc invoked on each of SUITE's task host accounts when
CYLC_VERSION is set to *the version running this command line tool*.
Different versions are reported but are not considered an error unless the
-e|--error option is specified, because different cylc versions from 6.0.0
onward should at least be backward compatible.

It is recommended that cylc versions be installed in parallel and access
configured via the cylc version wrapper as described in the cylc INSTALL
file and User Guide. This must be done on suite and task hosts. Users then get
the latest installed version by default, or (like tasks) a particular version
if $CYLC_VERSION is defined.

User -v/--verbose to see the command invoked to determine the remote version
(all remote cylc command invocations will be of the same form, which may be
site dependent -- see cylc global config documentation.

Arguments:
    SUITE   Suite name or path

Options:
  -h, --help   show this help message and exit
  -e, --error   Exit with error status if 7.3.0 is not available on
                          all remote accounts.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --suite-owner=OWNER   Specify suite owner
  -s NAME=VALUE, --set=NAME=VALUE
                          Set the value of a Jinja2 template variable in the
                          suite definition. This option can be used multiple
                          times on the command line. NOTE: these settings
                          persist across suite restarts, but can be set again on
                          the "cylc restart" command line if they need to be

```
                              overridden.
    --set-file=FILE           Set the value of Jinja2 template variables in the
                              suite definition from a file containing NAME=VALUE
                              pairs (one per line). NOTE: these settings persist
                              across suite restarts, but can be set again on the
                              "cylc restart" command line if they need to be
                              overridden.
```

E.2.8   checkpoint

Usage: cylc [control] checkpoint [OPTIONS] REG CHECKPOINT-NAME

Tell suite to checkpoint its current state.

Arguments:
```
    REG                           Suite name
    CHECKPOINT-NAME               Checkpoint name
```
Options:
```
  -h, --help                  show this help message and exit
    --user=USER               Other user account name. This results in command
                              reinvocation on the remote account.
    --host=HOST               Other host name. This results in command reinvocation
                              on the remote account.
  -v, --verbose               Verbose output mode.
    --debug                   Run suites in non-daemon mode, and show exception
                              tracebacks.
    --port=INT                Suite port number on the suite host. NOTE: this is
                              retrieved automatically if non-interactive ssh is
                              configured to the suite host.
    --use-ssh                 Use ssh to re-invoke the command on the suite host.
    --no-login                Do not use a login shell to run remote ssh commands.
                              The default is to use a login shell.
    --comms-timeout=SEC, --pyro-timeout=SEC
                              Set a timeout for network connections to the running
                              suite. The default is no timeout. For task messaging
                              connections see site/user config file documentation.
    --print-uuid              Print the client UUID to stderr. This can be matched
                              to information logged by the receiving suite daemon.
    --set-uuid=UUID           Set the client UUID manually (e.g. from prior use of
                              --print-uuid). This can be used to log multiple
                              commands under the same UUID (but note that only the
                              first [info] command from the same client ID will be
                              logged unless the suite is running in debug mode).
  -f, --force                 Do not ask for confirmation before acting. Note that
                              it is not necessary to use this option if interactive
                              command prompts have been disabled in the site/user
                              config files.
```

E.2.9   conditions

Usage: cylc [license] warranty [--help]
Cylc is release under the GNU General Public License v3.0
This command prints the GPL v3.0 license in full.

Options:
```
  --help   Print this usage message.
```

E.2.10   cycle-point

Usage: cylc [util] cycle-point [OPTIONS] [POINT]

Cycle point date-time offset computation, and filename templating.

Filename templating replaces elements of a template string with corresponding
elements of the current or given cycle point.

Use ISO 8601 or posix date-time format elements:

```
  % cylc cyclepoint 2010080T00 --template foo-CCYY-MM-DD-Thh.nc
  foo-2010-08-08-T00.nc
  % cylc cyclepoint 2010080T00 --template foo-%Y-%m-%d-Thh.nc
  foo-2010-08-08-T00.nc
```

Other examples:

```
1) print offset from an explicit cycle point:
  % cylc [util] cycle-point --offset-hours=6 20100823T1800Z
  20100824T0000Z

2) print offset from $CYLC_TASK_CYCLE_POINT (as in suite tasks):
  % export CYLC_TASK_CYCLE_POINT=20100823T1800Z
  % cylc cycle-point --offset-hours=-6
  20100823T1200Z

3) cycle point filename templating, explicit template:
  % export CYLC_TASK_CYCLE_POINT=2010-08
  % cylc cycle-point --offset-years=2 --template=foo-CCYY-MM.nc
  foo-2012-08.nc

4) cycle point filename templating, template in a variable:
  % export CYLC_TASK_CYCLE_POINT=2010-08
  % export MYTEMPLATE=foo-CCYY-MM.nc
  % cylc cycle-point --offset-years=2 --template=MYTEMPLATE
  foo-2012-08.nc
```

Arguments:
```
  [POINT]  ISO 8601 date-time, e.g. 20140201T0000Z, default
     $CYLC_TASK_CYCLE_POINT
```

Options:
```
  -h, --help              show this help message and exit
  --offset-hours=HOURS    Add N hours to CYCLE (may be negative)
  --offset-days=DAYS      Add N days to CYCLE (N may be negative)
  --offset-months=MONTHS
                          Add N months to CYCLE (N may be negative)
  --offset-years=YEARS    Add N years to CYCLE (N may be negative)
  --offset=ISO_OFFSET     Add an ISO 8601-based interval representation to CYCLE
  --equal=POINT2          Succeed if POINT2 is equal to POINT (format agnostic).
  --template=TEMPLATE     Filename template string or variable
  --time-zone=TEMPLATE    Control the formatting of the result's timezone e.g.
                          (Z, +13:00, -hh
  --num-expanded-year-digits=NUMBER
                          Specify a number of expanded year digits to print in
                          the result
  --print-year            Print only CCYY of result
  --print-month           Print only MM of result
  --print-day             Print only DD of result
  --print-hour            Print only hh of result
```

E.2.11   diff

Compare two suite definitions and display any differences.

Differencing is done after parsing the suite.rc files so it takes
account of default values that are not explicitly defined, it disregards
the order of configuration items, and it sees any include-file content
after inlining has occurred.

Files in the suite bin directory and other sub-directories of the
suite definition directory are not currently differenced.

Arguments:
```
  SUITE1                  Suite name or path
  SUITE2                  Suite name or path
```

Options:
```
  -h, --help              show this help message and exit
```

```
   -n, --nested            print suite.rc section headings in nested form.
   --user=USER             Other user account name. This results in command
                           reinvocation on the remote account.
   --host=HOST             Other host name. This results in command reinvocation
                           on the remote account.
   -v, --verbose           Verbose output mode.
   --debug                 Run suites in non-daemon mode, and show exception
                           tracebacks.
   --suite-owner=OWNER     Specify suite owner
   -s NAME=VALUE, --set=NAME=VALUE
                           Set the value of a Jinja2 template variable in the
                           suite definition. This option can be used multiple
                           times on the command line. NOTE: these settings
                           persist across suite restarts, but can be set again on
                           the "cylc restart" command line if they need to be
                           overridden.
   --set-file=FILE         Set the value of Jinja2 template variables in the
                           suite definition from a file containing NAME=VALUE
                           pairs (one per line). NOTE: these settings persist
                           across suite restarts, but can be set again on the
                           "cylc restart" command line if they need to be
                           overridden.
   --icp=CYCLE_POINT       Set initial cycle point. Required if not defined in
                           suite.rc.
```

E.2.12   documentation

Usage: cylc [info] documentation|browse [OPTIONS] [SUITE]

Open cylc or suite documentation in your browser or PDF viewer (as defined in cylc global config files).

```
% cylc doc [OPTIONS]
   Open local or internet [--www] cylc documentation (locations must be
specified in cylc global config files).

% cylc doc -u [-t TASK] SUITE
    Open suite or task documentation if corresponding URL items are specified
in the suite definition.
```

Arguments:
```
   [TARGET]     File, URL, or suite name
```

Options:
```
  -h, --help               show this help message and exit
  -p, --pdf                Open the PDF User Guide directly.
  -w, --www                Open the cylc internet homepage
  -t TASK_NAME, --task=TASK_NAME
                           Browse task documentation URLs.
  -s, --stdout             Just print the URL to stdout.
  --user=USER              Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST              Other host name. This results in command reinvocation
                           on the remote account.
```

E.2.13   dump

Usage: cylc [info] dump [OPTIONS] REG

Print state information (e.g. the state of each task) from a running suite. For small suites 'watch cylc [info] dump SUITE' is an effective non-GUI real time monitor (but see also 'cylc monitor').

For more information about a specific task, such as the current state of its prerequisites and outputs, see 'cylc [info] show'.

Examples:
 Display the state of all running tasks, sorted by cycle point:
 % cylc [info] dump --tasks --sort SUITE | grep running

Display the state of all tasks in a particular cycle point:
% cylc [info] dump -t SUITE | grep 2010082406

Arguments:
    REG   Suite name

Options:
  -h, --help   show this help message and exit
  -g, --global   Global information only.
  -t, --tasks   Task states only.
  -r, --raw, --raw-format
                            Display raw format.
  -s, --sort   Task states only; sort by cycle point instead of name.
  --user=USER   Other user account name. This results in command
                            reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                            tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                            retrieved automatically if non-interactive ssh is
                            configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                            The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                            Set a timeout for network connections to the running
                            suite. The default is no timeout. For task messaging
                            connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                            to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                            --print-uuid). This can be used to log multiple
                            commands under the same UUID (but note that only the
                            first [info] command from the same client ID will be
                            logged unless the suite is running in debug mode).

E.2.14   edit

Usage: cylc [prep] edit [OPTIONS] SUITE

Edit suite definitions without having to move to their directory
locations, and with optional reversible inlining of include-files. Note
that Jinja2 suites can only be edited in raw form but the processed
version can be viewed with 'cylc [prep] view -p'.

1/cylc [prep] edit SUITE
Change to the suite definition directory and edit the suite.rc file.

2/ cylc [prep] edit -i,--inline SUITE
Edit the suite with include-files inlined between special markers. The
original suite.rc file is temporarily replaced so that the inlined
version is "live" during editing (i.e. you can run suites during
editing and cylc will pick up changes to the suite definition). The
inlined file is then split into its constituent include-files
again when you exit the editor. Include-files can be nested or
multiply-included; in the latter case only the first inclusion is
inlined (this prevents conflicting changes made to the same file).

3/ cylc [prep] edit --cleanup SUITE
Remove backup files left by previous INLINED edit sessions.

INLINED EDITING SAFETY: The suite.rc file and its include-files are
automatically backed up prior to an inlined editing session. If the
editor dies mid-session just invoke 'cylc edit -i' again to recover from
the last saved inlined file. On exiting the editor, if any of the
original include-files are found to have changed due to external
intervention during editing you will be warned and the affected files
will be written to new backups instead of overwriting the originals.
Finally, the inlined suite.rc file is also backed up on exiting

the editor, to allow recovery in case of accidental corruption of the
include-file boundary markers in the inlined file.

The edit process is spawned in the foreground as follows:
  % <editor> suite.rc
Where <editor> is defined in the cylc site/user config files.

See also 'cylc [prep] view'.

Arguments:
    SUITE   Suite name or path

Options:
  -h, --help   show this help message and exit
  -i, --inline   Edit with include-files inlined as described above.
  --cleanup   Remove backup files left by previous inlined edit
                          sessions.
  -g, --gui   Force use of the configured GUI editor.
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --suite-owner=OWNER   Specify suite owner

E.2.15   email-suite

Usage: cylc [hook] email-suite EVENT SUITE MESSAGE

This is a simple suite event hook script that sends an email.
The command line arguments are supplied automatically by cylc.

For example, to get an email alert when a suite shuts down:

SUITE.RC
[cylc]
   [[environment]]
      MAIL_ADDRESS = foo@bar.baz.waz
   [[events]]
      shutdown handler = cylc email-suite

See the Suite.rc Reference (Cylc User Guide) for more information
on suite and task event hooks and event handler scripts.

E.2.16   email-task

Usage: cylc [hook] email-task EVENT SUITE TASKID MESSAGE

This is a simple task event hook handler script that sends an email.
The command line arguments are supplied automatically by cylc.

For example, to get an email alert whenever any task fails:

SUITE.RC
[cylc]
   [[environment]]
      MAIL_ADDRESS = foo@bar.baz.waz
[runtime]
   [[root]]
      [[[events]]]
         failed handler = cylc email-task

See the Suite.rc Reference (Cylc User Guide) for more information
on suite and task event hooks and event handler scripts.

E.2.17   ext-trigger

Usage: cylc [control] ext-trigger [OPTIONS] REG MSG ID

Report an external event message to a suite daemon. It is expected that a
task in the suite has registered the same message as an external trigger - a
special prerequisite to be satisifed by an external system, via this command,
rather than by triggering off other tasks.

The ID argument should uniquely distinguish one external trigger event from the
next. When a task's external trigger is satisfied by an incoming message, the
message ID is broadcast to all downstream tasks in the cycle point as
$CYLC_EXT_TRIGGER_ID so that they can use it - e.g. to identify a new data file
that the external triggering system is responding to.

Use the retry options in case the target suite is down or out of contact.

The suite passphrase must be installed in $HOME/.cylc/<SUITE>/.

Note: to manually trigger a task use 'cylc trigger', not this command.

Arguments:
    REG   Suite name
    MSG   External trigger message
    ID   Unique trigger ID

Options:
  -h, --help   show this help message and exit
  --max-tries=INT   Maximum number of send attempts (default 5).
  --retry-interval=SEC   Delay in seconds before retrying (default 10.0).
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                          Set a timeout for network connections to the running
                          suite. The default is no timeout. For task messaging
                          connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                          to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                          --print-uuid). This can be used to log multiple
                          commands under the same UUID (but note that only the
                          first [info] command from the same client ID will be
                          logged unless the suite is running in debug mode).
  -f, --force   Do not ask for confirmation before acting. Note that
                          it is not necessary to use this option if interactive
                          command prompts have been disabled in the site/user
                          config files.

E.2.18   get-directory

Usage: cylc [prep] get-directory REG

Retrieve and print the source directory location of suite REG.
Here's an easy way to move to a suite source directory:
  $ cd $(cylc get-dir REG).

Arguments:
    SUITE   Suite name or path

Options:
  -h, --help   show this help message and exit
  --user=USER   Other user account name. This results in command

```
                            reinvocation on the remote account.
  --host=HOST               Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose             Verbose output mode.
  --debug                   Run suites in non-daemon mode, and show exception
                            tracebacks.
  --suite-owner=OWNER       Specify suite owner
```

E.2.19   get-gui-config

```
Usage: cylc [admin] get-gui-config [OPTIONS]

Print gcylc configuration settings.

By default all settings are printed. For specific sections or items
use -i/--item and wrap parent sections in square brackets:
   cylc get-gui-config --item '[themes][default]succeeded'
Multiple items can be specified at once.

Options:
  -h, --help              show this help message and exit
  -v, --verbose           Print extra information.
  --debug                 Show exception tracebacks.
  -i [SEC...]ITEM, --item=[SEC...]ITEM
                          Item or section to print (multiple use allowed).
  --sparse                Only print items explicitly set in the config files.
  -p, --python            Print native Python format.
```

E.2.20   get-site-config

```
Usage: cylc [admin] get-site-config [OPTIONS]

Print cylc site/user configuration settings.

By default all settings are printed. For specific sections or items
use -i/--item and wrap parent sections in square brackets:
   cylc get-site-config --item '[editors]terminal'
Multiple items can be specified at once.

Options:
  -h, --help              show this help message and exit
  -i [SEC...]ITEM, --item=[SEC...]ITEM
                          Item or section to print (multiple use allowed).
  --sparse                Only print items explicitly set in the config files.
  -p, --python            Print native Python format.
  --print-run-dir         Print the configured cylc run directory.
  --print-site-dir        Print the cylc site configuration directory location.
  -v, --verbose           Print extra information.
  --debug                 Show exception tracebacks.
```

E.2.21   get-suite-config

```
Usage: cylc [info] get-suite-config [OPTIONS] SUITE

Print parsed suite configuration items, after runtime inheritance.

By default all settings are printed. For specific sections or items
use -i/--item and wrap sections in square brackets, e.g.:
   cylc get-suite-config --item '[scheduling]initial cycle point'
Multiple items can be retrieved at once.

By default, unset values are printed as an empty string, or (for
historical reasons) as "None" with -o/--one-line. These defaults
can be changed with the -n/--null-value option.

Example:
  |# SUITE.RC
  |[runtime]
  |    [[modelX]]
```

```
|          [[[environment]]]
|              FOO = foo
|              BAR = bar

$ cylc get-suite-config --item=[runtime][modelX][environment]FOO SUITE
foo

$ cylc get-suite-config --item=[runtime][modelX][environment] SUITE
FOO = foo
BAR = bar

$ cylc get-suite-config --item=[runtime][modelX] SUITE
...
[[[environment]]]
    FOO = foo
    BAR = bar
...
```

Arguments:
```
  SUITE                     Suite name or path
```

Options:
```
  -h, --help                show this help message and exit
  -i [SEC...]ITEM, --item=[SEC...]ITEM
                            Item or section to print (multiple use allowed).
  -r, --sparse              Only print items explicitly set in the config files.
  -p, --python              Print native Python format.
  -a, --all-tasks           For [runtime] items (e.g. --item='script') report
                            values for all tasks prefixed by task name.
  -n STRING, --null-value=STRING
                            The string to print for unset values (default
                            nothing).
  -m, --mark-up             Prefix each line with '!cylc!'.
  -o, --one-line            Print multiple single-value items at once.
  -t, --tasks               Print the suite task list [DEPRECATED: use 'cylc list
                            SUITE'].
  -u RUN_MODE, --run-mode=RUN_MODE
                            Get config for suite run mode.
  --user=USER               Other user account name. This results in command
                            reinvocation on the remote account.
  --host=HOST               Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose             Verbose output mode.
  --debug                   Run suites in non-daemon mode, and show exception
                            tracebacks.
  --suite-owner=OWNER       Specify suite owner
  -s NAME=VALUE, --set=NAME=VALUE
                            Set the value of a Jinja2 template variable in the
                            suite definition. This option can be used multiple
                            times on the command line. NOTE: these settings
                            persist across suite restarts, but can be set again on
                            the "cylc restart" command line if they need to be
                            overridden.
  --set-file=FILE           Set the value of Jinja2 template variables in the
                            suite definition from a file containing NAME=VALUE
                            pairs (one per line). NOTE: these settings persist
                            across suite restarts, but can be set again on the
                            "cylc restart" command line if they need to be
                            overridden.
  --icp=CYCLE_POINT         Set initial cycle point. Required if not defined in
                            suite.rc.
```

E.2.22   get-suite-contact

```
Usage: cylc [info] get-suite-contact [OPTIONS] REG
```

Print contact information of running suite REG.

Arguments:
```
  REG                       Suite name
```

```
Options:
  -h, --help      show this help message and exit
  --user=USER     Other user account name. This results in command reinvocation
                  on the remote account.
  --host=HOST     Other host name. This results in command reinvocation on the
                  remote account.
  -v, --verbose   Verbose output mode.
  --debug         Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.23   get-suite-version

`Usage: cylc [info] get-suite-version [OPTIONS] REG`

Interrogate running suite REG to find what version of cylc is running it.

To find the version you've invoked at the command line see `"cylc version"`.

`Arguments:`
```
  REG                  Suite name
```

`Options:`
```
  -h, --help              show this help message and exit
  --user=USER             Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST             Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose           Verbose output mode.
  --debug                 Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT              Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh               Use ssh to re-invoke the command on the suite host.
  --no-login              Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                          Set a timeout for network connections to the running
                          suite. The default is no timeout. For task messaging
                          connections see site/user config file documentation.
  --print-uuid            Print the client UUID to stderr. This can be matched
                          to information logged by the receiving suite daemon.
  --set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                          --print-uuid). This can be used to log multiple
                          commands under the same UUID (but note that only the
                          first [info] command from the same client ID will be
                          logged unless the suite is running in debug mode).
  -f, --force             Do not ask for confirmation before acting. Note that
                          it is not necessary to use this option if interactive
                          command prompts have been disabled in the site/user
                          config files.
```

E.2.24   gpanel

`Usage: cylc gpanel [OPTIONS]`

This is a cylc scan panel applet for monitoring running suites on a set of hosts in GNOME 2.

To install this applet, run `"cylc gpanel --install"` and follow the instructions that it gives you.

This applet can be tested using the `--test` option.

To customize themes, copy $CYLC_DIR/conf/gcylcrc/gcylc.rc.eg to $HOME/.cylc/gcylc.rc and follow the instructions in the file.

To configure default suite hosts, edit the [suite host scanning]hosts entry in your global.rc file.

`Options:`

```
-h, --help   show this help message and exit
--compact    Switch on compact mode at runtime.
--install    Install the panel applet.
--test       Run in a standalone window.
```

E.2.25   graph

```
       Plot the suite.rc dependency graph for SUITE.
         2/ cylc [prep] graph [OPTIONS] -f,--file FILE
       Plot the specified dot-language graph file.
         3/ cylc [prep] graph [OPTIONS] --reference SUITE [START[STOP]]
       Print out a reference format for the dependencies in SUITE.
         4/ cylc [prep] graph [OPTIONS] --output-file FILE SUITE
       Plot SUITE dependencies to a file FILE with a extension-derived format.
       If FILE endswith ".png", output in PNG format, etc.
```

```
Plot suite dependency graphs in an interactive graph viewer.
```

```
If START is given it overrides "[visualization] initial cycle point" to
determine the start point of the graph, which defaults to the suite initial
cycle point. If STOP is given it overrides "[visualization] final cycle point"
to determine the end point of the graph, which defaults to the graph start
point plus "[visualization] number of cycle points" (which defaults to 3).
The graph start and end points are adjusted up and down to the suite initial
and final cycle points, respectively, if necessary.
```

```
The "Save" button generates an image of the current view, of format (e.g. png,
svg, jpg, eps) determined by the filename extension. If the chosen format is
not available a dialog box will show those that are available.
```

```
If the optional output filename is specified, the viewer will not open and a
graph will be written directly to the file.
```

```
GRAPH VIEWER CONTROLS:
    * Center on a node: left-click.
    * Pan view: left-drag.
    * Zoom: +/- buttons, mouse-wheel, or ctrl-left-drag.
    * Box zoom: shift-left-drag.
    * "Best Fit" and "Normal Size" buttons.
    * Left-to-right graphing mode toggle button.
    * "Ignore suicide triggers" button.
    * "Save" button: save an image of the view.
  Family (namespace) grouping controls:
    Toolbar:
    * "group" - group all families up to root.
    * "ungroup" - recursively ungroup all families.
    Right-click menu:
    * "group" - close this node's parent family.
    * "ungroup" - open this family node.
    * "recursive ungroup" - ungroup all families below this node.
```

```
   [SUITE]                  Suite name or path
   [START]                  Initial cycle point (default: suite initial point)
   [STOP]                   Final cycle point (default: initial + 3 points)
```

```
  -h, --help              show this help message and exit
  -u, --ungrouped         Start with task families ungrouped (the default is
                          grouped).
  -n, --namespaces        Plot the suite namespace inheritance hierarchy (task
                          run time properties).
  -f FILE, --file=FILE    View a specific dot-language graphfile.
  --filter=NODE_NAME_PATTERN
                          Filter out one or many nodes.
  -O FILE, --output-file=FILE
                          Output to a specific file, with a format given by
                          --output-format or extrapolated from the extension.
                          '-' implies stdout in plain format.
  --output-format=FORMAT
```

```
                              Specify a format for writing out the graph to
                              --output-file e.g. png, svg, jpg, eps, dot. 'ref' is a
                              special sorted plain text format for comparison and
                              reference purposes.
  -r, --reference             Output in a sorted plain text format for comparison
                              purposes. If not given, assume --output-file=-.
  --show-suicide              Show suicide triggers.  They are not shown by default,
                              unless toggled on with the tool bar button.
  --user=USER                 Other user account name. This results in command
                              reinvocation on the remote account.
  --host=HOST                 Other host name. This results in command reinvocation
                              on the remote account.
  -v, --verbose               Verbose output mode.
  --debug                     Run suites in non-daemon mode, and show exception
                              tracebacks.
  --suite-owner=OWNER         Specify suite owner
  -s NAME=VALUE, --set=NAME=VALUE
                              Set the value of a Jinja2 template variable in the
                              suite definition. This option can be used multiple
                              times on the command line. NOTE: these settings
                              persist across suite restarts, but can be set again on
                              the "cylc restart" command line if they need to be
                              overridden.
  --set-file=FILE             Set the value of Jinja2 template variables in the
                              suite definition from a file containing NAME=VALUE
                              pairs (one per line). NOTE: these settings persist
                              across suite restarts, but can be set again on the
                              "cylc restart" command line if they need to be
                              overridden.
```

E.2.26   graph-diff

```
Usage: cylc graph-diff [OPTIONS] SUITE1 SUITE2 -- [GRAPH_OPTIONS_ARGS]

Difference 'cylc graph --reference' output for SUITE1 and SUITE2.

OPTIONS: Use '-g' to launch a graphical diff utility.
         Use '--diff-cmd=MY_DIFF_CMD' to use a custom diff tool.

SUITE1, SUITE2: Suite names to compare.
GRAPH_OPTIONS_ARGS: Options and arguments passed directly to cylc graph.
```

E.2.27   gscan

```
Usage: cylc gscan [OPTIONS]

This is the cylc scan gui for monitoring running suites on a set of
hosts.

To customize themes copy $CYLC_DIR/conf/gcylcrc/gcylc.rc.eg to
$HOME/.cylc/gcylc.rc and follow the instructions in the file.

Arguments:
   [HOSTS ...]                Hosts to scan instead of the configured hosts.

Options:
  -h, --help                 show this help message and exit
  -a, --all, --all-suites
                             List all suites found on all scanned hosts (the
                             default is just your own suites).
  -n PATTERN, --name=PATTERN
                             List suites with name matching PATTERN (regular
                             expression). Defaults to any name. Can be used
                             multiple times.
  -o PATTERN, --suite-owner=PATTERN
                             List suites with owner matching PATTERN (regular
                             expression). Defaults to just your own suites. Can be
                             used multiple times.
  --comms-timeout=SEC        Set a timeout for network connections to each running
                             suite. The default is 5 seconds.
```

```
--poll-interval=SECONDS
                        Polling interval (time between updates) in seconds
--user=USER             Other user account name. This results in command
                        reinvocation on the remote account.
--host=HOST             Other host name. This results in command reinvocation
                        on the remote account.
-v, --verbose           Verbose output mode.
--debug                 Run suites in non-daemon mode, and show exception
                        tracebacks.
--port=INT              Suite port number on the suite host. NOTE: this is
                        retrieved automatically if non-interactive ssh is
                        configured to the suite host.
--use-ssh               Use ssh to re-invoke the command on the suite host.
--no-login              Do not use a login shell to run remote ssh commands.
                        The default is to use a login shell.
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
```

E.2.28   gui

Usage: cylc gui [OPTIONS] [REG] [USER_AT_HOST]
gcylc [OPTIONS] [REG] [USER_AT_HOST]

This is the cylc Graphical User Interface.

The USER_AT_HOST argument allows suite selection by 'cylc scan' output:
  cylc gui $(cylc scan | grep <suite_name>)

Local suites can be opened and switched between from within gcylc. To connect
to running remote suites (whose passphrase you have installed) you must
currently use --host and/or --user on the gcylc command line.

Available task state color themes are shown under the View menu. To customize
themes copy $CYLC_DIR/conf/gcylcrc/gcylc.rc.eg to $HOME/.cylc/gcylc.rc and
follow the instructions in the file.

To see current configuration settings use "cylc get-gui-config".

In the graph view, View -> Options -> "Write Graph Frames" writes .dot graph
files to the suite share directory (locally, for a remote suite). These can
be processed into a movie by \$CYLC_DIR/dev/bin/live-graph-movie.sh=.

Arguments:
    [REG]   Suite name
    [USER_AT_HOST]   user@host:port, shorthand for --user, --host &
        --port.

Options:
  -h, --help   show this help message and exit
  -r, --restricted   Restrict display to 'active' task states: submitted,
                          submit-failed, submit-retrying, running, failed,
                          retrying; and disable the graph view. This may be
                          needed for very large suites. The state summary icons
                          in the status bar still represent all task proxies.
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
```

```
                           The default is to use a login shell.
  --comms-timeout=SEC , --pyro-timeout=SEC
                           Set a timeout for network connections to the running
                           suite. The default is no timeout. For task messaging
                           connections see site/user config file documentation.
  --print-uuid             Print the client UUID to stderr. This can be matched
                           to information logged by the receiving suite daemon.
  --set-uuid=UUID          Set the client UUID manually (e.g. from prior use of
                           --print-uuid). This can be used to log multiple
                           commands under the same UUID (but note that only the
                           first [info] command from the same client ID will be
                           logged unless the suite is running in debug mode).
  -s NAME=VALUE , --set=NAME=VALUE
                           Set the value of a Jinja2 template variable in the
                           suite definition. This option can be used multiple
                           times on the command line. NOTE: these settings
                           persist across suite restarts , but can be set again on
                           the "cylc restart" command line if they need to be
                           overridden.
  --set-file=FILE          Set the value of Jinja2 template variables in the
                           suite definition from a file containing NAME=VALUE
                           pairs (one per line). NOTE: these settings persist
                           across suite restarts , but can be set again on the
                           "cylc restart" command line if they need to be
                           overridden.
```

E.2.29   hold

Usage: cylc [control] hold [OPTIONS] REG [TASKID ...]

Hold one or more waiting tasks (cylc hold REG TASKID ...), or
a whole suite (cylc hold REG).

Held tasks do not submit even if they are ready to run.

See also 'cylc [control] release'.

A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example , to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
    REG   Suite name
    [TASKID ...]   Task identifiers

Options:
  -h, --help   show this help message and exit
  --after=CYCLE_POINT   Hold whole suite AFTER this cycle point.
  --user=USER   Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                           on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode , and show exception
                           tracebacks.

```
--port=INT              Suite port number on the suite host. NOTE: this is
                        retrieved automatically if non-interactive ssh is
                        configured to the suite host.
--use-ssh               Use ssh to re-invoke the command on the suite host.
--no-login              Do not use a login shell to run remote ssh commands.
                        The default is to use a login shell.
--comms-timeout=SEC, --pyro-timeout=SEC
                        Set a timeout for network connections to the running
                        suite. The default is no timeout. For task messaging
                        connections see site/user config file documentation.
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
-f, --force             Do not ask for confirmation before acting. Note that
                        it is not necessary to use this option if interactive
                        command prompts have been disabled in the site/user
                        config files.
-m, --family            (Obsolete) This option is now ignored and is retained
                        for backward compatibility only. TASKID in the
                        argument list can be used to match task and family
                        names regardless of this option.
--no-multitask-compat
                        Disallow backward compatible multitask interface.
```

E.2.30 import-examples

Usage: cylc [admin] import-examples DIR

Copy the cylc example suites to DIR and register them for use under the GROUP
suite name group.

Arguments:
    DIR   destination directory

E.2.31 insert

Usage: cylc [control] insert [OPTIONS] REG TASKID [...]

Insert task proxies into a running suite. Uses of insertion include:
 1) insert a task that was excluded by the suite definition at start-up.
 2) reinstate a task that was previously removed from a running suite.
 3) re-run an old task that cannot be retriggered because its task proxy
 is no longer live in the a suite.

Be aware that inserted cycling tasks keep on cycling as normal, even if
another instance of the same task exists at a later cycle (instances of
the same task at different cycles can coexist, but a newly spawned task
will not be added to the pool if it catches up to another task with the
same ID).

See also 'cylc submit', for running tasks without the scheduler.

A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or

```
'BA[RZ].*:retrying'
```

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
    REG   Suite name
    TASKID [...]   Task identifier

Options:
  -h, --help   show this help message and exit
  --stop-point=CYCLE_POINT, --remove-point=CYCLE_POINT
                            Optional hold/stop cycle point for inserted task.
  --no-check   Add task even if the provided cycle point is not valid
                            for the given task.
  --user=USER   Other user account name. This results in command
                            reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                            tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                            retrieved automatically if non-interactive ssh is
                            configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                            The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                            Set a timeout for network connections to the running
                            suite. The default is no timeout. For task messaging
                            connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                            to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                            --print-uuid). This can be used to log multiple
                            commands under the same UUID (but note that only the
                            first [info] command from the same client ID will be
                            logged unless the suite is running in debug mode).
  -f, --force   Do not ask for confirmation before acting. Note that
                            it is not necessary to use this option if interactive
                            command prompts have been disabled in the site/user
                            config files.
  -m, --family   (Obsolete) This option is now ignored and is retained
                            for backward compatibility only. TASKID in the
                            argument list can be used to match task and family
                            names regardless of this option.
  --no-multitask-compat
                            Disallow backward compatible multitask interface.
```

E.2.32 job-logs-retrieve

Usage: cylc [hook] job-logs-retrieve [OPTIONS] HOST:HOST-PATH LOCALHOST-PATH

(This command is for internal use.)
Retrieve logs from a remote host for a task job.

Arguments:
    HOST:HOST-PATH   Path to remote job logs directory
    LOCALHOST-PATH   Path to local job logs directory

Options:
  -h, --help   show this help message and exit
  --max-size=SIZE   Don't transfer any file larger than SIZE.
  --user=USER   Other user account name. This results in command
                        reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation on
                        the remote account.

```
  -v, --verbose    Verbose output mode.
  --debug          Run suites in non-daemon mode, and show exception
                   tracebacks.
```

E.2.33   job-submit

```
Usage: cylc [task] job-submit [--remote-mode] JOB-FILE-PATH
```

```
(This command is for internal use. Users should use "cylc submit".)
Submit a job file.
```

```
Arguments:
    JOB-FILE-PATH                    the path of the job file

Options:
  -h, --help       show this help message and exit
  --remote-mode    Is this being run on a remote job host?
  --user=USER      Other user account name. This results in command reinvocation
                   on the remote account.
  --host=HOST      Other host name. This results in command reinvocation on the
                   remote account.
  -v, --verbose    Verbose output mode.
  --debug          Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.34   jobs-kill

```
Usage: cylc [control] jobs-kill JOB-LOG-ROOT [JOB-LOG-DIR ...]
```

```
(This command is for internal use. Users should use "cylc kill".) Read job
status files to obtain the names of the batch systems and the job IDs in the
systems. Invoke the relevant batch system commands to ask the batch systems to
terminate the jobs.
```

```
Arguments:
    JOB-LOG-ROOT                     The log/job sub-directory for the suite
    [JOB-LOG-DIR ...]                A point/name/submit_num sub-directory

Options:
  -h, --help       show this help message and exit
  --user=USER      Other user account name. This results in command reinvocation
                   on the remote account.
  --host=HOST      Other host name. This results in command reinvocation on the
                   remote account.
  -v, --verbose    Verbose output mode.
  --debug          Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.35   jobs-poll

```
Usage: cylc [control] jobs-poll JOB-LOG-ROOT [JOB-LOG-DIR ...]
```

```
(This command is for internal use. Users should use "cylc poll".) Read job
status files to obtain the statuses of the jobs. If necessary, Invoke the
relevant batch system commands to ask the batch systems for more statuses.
```

```
Arguments:
    JOB-LOG-ROOT                     The log/job sub-directory for the suite
    [JOB-LOG-DIR ...]                A point/name/submit_num sub-directory

Options:
  -h, --help       show this help message and exit
  --user=USER      Other user account name. This results in command reinvocation
                   on the remote account.
  --host=HOST      Other host name. This results in command reinvocation on the
```

```
                                  remote account.
    -v, --verbose    Verbose output mode.
    --debug          Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.36   jobs-submit

Usage: cylc [control] jobs-submit JOB-LOG-ROOT [JOB-LOG-DIR ...]

(This command is for internal use. Users should use "cylc submit".) Submit task jobs to relevant batch systems. On a remote job host, this command reads the job files from STDIN.

```
Arguments:
    JOB-LOG-ROOT                      The log/job sub-directory for the suite
    [JOB-LOG-DIR ...]                 A point/name/submit_num sub-directory

Options:
  -h, --help       show this help message and exit
  --remote-mode    Is this being run on a remote job host?
  --user=USER      Other user account name. This results in command reinvocation
                   on the remote account.
  --host=HOST      Other host name. This results in command reinvocation on the
                   remote account.
  -v, --verbose    Verbose output mode.
  --debug          Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.37   jobscript

Usage: cylc [prep] jobscript [OPTIONS] REG TASK

Generate a task job script and print it to stdout.

```
Here's how to capture the script in the vim editor:
  % cylc jobscript REG TASK | vim -
Emacs unfortunately cannot read from stdin:
  % cylc jobscript REG TASK > tmp.sh; emacs tmp.sh
```

This command wraps 'cylc [control] submit --dry-run'.
Other options (e.g. for suite host and owner) are passed
through to the submit command.

```
Options:
  -h,--help   - print this usage message.
 (see also 'cylc submit --help')

Arguments:
  REG          - Registered suite name.
  TASK         - Task ID (NAME.CYCLE_POINT)
```

E.2.38   kill

Usage: cylc [control] kill [OPTIONS] REG [TASKID ...]

Kill jobs of active tasks and update their statuses accordingly.

To kill one or more tasks, "cylc kill REG TASKID ..."; to kill all active tasks: "cylc kill REG".

```
A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
```

```
For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
```

* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
    REG   Suite name
    [TASKID ...]   Task identifiers

Options:
  -h, --help   show this help message and exit
  --user=USER   Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                           on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                           tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                           retrieved automatically if non-interactive ssh is
                           configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                           The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                           Set a timeout for network connections to the running
                           suite. The default is no timeout. For task messaging
                           connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                           to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                           --print-uuid). This can be used to log multiple
                           commands under the same UUID (but note that only the
                           first [info] command from the same client ID will be
                           logged unless the suite is running in debug mode).
  -f, --force   Do not ask for confirmation before acting. Note that
                           it is not necessary to use this option if interactive
                           command prompts have been disabled in the site/user
                           config files.
  -m, --family   (Obsolete) This option is now ignored and is retained
                           for backward compatibility only. TASKID in the
                           argument list can be used to match task and family
                           names regardless of this option.
  --no-multitask-compat
                           Disallow backward compatible multitask interface.

E.2.39   list

Usage: cylc [info|prep] list|ls [OPTIONS] SUITE

Print runtime namespace names (tasks and families), the first-parent
inheritance graph, or actual tasks for a given cycle range.

The first-parent inheritance graph determines the primary task family
groupings that are collapsible in gcylc suite views and the graph
viewer tool. To visualize the full multiple inheritance hierarchy use:
  'cylc graph -n'.

Arguments:
    SUITE   Suite name or path

Options:
  -h, --help   show this help message and exit
  -a, --all-tasks   Print all tasks, not just those used in the graph.
  -n, --all-namespaces   Print all runtime namespaces, not just tasks.

```
  -m, --mro                Print the linear "method resolution order" for each
                           namespace (the multiple-inheritance precedence order
                           as determined by the C3 linearization algorithm).
  -t, --tree               Print the first-parent inheritance hierarchy in tree
                           form.
  -b, --box                With -t/--tree, using unicode box characters. Your
                           terminal must be able to display unicode characters.
  -w, --with-titles        Print namespaces titles too.
  -p START[,STOP], --points=START[,STOP]
                           Print actual task IDs from the START [through STOP]
                           cycle points.
  --user=USER              Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST              Other host name. This results in command reinvocation
                           on the remote account.
  -v, --verbose            Verbose output mode.
  --debug                  Run suites in non-daemon mode, and show exception
                           tracebacks.
  --suite-owner=OWNER      Specify suite owner
  -s NAME=VALUE, --set=NAME=VALUE
                           Set the value of a Jinja2 template variable in the
                           suite definition. This option can be used multiple
                           times on the command line. NOTE: these settings
                           persist across suite restarts, but can be set again on
                           the "cylc restart" command line if they need to be
                           overridden.
  --set-file=FILE          Set the value of Jinja2 template variables in the
                           suite definition from a file containing NAME=VALUE
                           pairs (one per line). NOTE: these settings persist
                           across suite restarts, but can be set again on the
                           "cylc restart" command line if they need to be
                           overridden.
  --icp=CYCLE_POINT        Set initial cycle point. Required if not defined in
                           suite.rc.
```

E.2.40   ls-checkpoints

Usage: cylc [info] ls-checkpoints [OPTIONS] REG [ID ...]

In the absence of arguments and the --all option, list checkpoint IDs, their
time and events. Otherwise, display the latest and/or the checkpoints of suite
parameters, task pool and broadcast states in the suite runtime database.

```
Arguments:
  REG                      Suite name
  [ID ...]                 Checkpoint ID (default=latest)

Options:
  -h, --help     show this help message and exit
  -a, --all      Display data of all available checkpoints.
  --user=USER    Other user account name. This results in command reinvocation
                 on the remote account.
  --host=HOST    Other host name. This results in command reinvocation on the
                 remote account.
  -v, --verbose  Verbose output mode.
  --debug        Run suites in non-daemon mode, and show exception tracebacks.
```

E.2.41   message

Usage: cylc [task] message [OPTIONS] MESSAGE ...

This command is part of the cylc task messaging interface, used by
running tasks to communicate progress to their parent suite.

The message command can be used to report "message outputs" completed.
Other messages received by the suite daemon will just be logged.

Suite and task identity are determined from the task execution
environment supplied by the suite (or by the single task 'submit'

command, in which case case the message is just printed to stdout).

Options:
  -h, --help   show this help message and exit
  -p PRIORITY, --priority=PRIORITY
                          message priority: NORMAL, WARNING, or CRITICAL;
                          default NORMAL.
  -v, --verbose   Verbose output mode.

E.2.42   monitor

Usage: cylc [info] monitor [OPTIONS] REG [USER_AT_HOST]

A terminal-based live suite monitor. Exit with 'Ctrl-C'.

The USER_AT_HOST argument allows suite selection by 'cylc scan' output:
  cylc monitor $(cylc scan | grep <suite_name>)

Arguments:
   REG   Suite name
   [USER_AT_HOST]   user@host:port, shorthand for --user, --host &
      --port.

Options:
  -h, --help   show this help message and exit
  -a, --align   Align task names. Only useful for small suites.
  -r, --restricted   Restrict display to active task states. This may be
                          useful for monitoring very large suites. The state
                          summary line still reflects all task proxies.
  -s ORDER, --sort=ORDER
                          Task sort order: "definition" or "alphanumeric".The
                          default is definition order, as determined by global
                          config. (Definition order is the order that tasks
                          appear under [runtime] in the suite definition).
  -o, --once   Show a single view then exit.
  -u, --runahead   Display task proxies in the runahead pool (off by
                          default).
  -i SECONDS, --interval=SECONDS
                          Interval between suite state retrievals, in seconds
                          (default 1).
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                          Set a timeout for network connections to the running
                          suite. The default is no timeout. For task messaging
                          connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                          to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                          --print-uuid). This can be used to log multiple
                          commands under the same UUID (but note that only the
                          first [info] command from the same client ID will be
                          logged unless the suite is running in debug mode).

E.2.43   nudge

Usage: cylc [control] nudge [OPTIONS] REG

Cause the cylc task processing loop to be invoked in a running suite.

This happens automatically when the state of any task changes such that
task processing (dependency negotation etc.) is required, or if a
clock-trigger task is ready to run.

The main reason to use this command is to update the "estimated time till
completion" intervals shown in the tree-view suite control GUI, during
periods when nothing else is happening.

Arguments:
    REG   Suite name

Options:
  -h, --help   show this help message and exit
  --user=USER   Other user account name. This results in command
                            reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                            tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                            retrieved automatically if non-interactive ssh is
                            configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                            The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                            Set a timeout for network connections to the running
                            suite. The default is no timeout. For task messaging
                            connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                            to information logged by the receiving suite daemon.
  --set-uuid=UUID   Set the client UUID manually (e.g. from prior use of
                            --print-uuid). This can be used to log multiple
                            commands under the same UUID (but note that only the
                            first [info] command from the same client ID will be
                            logged unless the suite is running in debug mode).
  -f, --force   Do not ask for confirmation before acting. Note that
                            it is not necessary to use this option if interactive
                            command prompts have been disabled in the site/user
                            config files.

E.2.44   ping

Usage: cylc [discovery] ping [OPTIONS] REG [TASK]

If suite REG is running or TASK in suite REG is currently running,
exit with success status, else exit with error status.

Arguments:
    REG   Suite name
    [TASK]   Task NAME.CYCLE_POINT

Options:
  -h, --help   show this help message and exit
  --print-ports   Print the port range from the cylc site config file.
  --user=USER   Other user account name. This results in command
                            reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                            tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                            retrieved automatically if non-interactive ssh is
                            configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.

```
                         The default is to use a login shell.
--comms-timeout=SEC, --pyro-timeout=SEC
                         Set a timeout for network connections to the running
                         suite. The default is no timeout. For task messaging
                         connections see site/user config file documentation.
--print-uuid             Print the client UUID to stderr. This can be matched
                         to information logged by the receiving suite daemon.
--set-uuid=UUID          Set the client UUID manually (e.g. from prior use of
                         --print-uuid). This can be used to log multiple
                         commands under the same UUID (but note that only the
                         first [info] command from the same client ID will be
                         logged unless the suite is running in debug mode).
-f, --force              Do not ask for confirmation before acting. Note that
                         it is not necessary to use this option if interactive
                         command prompts have been disabled in the site/user
                         config files.
```

E.2.45   poll

Usage: cylc [control] poll [OPTIONS] REG [TASKID ...]

Poll jobs of active tasks to verify or update their statuses.

To poll one or more tasks, "cylc poll REG TASKID"; to poll all active
tasks: "cylc poll REG".

Note that automatic job polling can used to track task status on task hosts
that do not allow any communication by HTTPS or ssh back to the suite host
- see site/user config file documentation.

Polling is also done automatically on restarting a suite, for any tasks that
were recorded as submitted or running when the suite went down.

A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
   REG   Suite name
   [TASKID ...]   Task identifiers

Options:
  -h, --help   show this help message and exit
  --user=USER   Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                           on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                           tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                           retrieved automatically if non-interactive ssh is
                           configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.

```
                            The default is to use a login shell.
    --comms-timeout=SEC, --pyro-timeout=SEC
                            Set a timeout for network connections to the running
                            suite. The default is no timeout. For task messaging
                            connections see site/user config file documentation.
    --print-uuid            Print the client UUID to stderr. This can be matched
                            to information logged by the receiving suite daemon.
    --set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                            --print-uuid). This can be used to log multiple
                            commands under the same UUID (but note that only the
                            first [info] command from the same client ID will be
                            logged unless the suite is running in debug mode).
    -f, --force             Do not ask for confirmation before acting. Note that
                            it is not necessary to use this option if interactive
                            command prompts have been disabled in the site/user
                            config files.
    -m, --family            (Obsolete) This option is now ignored and is retained
                            for backward compatibility only. TASKID in the
                            argument list can be used to match task and family
                            names regardless of this option.
    --no-multitask-compat
                            Disallow backward compatible multitask interface.
```

E.2.46 print

Usage: cylc [prep] print [OPTIONS] [REGEX]

Print registered (installed) suites.

Note on result filtering:
  (a) The filter patterns are Regular Expressions, not shell globs, so
the general wildcard is '.*' (match zero or more of anything), NOT '*'.
  (b) For printing purposes there is an implicit wildcard at the end of
each pattern ('foo' is the same as 'foo/*'); use the string end marker
to prevent this ('foo$' matches only literal 'foo').

Arguments:
    [REGEX]   Suite name regular expression pattern

Options:
  -h, --help   show this help message and exit
  -t, --tree   Print suites in nested tree form.
  -b, --box   Use unicode box drawing characters in tree views.
  -a, --align   Align columns.
  -x   don't print suite definition directory paths.
  -y   Don't print suite titles.
  --fail   Fail (exit 1) if no matching suites are found.
  --user=USER   Other user account name. This results in command reinvocation
                  on the remote account.
  --host=HOST   Other host name. This results in command reinvocation on the
                  remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception tracebacks.

E.2.47 profile-battery

Usage: cylc profile-battery [-e [EXPERIMENT ...]] [-v [VERSION ...]]

Run profiling experiments against different versions of cylc. A list of
experiments can be specified after the -e flag, if not provided the experiment
"complex" will be chosen. A list of versions to profile against can be
specified after the -v flag, if not provided the current version will be used.

Experiments are stored in dev/profile-experiments, user experiments can be
stored in .profiling/experiments. Experiments are specified without the file
extension, experiments in .profiling/ will be chosen before those in dev/.

IMPORTANT: See dev/profile-experiments/example for an experiment template with
further details.

Versions are any valid git identifiers i.e. tags, branches, commits. To compare
results to different cylc versions either:
    * Supply cylc profile-battery with a complete list of the versions you wish
      to profile, it will then provide the option to checkout the required
      versions automatically.
    * Checkout each version manually running cylc profile-battery against only
      one version at a time. Once all results have been gathered you can then
      run cylc profile-battery with a complete list of versions.

Profiling will save results to .profiling/results.json where they can be used
for future comparisons. To list profiling results run:
    * cylc profile-battery --ls # list all results
    * cylc profile-battery --ls -e experiment # list all results for
                                        # experiment "experiment".
    * cylc profile-battery --ls --delete -v 6.1.2 # Delete all results for
                                        # version 6.1.2 (prompted).

If matplotlib and numpy are installed profiling generates plots which are
saved to .profiling/plots or presented in an interactive window using the -i
flag.

Results are stored along with a checksum for the experiment file. When an
experiment file is changed previous results are maintained, future results will
be stored separately. To copy results from an older version of an experiment
into those from the current one run:
    * cylc profile-battery --promote experiment@checksum
NOTE: At present results cannot be analysed without the experiment file so old
results must be "copied" in this way to be re-used.

The results output contain only a small number of metrics, to see a full list
of results use the --full option.

Options:
  -h, --help   show this help message and exit
  -e, --experiments   Specify list of experiments to run.
  -v, --versions   Specify cylc versions to profile. Git tags, branches,
                          commits are all valid.
  -i, --interactive   Open any plots in interactive window rather saving
                          them to files.
  -p, --no-plots   Don't generate any plots.
  --ls, --list-results   List all stored results. Experiments and versions to
                          list can be specified using --experiments and
                          --versions.
  --delete   Delete stored results (to be used in combination with
                          --list-results).
  -y, --yes   Answer yes to any user input. Will check-out cylc
                          versions as required.
  --full-results, --full
                          Display all gathered metrics.
  --lobf-order=LOBF_ORDER
                          The order (int)of the line of best fit to be drawn. 0
                          for no lobf, 1 for linear, 2 for quadratic ect.
  --promote=PROMOTE   Promote results from an older version of an experiment
                          to the current version. To be used when making non-
                          functional changes to an experiment.
  --test   For development purposes, run experiment without
                          saving results and regardless of any prior runs.

E.2.48   register

Usage: cylc [prep] register [OPTIONS] REG [PATH]

Register the suite definition located in PATH (or $PWD) as REG.

This creates the suite run directory, and authentication files in a
sub-directory called ".service/".

Suite names are the same as the directory path under the suite run directory.
They may contain alphanumeric characters plus '_' '-' and '/'.

Example: if the cylc run directory is $HOME/cylc-run (the default) and
/home/bob/suites/test is a suite source directory, then:

% cylc reg nwp/test1 /home/bob/suites/test

will create the following suite run directory:

/home/bob/cylc-run/nwp/test1
'-- .service
    |-- passphrase
    |-- source -> /home/bob/test
    |-- ssl.cert
    '-- ssl.pem

The suite can subsequently be started and targeted by cylc commands using
"nwp/test1" as the name.

Arguments:
    REG   Suite name
    [PATH]   Suite definition directory (defaults to $PWD)

Options:
  -h, --help   show this help message and exit
  --user=USER   Other user account name. This results in command reinvocation
                  on the remote account.
  --host=HOST   Other host name. This results in command reinvocation on the
                  remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception tracebacks.

E.2.49   release

Usage: cylc [control] release|unhold [OPTIONS] REG [TASKID ...]

Release one or more held tasks (cylc release REG TASKID)
or the whole suite (cylc release REG). Held tasks do not
submit even if they are ready to run.

See also 'cylc [control] hold'.

A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
    REG   Suite name
    [TASKID ...]   Task identifiers

Options:
  -h, --help   show this help message and exit
  --user=USER   Other user account name. This results in command
                            reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                            on the remote account.
  -v, --verbose   Verbose output mode.

```
--debug                   Run suites in non-daemon mode, and show exception
                          tracebacks.
--port=INT                Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
--use-ssh                 Use ssh to re-invoke the command on the suite host.
--no-login                Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
--comms-timeout=SEC, --pyro-timeout=SEC
                          Set a timeout for network connections to the running
                          suite. The default is no timeout. For task messaging
                          connections see site/user config file documentation.
--print-uuid              Print the client UUID to stderr. This can be matched
                          to information logged by the receiving suite daemon.
--set-uuid=UUID           Set the client UUID manually (e.g. from prior use of
                          --print-uuid). This can be used to log multiple
                          commands under the same UUID (but note that only the
                          first [info] command from the same client ID will be
                          logged unless the suite is running in debug mode).
-f, --force               Do not ask for confirmation before acting. Note that
                          it is not necessary to use this option if interactive
                          command prompts have been disabled in the site/user
                          config files.
-m, --family              (Obsolete) This option is now ignored and is retained
                          for backward compatibility only. TASKID in the
                          argument list can be used to match task and family
                          names regardless of this option.
--no-multitask-compat
                          Disallow backward compatible multitask interface.
```

E.2.50   reload

Usage: cylc [control] reload [OPTIONS] REG

Tell a suite to reload its definition at run time. All settings
including task definitions, with the exception of suite log
configuration, can be changed on reload. Note that defined tasks can be
be added to or removed from a running suite with the 'cylc insert' and
'cylc remove' commands, without reloading. This command also allows
addition and removal of actual task definitions, and therefore insertion
of tasks that were not defined at all when the suite started (you will
still need to manually insert a particular instance of a newly defined
task). Live task proxies that are orphaned by a reload (i.e. their task
definitions have been removed) will be removed from the task pool if
they have not started running yet. Changes to task definitions take
effect immediately, unless a task is already running at reload time.

If the suite was started with Jinja2 template variables set on the
command line (cylc run --set FOO=bar REG) the same template settings
apply to the reload (only changes to the suite.rc file itself are
reloaded).

If the modified suite definition does not parse, failure to reload will
be reported but no harm will be done to the running suite.

Arguments:
   REG   Suite name

Options:
```
-h, --help                show this help message and exit
--user=USER               Other user account name. This results in command
                          reinvocation on the remote account.
--host=HOST               Other host name. This results in command reinvocation
                          on the remote account.
-v, --verbose             Verbose output mode.
--debug                   Run suites in non-daemon mode, and show exception
                          tracebacks.
--port=INT                Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
--use-ssh                 Use ssh to re-invoke the command on the suite host.
```

```
--no-login             Do not use a login shell to run remote ssh commands.
                       The default is to use a login shell.
--comms-timeout=SEC,  --pyro-timeout=SEC
                       Set a timeout for network connections to the running
                       suite. The default is no timeout. For task messaging
                       connections see site/user config file documentation.
--print-uuid           Print the client UUID to stderr. This can be matched
                       to information logged by the receiving suite daemon.
--set-uuid=UUID        Set the client UUID manually (e.g. from prior use of
                       --print-uuid). This can be used to log multiple
                       commands under the same UUID (but note that only the
                       first [info] command from the same client ID will be
                       logged unless the suite is running in debug mode).
-f, --force            Do not ask for confirmation before acting. Note that
                       it is not necessary to use this option if interactive
                       command prompts have been disabled in the site/user
                       config files.
```

E.2.51   remove

Usage: cylc [control] remove [OPTIONS] REG TASKID [...]

Remove one or more tasks (cylc remove REG TASKID), or all tasks with a
given cycle point (cylc remove REG *.POINT) from a running suite.

Tasks will spawn successors first if they have not done so already.

A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
```
  REG                    Suite name
  TASKID [...]           Task identifiers
```

Options:
```
  -h, --help             show this help message and exit
  --no-spawn             Do not spawn successors before removal.
  --user=USER            Other user account name. This results in command
                         reinvocation on the remote account.
  --host=HOST            Other host name. This results in command reinvocation
                         on the remote account.
  -v, --verbose          Verbose output mode.
  --debug                Run suites in non-daemon mode, and show exception
                         tracebacks.
  --port=INT             Suite port number on the suite host. NOTE: this is
                         retrieved automatically if non-interactive ssh is
                         configured to the suite host.
  --use-ssh              Use ssh to re-invoke the command on the suite host.
  --no-login             Do not use a login shell to run remote ssh commands.
                         The default is to use a login shell.
  --comms-timeout=SEC,  --pyro-timeout=SEC
                         Set a timeout for network connections to the running
                         suite. The default is no timeout. For task messaging
                         connections see site/user config file documentation.
```

```
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
-f, --force             Do not ask for confirmation before acting. Note that
                        it is not necessary to use this option if interactive
                        command prompts have been disabled in the site/user
                        config files.
-m, --family            (Obsolete) This option is now ignored and is retained
                        for backward compatibility only. TASKID in the
                        argument list can be used to match task and family
                        names regardless of this option.
--no-multitask-compat
                        Disallow backward compatible multitask interface.
```

E.2.52   reset

Usage: cylc [control] reset [OPTIONS] REG [TASKID ...]

Force one or more task proxies in a running suite to change state and modify
their prerequisites and outputs accordingly. For example, --state=waiting
means "prerequisites not satisfied, outputs not completed"; --state=ready means
"prerequisites satisfied, outputs not completed" (this generally has the same
effect as using the "cylc trigger" command).

"cylc reset --state=spawn" is deprecated: use "cylc spawn" instead.

See the documentation for the -s/--state option for legal reset states.
A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
   REG   Suite name
   [TASKID ...]   Task identifiers

Options:
  -h, --help   show this help message and exit
  -s STATE, --state=STATE
                          Reset task state to STATE, can be succeeded, failed,
                          held, waiting, ready, expired
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
```

```
--no-login              Do not use a login shell to run remote ssh commands.
                        The default is to use a login shell.
--comms-timeout=SEC, --pyro-timeout=SEC
                        Set a timeout for network connections to the running
                        suite. The default is no timeout. For task messaging
                        connections see site/user config file documentation.
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
-f, --force             Do not ask for confirmation before acting. Note that
                        it is not necessary to use this option if interactive
                        command prompts have been disabled in the site/user
                        config files.
-m, --family            (Obsolete) This option is now ignored and is retained
                        for backward compatibility only. TASKID in the
                        argument list can be used to match task and family
                        names regardless of this option.
--no-multitask-compat
                        Disallow backward compatible multitask interface.
```

E.2.53   restart

Usage: cylc [control] restart [OPTIONS] REG

Start a suite run from the previous state. To start from scratch (cold or warm start) see the 'cylc run' command.

The scheduler runs in daemon mode unless you specify n/--no-detach or --debug.

Tasks recorded as submitted or running are polled at start-up to determine what happened to them while the suite was down.

Arguments:
```
   REG                  Suite name
```

Options:
```
 -h, --help             show this help message and exit
 --non-daemon           (deprecated: use --no-detach)
 -n, --no-detach        Do not daemonize the suite
 -a, --no-auto-shutdown
                        Do not shut down the suite automatically when all
                        tasks have finished. This flag overrides the
                        corresponding suite config item.
 --profile              Output profiling (performance) information
 --checkpoint=CHECKPOINT-ID
                        Specify the ID of a checkpoint to restart from
 --ignore-final-cycle-point
                        Ignore the final cycle point in the suite run
                        database. If one is specified in the suite definition
                        it will be used, however.
 --ignore-initial-cycle-point
                        Ignore the initial cycle point in the suite run
                        database. If one is specified in the suite definition
                        it will be used, however.
 --until=CYCLE_POINT    Shut down after all tasks have PASSED this cycle
                        point.
 --hold                 Hold (don't run tasks) immediately on starting.
 --hold-after=CYCLE_POINT
                        Hold (don't run tasks) AFTER this cycle point.
 -m STRING, --mode=STRING
                        Run mode: live, dummy, dummy-local, simulation
                        (default live).
 --reference-log        Generate a reference log for use in reference tests.
 --reference-test       Do a test run against a previously generated reference
                        log.
 -S SOURCE, --source=SOURCE
                        Specify the suite source.
```

```
--user=USER          Other user account name. This results in command
                     reinvocation on the remote account.
--host=HOST          Other host name. This results in command reinvocation
                     on the remote account.
-v, --verbose        Verbose output mode.
--debug              Run suites in non-daemon mode, and show exception
                     tracebacks.
-s NAME=VALUE, --set=NAME=VALUE
                     Set the value of a Jinja2 template variable in the
                     suite definition. This option can be used multiple
                     times on the command line. NOTE: these settings
                     persist across suite restarts, but can be set again on
                     the "cylc restart" command line if they need to be
                     overridden.
--set-file=FILE      Set the value of Jinja2 template variables in the
                     suite definition from a file containing NAME=VALUE
                     pairs (one per line). NOTE: these settings persist
                     across suite restarts, but can be set again on the
                     "cylc restart" command line if they need to be
                     overridden.
```

E.2.54   run

```
Usage: cylc [control] run|start [OPTIONS] REG [START_POINT]
```

Start a suite run from scratch, wiping out any previous suite state. To
restart from a previous state see 'cylc restart --help'.

The scheduler runs in daemon mode unless you specify --no-detach or --debug.

Any dependence on cycle points earlier than the start cycle point is ignored.

A "cold start" (the default) starts from the suite initial cycle point
(specified in the suite.rc or on the command line). Any dependence on tasks
prior to the suite initial cycle point is ignored.

A "warm start" (-w/--warm) starts from a given cycle point later than the suite
initial cycle point (specified in the suite.rc). Any dependence on tasks prior
to the given warm start cycle point is ignored. The suite initial cycle point
is preserved.

```
Arguments:
   REG                      Suite name
   [START_POINT]            Initial cycle point or 'now';
                            overrides the suite definition.

Options:
  -h, --help            show this help message and exit
  --non-daemon          (deprecated: use --no-detach)
  -n, --no-detach       Do not daemonize the suite
  -a, --no-auto-shutdown
                        Do not shut down the suite automatically when all
                        tasks have finished. This flag overrides the
                        corresponding suite config item.
  --profile             Output profiling (performance) information
  -w, --warm            Warm start the suite. The default is to cold start.
  --ict                 Does nothing, option for backward compatibility only
  --until=CYCLE_POINT    Shut down after all tasks have PASSED this cycle
                        point.
  --hold                Hold (don't run tasks) immediately on starting.
  --hold-after=CYCLE_POINT
                        Hold (don't run tasks) AFTER this cycle point.
  -m STRING, --mode=STRING
                        Run mode: live, dummy, dummy-local, simulation
                        (default live).
  --reference-log       Generate a reference log for use in reference tests.
  --reference-test      Do a test run against a previously generated reference
                        log.
  -S SOURCE, --source=SOURCE
                        Specify the suite source.
  --user=USER          Other user account name. This results in command
```

```
                        reinvocation on the remote account.
  --host=HOST           Other host name. This results in command reinvocation
                        on the remote account.
  -v, --verbose         Verbose output mode.
  --debug               Run suites in non-daemon mode, and show exception
                        tracebacks.
  -s NAME=VALUE, --set=NAME=VALUE
                        Set the value of a Jinja2 template variable in the
                        suite definition. This option can be used multiple
                        times on the command line. NOTE: these settings
                        persist across suite restarts, but can be set again on
                        the "cylc restart" command line if they need to be
                        overridden.
  --set-file=FILE       Set the value of Jinja2 template variables in the
                        suite definition from a file containing NAME=VALUE
                        pairs (one per line). NOTE: these settings persist
                        across suite restarts, but can be set again on the
                        "cylc restart" command line if they need to be
                        overridden.
```

E.2.55   scan

Usage: cylc [discovery] scan [OPTIONS] [HOSTS ...]

Print information about cylc suites currently running on scanned hosts. The
list of hosts to scan is determined by the global configuration "[suite host
scanning]" setting, or hosts can be specified explicitly on the command line.

By default, just your own suites are listed (this assumes your username is the
same on all scanned hosts). Use -a/--all-suites to see all suites on all hosts,
or restrict suites displayed with the -o/--owner and -n/--name options (with
--name the default owner restriction (i.e. just your own suites) is disabled.

Suite passphrases are not needed to get identity information (name and owner)
from suites running cylc >= 6.6.0. Titles, descriptions, state totals, and
cycle point state totals may also be revealed publicly, depending on global
and sutie authentication settings. Suite passphrases still grant full access
regardless of what is revealed publicly.

Passphrases are still required to get identity information from older suites
(cylc <= 6.5.0), otherwise you'll see "connection denied (security reasons)".

WARNING: a suite suspended with Ctrl-Z will cause port scans to hang until the
connection times out (see --comms-timeout).

Arguments:
   [HOSTS ...]   Hosts to scan instead of the configured hosts.

Options:
  -h, --help   show this help message and exit
  -a, --all, --all-suites
                        List all suites found on all scanned hosts (the
                        default is just your own suites).
  -n PATTERN, --name=PATTERN
                        List suites with name matching PATTERN (regular
                        expression). Defaults to any name. Can be used
                        multiple times.
  -o PATTERN, --suite-owner=PATTERN
                        List suites with owner matching PATTERN (regular
                        expression). Defaults to just your own suites. Can be
                        used multiple times.
  -d, --describe   Print suite titles and descriptions if available.
  -s, --state-totals   Print number of tasks in each state if available
                        (total, and by cycle point).
  -f, --full   Print all available information about each suite.
  -c, --color, --colour
                        Print task state summaries using terminal color
                        control codes.
  -b, --no-bold   Don't use any bold text in the command output.
  --print-ports   Print the port range from the site config file
                        ($CYLC_DIR/conf/global.rc).

```
--comms-timeout=SEC    Set a timeout for network connections to each running
                       suite. The default is 5 seconds.
--old, --old-format    Legacy output format ("suite owner host port").
-r, --raw, --raw-format
                       Parsable format ("suite|owner|host|property|value")
--user=USER            Other user account name. This results in command
                       reinvocation on the remote account.
--host=HOST            Other host name. This results in command reinvocation
                       on the remote account.
-v, --verbose          Verbose output mode.
--debug                Run suites in non-daemon mode, and show exception
                       tracebacks.
--port=INT             Suite port number on the suite host. NOTE: this is
                       retrieved automatically if non-interactive ssh is
                       configured to the suite host.
--use-ssh              Use ssh to re-invoke the command on the suite host.
--no-login             Do not use a login shell to run remote ssh commands.
                       The default is to use a login shell.
--print-uuid           Print the client UUID to stderr. This can be matched
                       to information logged by the receiving suite daemon.
--set-uuid=UUID        Set the client UUID manually (e.g. from prior use of
                       --print-uuid). This can be used to log multiple
                       commands under the same UUID (but note that only the
                       first [info] command from the same client ID will be
                       logged unless the suite is running in debug mode).
```

E.2.56   scp-transfer

Usage: cylc [util] scp-transfer [OPTIONS]

An scp wrapper for transferring a list of files and/or directories
at once. The source and target scp URLs can be local or remote (scp
can transfer files between two remote hosts). Passwordless ssh must
be configured appropriately.

ENVIRONMENT VARIABLE INPUTS:
$SRCE - list of sources (files or directories) as scp URLs.
$DEST - parallel list of targets as scp URLs.
The source and destination lists should be space-separated.

We let scp determine the validity of source and target URLs.
Target directories are created pre-copy if they don't exist.

Options:
 -v   - verbose: print scp stdout.
 --help - print this usage message.

E.2.57   search

Usage: cylc [prep] search|grep [OPTIONS] SUITE PATTERN [PATTERN2...]

Search for pattern matches in suite definitions and any files in the
suite bin directory. Matches are reported by line number and suite
section. An unquoted list of PATTERNs will be converted to an OR'd
pattern. Note that the order of command line arguments conforms to
normal cylc command usage (suite name first) not that of the grep
command.

Note that this command performs a text search on the suite definition,
it does not search the data structure that results from parsing the
suite definition - so it will not report implicit default settings.

For case insenstive matching use '(?i)PATTERN'.

Arguments:
   SUITE   Suite name or path
   PATTERN   Python-style regular expression
   [PATTERN2...]   Additional search patterns

Options:

```
-h, --help              show this help message and exit
-x                      Do not search in the suite bin directory
--user=USER             Other user account name. This results in command
                        reinvocation on the remote account.
--host=HOST             Other host name. This results in command reinvocation
                        on the remote account.
-v, --verbose           Verbose output mode.
--debug                 Run suites in non-daemon mode, and show exception
                        tracebacks.
--suite-owner=OWNER     Specify suite owner
```

E.2.58   set-runahead

Usage: cylc [control] set-runahead [OPTIONS] REG [HOURS]

Change the suite runahead limit in a running suite. This is the number of
hours that the fastest task is allowed to get ahead of the slowest. If a
task spawns beyond that limit it will be held back from running until the
slowest tasks catch up enough. WARNING: if you omit HOURS no runahead
limit will be set - DO NOT DO THIS for for any cycling suite that has
no near stop cycle set and is not constrained by clock-trigger tasks.

Arguments:
```
   REG                     Suite name
   [HOURS]                 Runahead limit (default: no limit)
```

Options:
```
  -h, --help              show this help message and exit
  --user=USER             Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST             Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose           Verbose output mode.
  --debug                 Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT              Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh               Use ssh to re-invoke the command on the suite host.
  --no-login              Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                          Set a timeout for network connections to the running
                          suite. The default is no timeout. For task messaging
                          connections see site/user config file documentation.
  --print-uuid            Print the client UUID to stderr. This can be matched
                          to information logged by the receiving suite daemon.
  --set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                          --print-uuid). This can be used to log multiple
                          commands under the same UUID (but note that only the
                          first [info] command from the same client ID will be
                          logged unless the suite is running in debug mode).
  -f, --force             Do not ask for confirmation before acting. Note that
                          it is not necessary to use this option if interactive
                          command prompts have been disabled in the site/user
                          config files.
```

E.2.59   set-verbosity

Usage: cylc [control] set-verbosity [OPTIONS] REG LEVEL

Change the logging priority level of a running suite. Only messages at
or above the chosen priority level will be logged; for example, if you
choose WARNING, only warnings and critical messages will be logged.

Arguments:
```
   REG                     Suite name
   LEVEL                   INFO, WARNING, NORMAL, CRITICAL, ERROR, DEBUG
```

Options:

```
   -h, --help              show this help message and exit
   --user=USER             Other user account name. This results in command
                           reinvocation on the remote account.
   --host=HOST             Other host name. This results in command reinvocation
                           on the remote account.
   -v, --verbose           Verbose output mode.
   --debug                 Run suites in non-daemon mode, and show exception
                           tracebacks.
   --port=INT              Suite port number on the suite host. NOTE: this is
                           retrieved automatically if non-interactive ssh is
                           configured to the suite host.
   --use-ssh               Use ssh to re-invoke the command on the suite host.
   --no-login              Do not use a login shell to run remote ssh commands.
                           The default is to use a login shell.
   --comms-timeout=SEC, --pyro-timeout=SEC
                           Set a timeout for network connections to the running
                           suite. The default is no timeout. For task messaging
                           connections see site/user config file documentation.
   --print-uuid            Print the client UUID to stderr. This can be matched
                           to information logged by the receiving suite daemon.
   --set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                           --print-uuid). This can be used to log multiple
                           commands under the same UUID (but note that only the
                           first [info] command from the same client ID will be
                           logged unless the suite is running in debug mode).
   -f, --force             Do not ask for confirmation before acting. Note that
                           it is not necessary to use this option if interactive
                           command prompts have been disabled in the site/user
                           config files.
```

E.2.60   show

Usage: cylc [info] show [OPTIONS] REG [TASKID ...]

Interrogate a suite daemon for the suite title and description; or for the title and description of one of its tasks; or for the current state of the prerequisites, outputs, and clock-triggering of a specific task instance. A TASKID is an identifier for matching individual task proxies and/or families of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To avoid this, use the '--no-multitask-compat' option, or use the new syntax (with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
```
   REG                     Suite name
   [TASKID ...]            Task names or identifiers
```

Options:
```
   -h, --help              show this help message and exit
   --list-prereqs          Print a task's pre-requisites as a list.
   --user=USER             Other user account name. This results in command
                           reinvocation on the remote account.
   --host=HOST             Other host name. This results in command reinvocation
                           on the remote account.
   -v, --verbose           Verbose output mode.
   --debug                 Run suites in non-daemon mode, and show exception
                           tracebacks.
```

```
--port=INT              Suite port number on the suite host. NOTE: this is
                        retrieved automatically if non-interactive ssh is
                        configured to the suite host.
--use-ssh               Use ssh to re-invoke the command on the suite host.
--no-login              Do not use a login shell to run remote ssh commands.
                        The default is to use a login shell.
--comms-timeout=SEC, --pyro-timeout=SEC
                        Set a timeout for network connections to the running
                        suite. The default is no timeout. For task messaging
                        connections see site/user config file documentation.
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
-m, --family            (Obsolete) This option is now ignored and is retained
                        for backward compatibility only. TASKID in the
                        argument list can be used to match task and family
                        names regardless of this option.
--no-multitask-compat
                        Disallow backward compatible multitask interface.
```

E.2.61   spawn

Usage: cylc [control] spawn [OPTIONS] REG [TASKID ...]

```
Force one or more task proxies to spawn successors at the next cycle point
in their sequences.  This is useful if you need to run successive instances
of a task out of order.

A TASKID is an identifier for matching individual task proxies and/or families
of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To
avoid this, use the '--no-multitask-compat' option, or use the new syntax
(with a '/' or a '.') when specifying 2 TASKID arguments.
```

Arguments:
```
  REG                      Suite name
  [TASKID ...]             Task identifiers
```

Options:
```
  -h, --help              show this help message and exit
  --user=USER             Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST             Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose           Verbose output mode.
  --debug                 Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT              Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh               Use ssh to re-invoke the command on the suite host.
  --no-login              Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
```

```
--comms-timeout=SEC, --pyro-timeout=SEC
                        Set a timeout for network connections to the running
                        suite. The default is no timeout. For task messaging
                        connections see site/user config file documentation.
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
-f, --force             Do not ask for confirmation before acting. Note that
                        it is not necessary to use this option if interactive
                        command prompts have been disabled in the site/user
                        config files.
-m, --family            (Obsolete) This option is now ignored and is retained
                        for backward compatibility only. TASKID in the
                        argument list can be used to match task and family
                        names regardless of this option.
--no-multitask-compat
                        Disallow backward compatible multitask interface.
```

E.2.62 stop

Usage: cylc [control] stop|shutdown [OPTIONS] REG [STOP]

Tell a running suite daemon to shut down. In order to prevent failures going
unnoticed, suites only shut down automatically at a final cycle point if no
failed tasks are present. There are several shutdown methods:

1. (default) stop after current active tasks finish
2. (--now) stop immediately, orphaning current active tasks
3. (--kill) stop after killing current active tasks
4. (with STOP as a cycle point) stop after cycle point STOP
5. (with STOP as a task ID) stop after task ID STOP has succeeded
6. (--wall-clock=T) stop after time T (an ISO 8601 date-time format e.g.
   CCYYMMDDThh:mm, CCYY-MM-DDThh, etc).

Tasks that become ready after the shutdown is ordered will be submitted
immediately if the suite is restarted. Remaining task event handlers and job
poll and kill commands, however, will be executed prior to shutdown, unless
--now is used.

This command exits immediately unless --max-polls is greater than zero, in
which case it polls to wait for suite shutdown.

Arguments:
```
   REG                     Suite name
   [STOP]                  a/ task POINT (cycle point), or
                               b/ ISO 8601 date-time (clock time), or
                               c/ TASK (task ID).
```

Options:
```
  -h, --help              show this help message and exit
  -k, --kill              Shut down after killing currently active tasks.
  -n, --now               Shut down without waiting for active tasks to
                          complete. If this option is specified once, wait for
                          task event handler, job poll/kill to complete. If this
                          option is specified more than once, tell the suite to
                          terminate immediately.
  -w STOP, --wall-clock=STOP
                          Shut down after time STOP (ISO 8601 formatted)
  --max-polls=INT         Maximum number of polls (default 0).
  --interval=SECS         Polling interval in seconds (default 60).
  --user=USER             Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST             Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose           Verbose output mode.
  --debug                 Run suites in non-daemon mode, and show exception
                          tracebacks.
```

```
--port=INT              Suite port number on the suite host. NOTE: this is
                        retrieved automatically if non-interactive ssh is
                        configured to the suite host.
--use-ssh               Use ssh to re-invoke the command on the suite host.
--no-login              Do not use a login shell to run remote ssh commands.
                        The default is to use a login shell.
--comms-timeout=SEC, --pyro-timeout=SEC
                        Set a timeout for network connections to the running
                        suite. The default is no timeout. For task messaging
                        connections see site/user config file documentation.
--print-uuid            Print the client UUID to stderr. This can be matched
                        to information logged by the receiving suite daemon.
--set-uuid=UUID         Set the client UUID manually (e.g. from prior use of
                        --print-uuid). This can be used to log multiple
                        commands under the same UUID (but note that only the
                        first [info] command from the same client ID will be
                        logged unless the suite is running in debug mode).
-f, --force             Do not ask for confirmation before acting. Note that
                        it is not necessary to use this option if interactive
                        command prompts have been disabled in the site/user
                        config files.
```

E.2.63   submit

```
Usage: cylc [task] submit|single [OPTIONS] REG TASK
```

Submit a single task to run just as it would be submitted by its suite. Task
messaging commands will print to stdout but will not attempt to communicate
with the suite (which does not need to be running).

For tasks present in the suite graph the given cycle point is adjusted up to
the next valid cycle point for the task. For tasks defined under runtime but
not present in the graph, the given cycle point is assumed to be valid.

WARNING: do not 'cylc submit' a task that is running in its suite at the
same time - both instances will attempt to write to the same job logs.

```
Arguments:
   REG                  Suite name
   TASK                 Target task (NAME.CYCLE_POINT)

Options:
  -h, --help            show this help message and exit
  -d, --dry-run         Generate the job script for the task, but don't submit
                        it.
  --user=USER           Other user account name. This results in command
                        reinvocation on the remote account.
  --host=HOST           Other host name. This results in command reinvocation
                        on the remote account.
  -v, --verbose         Verbose output mode.
  --debug               Run suites in non-daemon mode, and show exception
                        tracebacks.
  -s NAME=VALUE, --set=NAME=VALUE
                        Set the value of a Jinja2 template variable in the
                        suite definition. This option can be used multiple
                        times on the command line. NOTE: these settings
                        persist across suite restarts, but can be set again on
                        the "cylc restart" command line if they need to be
                        overridden.
  --set-file=FILE       Set the value of Jinja2 template variables in the
                        suite definition from a file containing NAME=VALUE
                        pairs (one per line). NOTE: these settings persist
                        across suite restarts, but can be set again on the
                        "cylc restart" command line if they need to be
                        overridden.
  --icp=CYCLE_POINT     Set initial cycle point. Required if not defined in
                        suite.rc.
```

E.2.64   suite-state

Usage: cylc suite-state REG [OPTIONS]

Print task states retrieved from a suite database; or (with --task,
--point, and --status) poll until a given task reaches a given state; or (with
--task, --point, and --message) poll until a task receives a given message.
Polling is configurable with --interval and --max-polls; for a one-off
check use --max-polls=1. The suite database does not need to exist at
the time polling commences but allocated polls are consumed waiting for
it (consider max-polls*interval as an overall timeout).

Note for non-cycling tasks --point=1 must be provided.

Important: cylc suite-state only works with task states and does not work with
task messages.

For your own suites the database location is determined by your
site/user config. For other suites, e.g. those owned by others, or
mirrored suite databases, use --run-dir=DIR to specify the location.

Example usages:
  cylc suite-state REG --task=TASK --point=POINT --status=STATUS
returns 0 if TASK.POINT reaches STATUS before the maximum number of
polls, otherwise returns 1.

  cylc suite-state REG --task=TASK --point=POINT --status=STATUS --offset=PT6H
adds 6 hours to the value of CYCLE for carrying out the polling operation.

  cylc suite-state REG --task=TASK --status=STATUS --task-point
uses CYLC_TASK_CYCLE_POINT environment variable as the value for the CYCLE
to poll. This is useful when you want to use cylc suite-state in a cylc task.

Arguments:
   REG   Suite name

Options:
  -h, --help   show this help message and exit
  -t TASK, --task=TASK Specify a task to check the state of.
  -p CYCLE, --point=CYCLE
                           Specify the cycle point to check task states for.
  -T, --task-point   Use the CYLC_TASK_CYCLE_POINT environment variable as
                           the cycle point to check task states for. Shorthand
                           for --point=$CYLC_TASK_CYCLE_POINT
  --template=TEMPLATE   Cyclepoint template string, used to reformat
                           cyclepoints for querying suites
  -d DIR, --run-dir=DIR
                           The top level cylc run directory if non-standard. The
                           database should be DIR/REG/log/db. Use to interrogate
                           suites owned by others, etc.; see note above.
  -s OFFSET, --offset=OFFSET
                           Specify an offset to add to the targeted cycle point
  -S STATUS, --status=STATUS
                           Specify a particular status or triggering condition to
                           check for. Valid triggering conditions to check for
                           include: 'fail', 'finish', 'start', 'submit' and
                           'succeed'. Valid states to check for include:
                           'runahead', 'waiting', 'held', 'queued', 'ready',
                           'expired', 'submitted', 'submit-failed', 'submit-
                           retrying', 'running', 'succeeded', 'failed' and
                           'retrying'.
  -m MSG, --message=MSG
                           Specify a particular message to check for.
  --max-polls=INT   Maximum number of polls (default 10).
  --interval=SECS   Polling interval in seconds (default 60).
  --user=USER   Other user account name. This results in command
                           reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                           on the remote account.
  -v, --verbose   Verbose output mode.

E.2.65   test-battery

Usage: cylc test-battery [...]

Run automated cylc and parsec tests under [FILES or DIRECTORIES].
Test locations default to the following directory tree:
  /home/vagrant/cylc/tests/

Some tests (e.g. those specific to particular batch schedulers) can be
configured in your site/user config file. A few others still submit jobs
to a user@host account taken from the environment:
  $CYLC_TEST_TASK_HOST # default localhost
  $CYLC_TEST_TASK_OWNER # defaut $USER

Requirements:
  * Passwordless ssh must be configured to task host accounts.
  * Some test suites submit jobs to 'at' so atd must be running.

Options and arguments are appended to the "prove -j $NPROC -s -r ${@:-tests}"
command, where NPROC is the number of child processes that can be used to run
the test files.

Some tests use a clean global config file. If some items from your site config
file are needed in this, e.g. to get remote test hosts working, add them to
/home/vagrant/cylc/conf/global-tests.rc.

The command normally uses the "process pool size" setting (default=4) in the
site/user global configuration file to determine the number of tests to run in
parallel. You can also change the amount of concurrency with the "-j N" option.

Suite run directories are cleaned up on the suite host for passing tests -
otherwise they are left alone.

To output stderr from failed tests to the terminal,
"export CYLC_TEST_DEBUG=true" before running this command.

The command normally uses "diff -u" to compare files. However, if an alternate
command such as "xxdiff -D" is desirable (e.g. for debugging), "export
CYLC_TEST_DIFF_CMD=xxdiff -D".

Commits or Pull Requests to cylc/cylc on GitHub will trigger Travis CI to run
generic (non platform-specific) tests - see /home/vagrant/cylc/.travis.yml.
After enabling Travis CI for your own cylc fork, you can skip generic tests
locally by setting CYLC_TEST_RUN_GENERIC=false.

By default all tests are executed. To run just a subset of them:
  * list individual tests or test directories to run on the comand line
  * list individual tests or test directories to skip in $CYLC_TEST_SKIP
  * skip all generic tests with CYLC_TEST_RUN_GENERIC=false
  * skip all platform-specific tests with CYLC_TEST_RUN_PLATFORM=false
  List specific tests relative to /home/vagrant/cylc (i.e. starting with "test/
     ").
Some platform-specific tests are automatically skipped, depending on platform.

FOR DEVELOPERS:
  * Platform-specific tests must set "CYLC_TEST_IS_GENERIC=false" before
    sourcing the test_header.
  * Tests requiring the sqlite3 CLI must be skipped if sqlite3 is not installed
    (it is not otherwise a Cylc software prerequisite):
| if ! which sqlite3 > /dev/null; then
| # Skip the remaining 3 tests.
| skip 3 "sqlite3 not installed?"
| purge_suite $SUITE_NAME
| exit 0
| fi

For more information see "Reference Tests" in the User Guide.

Options:
  -h, --help   Print this help message and exit.

Examples:

Run the full test suite with the default options.

```
   cylc test-battery
Run the full test suite with 12 processes.
   cylc test-battery -j 12
Run only tests under "tests/cyclers/" with 12 processes.
   cylc test-battery -j 12 tests/cyclers
Run only "tests/cyclers/16-weekly.t" in verbose mode
   cylc test-battery -v tests/cyclers/16-weekly.t
Run only tests under "tests/cyclers/" with 12 processes, and skip 00-daily.t
   export CYLC_TEST_SKIP=tests/cyclers/00-daily.t
   cylc test-battery -j 12 tests/cyclers
```

E.2.66   trigger

Usage: cylc [control] trigger [OPTIONS] REG [TASKID ...]

Manually trigger one or more tasks. Waiting tasks will be queued (cylc internal queues) and will submit as normal when released by the queue; queued tasks will submit immediately even if that violates the queue limit (so you may need to trigger a queue-limited task twice to get it to submit).

For single tasks you can use "--edit" to edit the generated job script before it submits, to apply one-off changes. A diff between the original and edited job script will be saved to the task job log directory.

A TASKID is an identifier for matching individual task proxies and/or families of them. It can be written in these syntaxes:
* [CYCLE-POINT-GLOB/]TASK-NAME-GLOB[:TASK-STATE]
* [CYCLE-POINT-GLOB/]FAMILY-NAME-GLOB[:TASK-STATE]
* TASK-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]
* FAMILY-NAME-GLOB[.CYCLE-POINT-GLOB][:TASK-STATE]

For example, to match:
* all tasks in a cycle: '20200202T0000Z/*' or '*.20200202T0000Z'
* all tasks in the submitted status: ':submitted'
* retrying 'foo*' tasks in 0000Z cycles: 'foo*.*0000Z:retrying' or
  '*0000Z/foo*:retrying'
* retrying tasks in 'BAR' family: '*/BAR:retrying' or 'BAR.*:retrying'
* retrying tasks in 'BAR' or 'BAZ' families: '*/BA[RZ]:retrying' or
  'BA[RZ].*:retrying'

The old 'MATCH POINT' syntax will be automatically detected and supported. To avoid this, use the '--no-multitask-compat' option, or use the new syntax (with a '/' or a '.') when specifying 2 TASKID arguments.

Arguments:
   REG   Suite name
    [TASKID ...]   Task identifiers

Options:
  -h, --help   show this help message and exit
  -e, --edit   Manually edit the job script before running it.
  -g, --geditor   (with --edit) force use of the configured GUI editor.
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --port=INT   Suite port number on the suite host. NOTE: this is
                          retrieved automatically if non-interactive ssh is
                          configured to the suite host.
  --use-ssh   Use ssh to re-invoke the command on the suite host.
  --no-login   Do not use a login shell to run remote ssh commands.
                          The default is to use a login shell.
  --comms-timeout=SEC, --pyro-timeout=SEC
                          Set a timeout for network connections to the running
                          suite. The default is no timeout. For task messaging
                          connections see site/user config file documentation.
  --print-uuid   Print the client UUID to stderr. This can be matched
                          to information logged by the receiving suite daemon.
```

```
--set-uuid=UUID           Set the client UUID manually (e.g. from prior use of
                          --print-uuid). This can be used to log multiple
                          commands under the same UUID (but note that only the
                          first [info] command from the same client ID will be
                          logged unless the suite is running in debug mode).
-f, --force               Do not ask for confirmation before acting. Note that
                          it is not necessary to use this option if interactive
                          command prompts have been disabled in the site/user
                          config files.
-m, --family              (Obsolete) This option is now ignored and is retained
                          for backward compatibility only. TASKID in the
                          argument list can be used to match task and family
                          names regardless of this option.
--no-multitask-compat
                          Disallow backward compatible multitask interface.
```

E.2.67   upgrade-run-dir

Usage: cylc [admin] upgrade-run-dir SUITE

For one-off conversion of a suite run directory to cylc-6 format.

Arguments:
    SUITE   suite name or run directory path

Options:
  -h, --help   show this help message and exit

E.2.68   validate

Usage: cylc [prep] validate [OPTIONS] SUITE

Validate a suite definition against the official specification
files held in $CYLC_DIR/conf/suiterc/.

If the suite definition uses include-files reported line numbers
will correspond to the inlined version seen by the parser; use
'cylc view -i,--inline SUITE' for comparison.

Arguments:
    SUITE   Suite name or path

Options:
  -h, --help   show this help message and exit
  --strict   Fail any use of unsafe or experimental features.
                          Currently this just means naked dummy tasks (tasks
                          with no corresponding runtime section) as these may
                          result from unintentional typographic errors in task
                          names.
  -o FILENAME, --output=FILENAME
                          Specify a file name to dump the processed suite.rc.
  --profile   Output profiling (performance) information
  -u RUN_MODE, --run-mode=RUN_MODE
                          Validate for run mode.
  --user=USER   Other user account name. This results in command
                          reinvocation on the remote account.
  --host=HOST   Other host name. This results in command reinvocation
                          on the remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception
                          tracebacks.
  --suite-owner=OWNER   Specify suite owner
  -s NAME=VALUE, --set=NAME=VALUE
                          Set the value of a Jinja2 template variable in the
                          suite definition. This option can be used multiple
                          times on the command line. NOTE: these settings
                          persist across suite restarts, but can be set again on
                          the "cylc restart" command line if they need to be
                          overridden.
  --set-file=FILE   Set the value of Jinja2 template variables in the
```

```
                         suite definition from a file containing NAME=VALUE
                         pairs (one per line). NOTE: these settings persist
                         across suite restarts, but can be set again on the
                         "cylc restart" command line if they need to be
                         overridden.
  --icp=CYCLE_POINT      Set initial cycle point. Required if not defined in
                         suite.rc.
```

E.2.69   version

Usage: cylc [info] version

Print the cylc version invoked at the command line.

Note that "cylc -v,--version" just prints the version string from the main
command interface, whereas this is a proper cylc command that can take the
standard --host and --user options, etc.

For the cylc version of running a suite daemon see
  "cylc get-suite-version".

Arguments:

Options:
  -h, --help   show this help message and exit
  --long   Print the path to the current cylc version
  --user=USER   Other user account name. This results in command reinvocation
                  on the remote account.
  --host=HOST   Other host name. This results in command reinvocation on the
                  remote account.
  -v, --verbose   Verbose output mode.
  --debug   Run suites in non-daemon mode, and show exception tracebacks.

E.2.70   view

Usage: cylc [prep] view [OPTIONS] SUITE

View a read-only temporary copy of suite NAME's suite.rc file, in your
editor, after optional include-file inlining and Jinja2 preprocessing.

The edit process is spawned in the foreground as follows:
  % <editor> suite.rc
Where <editor> is defined in the cylc site and user config files
($CYLC_DIR/conf/global.rc and $HOME/.cylc/global.rc).

For remote host or owner, the suite will be printed to stdout unless
the '-g,--gui' flag is used to spawn a remote GUI edit session.

See also 'cylc [prep] edit'.

Arguments:
   SUITE   Suite name or path

Options:
  -h, --help   show this help message and exit
  -i, --inline   Inline include-files.
  -j, --jinja2   View after Jinja2 template processing (implies
                          '-i/--inline' as well).
  -p, --process   View after all processing (Jinja2, inlining, line-
                          continuation joining).
  -m, --mark   (With '-i') Mark inclusions in the left margin.
  -l, --label   (With '-i') Label file inclusions with the file name.
                          Line numbers will not correspond to those reported by
                          the parser.
  --single   (With '-i') Inline only the first instances of any
                          multiply-included files. Line numbers will not
                          correspond to those reported by the parser.
  -c, --cat   Concatenate continuation lines (line numbers will not
                          correspond to those reported by the parser).
  -g, --gui   Force use of the configured GUI editor.
```

```
--stdout              Print the suite definition to stdout.
--mark-for-edit       (With '-i') View file inclusion markers as for 'cylc
                      edit --inline'.
--user=USER           Other user account name. This results in command
                      reinvocation on the remote account.
--host=HOST           Other host name. This results in command reinvocation
                      on the remote account.
-v, --verbose         Verbose output mode.
--debug               Run suites in non-daemon mode, and show exception
                      tracebacks.
--suite-owner=OWNER   Specify suite owner
-s NAME=VALUE, --set=NAME=VALUE
                      Set the value of a Jinja2 template variable in the
                      suite definition. This option can be used multiple
                      times on the command line. NOTE: these settings
                      persist across suite restarts, but can be set again on
                      the "cylc restart" command line if they need to be
                      overridden.
--set-file=FILE       Set the value of Jinja2 template variables in the
                      suite definition from a file containing NAME=VALUE
                      pairs (one per line). NOTE: these settings persist
                      across suite restarts, but can be set again on the
                      "cylc restart" command line if they need to be
                      overridden.
```

E.2.71   warranty

```
Usage: cylc [license] warranty [--help]
   Cylc is released under the GNU General Public License v3.0
This command prints the GPL v3.0 disclaimer of warranty.
Options:
  --help   Print this usage message.
```

F   The gcylc Graph View

The graph view in the gcylc GUI shows the structure of the suite as it evolves. It can work well even for large suites, but be aware that the graphviz layout engine has to do a new global layout every time a task proxy appears in or disappears from the task pool. The following may help mitigate any jumping layout problems:

- The disconnect button can be used to temporarily prevent the graph from changing as the suite evolves.
- The greyed-out base nodes, which are only present to fill out the graph structure, can be toggled off (but this will split the graph into disconnected sub-trees).
- Right-click on a task and choose the "Focus" option to restrict the graph display to that task's cycle point. Anything interesting happening in other cycle points will show up as disconnected rectangular nodes to the right of the graph (and you can click on those to instantly refocus to their cycle points).
- Task filtering is the ultimate quick route to focusing on just the tasks you're interested in, but this will destroy the graph structure.

G   Cylc README File

```
# The Cylc Workflow Engine

[![Build Status](https://travis-ci.org/cylc/cylc.svg?branch=master)](https://
   travis-ci.org/cylc/cylc)
[![DOI](https://zenodo.org/badge/1836229.svg)](https://zenodo.org/badge/
   latestdoi/1836229)
```

Cylc (âĂśilkâĂİ) orchestrates complex distributed suites of interdependent
    cycling
tasks. It was originally designed for environmental forecasting systems at
[NIWA](https://www.niwa.co.nz).

Copyright and Terms of Use

Copyright (C) 2008-2017 NIWA

Cylc is free software: you can redistribute it and/or modify it under the terms
of the GNU General Public License as published by the Free Software Foundation,
either version 3 of the License, or (at your option) any later version.

Cylc is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with
cylc. If not, see [GNU licenses](http://www.gnu.org/licenses/).

Cylc Documentation
 * See [The Cylc Home Page](https://cylc.github.io/cylc)

Code Contributors
<!-- git shortlog -s -n -->
 * Hilary Oliver
 * Matt Shin
 * Ben Fitzpatrick
 * Andrew Clark
 * Oliver Sanders
 * Luis Kornblueh
 * Kerry Day
 * David Matthews
 * Tim Whitcomb
 * Scott Wales
 * Annette Osprey
 * Bruno Kinoshita
 * Domingo Manubens Gil
 * Jonny Williams
 * Alex Reinecke
 * Chan Wilson
 * Kevin Pulo

Acknowledgement For Non-Cylc Work:
Licences for non-cylc work included in this distribution can be found in the
`licences/` directory.
 * `lib/cherrypy/`:
    External software library released under a BSD license.
    Minor modification to ignore an import warning.
    See [cherrypy](http://www.cherrypy.org/).
 * `lib/isodatetime/`:
    Unmodified external software library released under the LGPL license.
    See [metomi/isodatetime](https://github.com/metomi/isodatetime/).
 * `lib/jinja2/`:
    External software library released under a BSD license.
    See [Jinja2](http://jinja.pocoo.org/).
 * `lib/markupsafe/`:
    External software library released under a BSD license, used by Jinja2.
    See [MarkupSafe](http://www.pocoo.org/projects/markupsafe/).
 * `lib/xdot.py`:
    External software released under the LGPL license.
    Modifications based on version 0.6. See
    [xdot](https://github.com/jrfonseca/xdot.py)

H   Cylc INSTALL File

Cylc Installation.

**See [The Cylc User Guide](https://cylc.github.io/cylc/documentation.html) for

detailed instructions.**

Cylc must be installed on suite and task job hosts, although the external
software packages (below) are not required on job hosts.

Required External Software Packages

These can be installed once on suite hosts updated infrequently.

 * graphviz
 * pygraphviz

Installing Cylc

Download the latest tarball from https://github.com/cylc/cylc/releases.

Successive Cylc releases should be installed side-by-side under a location such
as '/opt/cylc/':

```bash
cd /opt/cylc/
tar xzf cylc-7.2.1.tar.gz
# DO NOT CHANGE THE NAME OF THE UNPACKED CYLC SOURCE DIRECTORY.
cd cylc-7.2.1
export PATH=$PWD/bin:$PATH
make
```

When you type 'make':
  * A file called VERSION is created, containing the Cylc version number
    * The version number is taken from the name of the parent directory: DO NOT
      CHANGE THE NAME OF THE UNPACKED CYLC SOURCE DIRECTORY
  * The Cylc documentation is generated from source and put in doc/install/

Once installed, Cylc commands should be invoked via the supplied central
wrapper script that selects between the available versions. This allows
long-running suites (and their task jobs) to stick with older versions if
necessary. The wrapper should be edited to point to the Cylc install location:

```bash
cp /opt/cylc/cylc-7.2.1/admin/cylc-wrapper /usr/local/bin/cylc
# (now edit '/usr/local/bin/cylc' as per in-file instructions...)
```

Installing The Documentation

After running 'make' you can copy the entire 'doc/install' directory to a
convenient location such as '/var/www/html/', and update your Cylc site config
file to point to the intranet location.

Cloning The Cylc Repository

To participate in Cylc development fork [Cylc on
GitHub](https://github.com/cylc/cylc) and clone it locally. Changes should be
developed in feature branches then pushed to your GitHub fork before issuing a
Pull Request to the team. Please discuss proposed changes before you begin
work.

I   Cylc Development History - Major Changes

  * **pre-cylc-3** - early versions focused on the new scheduling algorithm. A suite was a col-
    lection of "task definition files" that encoded the prerequisites and outputs of each task,
    exposing cylc's self-organising nature. Tasks could be transferred from one suite to an-
    other by simply copying their taskdef files over and checking prerequisite and output con-
    sistency. Global suite structure was not easy to discern until run time (although cylc-2
    could generate resolved run time dependency graphs).

- **cylc-3** - a new suite design interface: dependency graph and task runtime properties defined in a single structured, validated, configuration file - the suite.rc file; graphical user interface; suite graphing.
- **cylc-4** - refined and organized the suite.rc file structure; task runtime properties defined by an efficient inheritance hierarchy; support for the Jinja2 template processor in suite definitions.
- **cylc-5** - multi-threading for continuous network request handling and job submission; more task states to distinguish job submission from execution; dependence between suites via new suite run databases; polling and killing of real task jobs; polling as task communications option.
- **cylc-6** - specification of all date-times and cycling workflows via the ISO8601 date-times, durations, and recurrence expressions; integer cycling; a multi-process pool to execute job submissions, event handlers, and poll and kill commands.
- **cylc-7** - Replaced the Pyro communications layer with RESTful HTTPS. Removed deprecated pre cylc-6 syntax and features.

J   Communication Method

Cylc suite daemons and clients (commands, cylc gui, task messaging) communicate via particular ports using the HTTPS protocol, secured by HTTP Digest Authentication using the suite's 20-random-character private passphrase and private SSL certificate.

This is enabled via the included-in-cylc cherrypy library (for the server) and either the Python requests library (if available) or the built-in Python libraries for the clients.

All suites are entirely isolated from one another.

K   Cylc 6 Migration Reference

Cylc 6 introduced new date-time-related syntax for the suite.rc file. In some places, this is quite radically different from the earlier syntax.

K.1   Timeouts and Delays

Timeouts and delays such as `[cylc][[events]]timeout` or `[runtime][[my_task]]retry delays` were written in a purely numeric form before cylc 6, in seconds, minutes (most common), or hours, depending on the setting.

They are now written in an ISO 8601 duration form, which has the benefit that the units are user-selectable (use 1 day instead of 1440 minutes) and explicit.

Nearly all timeouts and delays in cylc were in minutes, except for:
`[runtime][[my_task]][[[suite state polling]]]interval`
`[runtime][[my_task]][[[simulation mode]]]run time range`
which were in seconds, and
`[scheduling]runahead limit`
which was in hours (this is a special case discussed below in K.2).

See Table 1.

Table 1: Timeout/Delay Syntax Change Examples

Setting	Pre-Cylc-6	Cylc-6+
`[cylc][[events]]timeout`	180	PT3H
`[runtime][[my_task]]retry delays`	2*30, 360, 1440	2*PT30M, PT6H, P1D
`[runtime][[my_task]][[[suite state polling]]]interval`	2	PT2S

K.2   Runahead Limit

See A.3.7.

The `[scheduling]runahead limit` setting was written as a number of hours in pre-cylc-6 suites. This is now in ISO 8601 format for date-time cycling suites, so `[scheduling]runahead limit=36` would be written `[scheduling]runahead limit=PT36H`.

There is a new preferred alternative to `runahead limit`, `[scheduling]max active cycle points`. This allows the user to configure how many cycle points can run at once (default `3`). See A.3.8.

K.3   Cycle Time/Cycle Point

See A.3.2.

The following suite.rc settings have changed name (Table 2):

Table 2: Cycle Point Renaming

Pre-Cylc-6	Cylc-6+
`[scheduling]initial cycle time`	`[scheduling]initial cycle point`
`[scheduling]final cycle time`	`[scheduling]final cycle point`
`[visualization]initial cycle time`	`[visualization]initial cycle point`
`[visualization]final cycle time`	`[visualization]final cycle point`

This change is to reflect the fact that cycling in cylc 6+ can now be over e.g. integers instead of being purely based on date-time.

Date-times written in `initial cycle time` and `final cycle time` were in a cylc-specific 10-digit (or less) `CCYYMMDDhh` format, such as `2014021400` for 00:00 on the 14th of February 2014.

Date-times are now required to be ISO 8601 compatible. This can be achieved easily enough by inserting a `T` between the day and the hour digits.

Table 3: Cycle Point Syntax Example

Setting	Pre-Cylc-6	Cylc-6+
`[scheduling]initial cycle time`	2014021400	20140214T00

K.4   Cycling

Special *start-up* and *cold-start* tasks have been removed from cylc 6. Instead, use the initial/run-once notation as detailed in 7.23.2 and 9.3.4.6.

Repeating asynchronous tasks have also been removed because non date-time workflows can now be handled more easily with integer cycling. See for instance the satellite data processing example documented in 9.3.4.7.

For repeating tasks with hour-based cycling the syntax has only minor changes:

Pre-cylc-6:

```
[scheduling]
    ...
    [[dependencies]]
        [[[0,12]]]
            graph = foo[T-12] => foo & bar => baz

[scheduling]
    ...
    [[dependencies]]
        [[[T00,T12]]]
            graph = foo[-PT12H] => foo & bar => baz
```

Hour-based cycling section names are easy enough to convert, as seen in Table 4.

Table 4: Hourly Cycling Sections

Pre-Cylc-6	Cylc-6+
`[scheduling][[dependencies]][[[0]]]`	`[scheduling][[dependencies]][[[T00]]]`
`[scheduling][[dependencies]][[[6]]]`	`[scheduling][[dependencies]][[[T06]]]`
`[scheduling][[dependencies]][[[12]]]`	`[scheduling][[dependencies]][[[T12]]]`
`[scheduling][[dependencies]][[[18]]]`	`[scheduling][[dependencies]][[[T18]]]`

The graph text in hour-based cycling is also easy to convert, as seen in Table 5.

Table 5: Hourly Cycling Offsets

Pre-Cylc-6	Cylc-6+
`my_task[T-6]`	`my_task[-PT6H]`
`my_task[T-12]`	`my_task[-PT12H]`
`my_task[T-24]`	`my_task[-PT24H]` or even `my_task[-P1D]`

K.5   No Implicit Creation of Tasks by Offset Triggers

Prior to cylc-6 inter-cycle triggers implicitly created task instances at the offset cycle points. For example, this pre cylc-6 suite automatically creates instances of task `foo` at the offset hours `3,9,15,21` each day, for task `bar` to trigger off at `0,6,12,18`:

```
# Pre cylc-6 implicit cycling.
[scheduling]
    initial cycle time = 2014080800
    [[dependencies]]
        [[[00,06,12,18]]]
            # This creates foo instances at 03,09,15,21:
            graph = foo[T-3] => bar
```

237

You can run this suite to see how it works (cylc-6+ is backward-compatible: it generates the old-style behaviour when it detects an old-style suite definition). Here's the direct translation of this suite to cylc-6+ format:

```
# In cylc -6+ this suite will stall.
[scheduling]
   initial cycle point = 20140808T00
   [[dependencies]]
      [[[T00,T06,T12,T18]]]
         # This does NOT create foo instances at 03,09,15,21:
         graph = foo[-PT3H] => bar
```

If you run this suite, `bar.20140808T00` will execute at start-up because dependence on tasks prior to the initial cycle point is ignored, but the suite will then stall with `bar.20140808T06` waiting on `foo.20140808T03` (right-click on the task in the GUI to see its prerequisites), which does not exist because the offset `foo` instances have not been created. Note that if you graph the suite the offset `foo` instances do appear. That's because the graph expresses dependence, and `bar` really does depend on these instances of `foo`. The problem is just that the tasks don't get created at run time, which is why the suite stalls. Here's the correct way to get the desired behaviour in cylc-6+:

```
# Cylc -6+ requires explicit task instance creation.
[scheduling]
   initial cycle point = 20140808T00
   [[dependencies]]
      [[[T03,T09,T15,T21]]]
         graph = foo
      [[[T00,T06,T12,T18]]]
         graph = foo[-PT3H] => bar
```

Implicit creation of task instances by offset triggers has been disabled because it is error prone: if a trigger offset is wrong it should cause a triggering failure rather than create task instances at incorrect cycle points.

L   Known Issues

L.1   Current Known Issues

The best place to find current known issues is on Github: https://github.com/cylc/cylc/issues.

L.2   Notable Known Issues

L.2.1   Use of pipes in job scripts

In bash, the return status of a pipeline is normally the exit status of the last command. This is unsafe, because if any command in the pipeline fails, the script will continue nevertheless.

For safety, a cylc task job script running in bash will have the `set -o pipefail` option turned on automatically. If a pipeline exists in a task's `script`, etc section, the failure of any part of a pipeline will cause the command to return a non-zero code at the end, which will be reported as a task job failure. Due to the unique nature of a pipeline, the job file will trap the failure of the individual commands, as well as the whole pipeline, and will attempt to report a failure

back to the suite twice. The second message is ignored by the suite, and so the behaviour can be safely ignored. (You should probably still investigate the failure, however!)

M   GNU GENERAL PUBLIC LICENSE v3.0

Copyright © 2007 Free Software Foundation, Inc. `http://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies of this

license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend

this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of

technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

(a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

(b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

(c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

(d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified

object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or

(2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to

the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IM-PLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE EN-TIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, IN-CIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.