Upstart Intro, Cookbook and Best Practises

Document generated from reStructuredText plaintext markup source on 2013-01-25 at 08:12:20 from Bazaar branch branch. See Upstart Documenters and Upstart Cookbook. (revision $Revision-Id).

Contents

1 Meta

1.1 Document Version

This is document edit 175.

See footer for further details.

1.2 Authors

Authors:

- James Hunt <james.hunt@canonical.com>
- Clint Byrum <clint.byrum@canonical.com>

1.3 Acknowledgements

The Authors are grateful to the following individuals who have provided valuable input to this document:

- Colin Watson (Canonical)
- Scott James Remnant (Canonical, Google), author of Upstart.
- James Page (Canonical)
- Joel Ebel (Google)
- Mark Russell (Canonical)
- Bradley Ayers
- Kenneth Porter
- Roberto Alsina (Canonical), reStructuredText Guru.

1.4 Purpose

The purpose of this document is multi-faceted. It is intended as:

- A gentle introduction to Upstart.
- A Cookbook of recipes and best-practises for solving common and not so common problems.
- An extended guide to the configuration syntax of Upstart.

It attempts to explain the intricacies of Upstart with worked examples and lots of details.

Note that the reference documentation for Upstart will *always* be the manual pages: this is merely a supplement to them.

1.5 Suggestions and Errata

Bad documentation is often worse than no documentation. If you find a problem with this document, however small...

- spelling error
- grammatical error
- factual error
- inconsistency
- lack of clarity
- ambiguous or misleading content
- missing information
- *et cetera*

... or if you'd like to see some particular feature covered *please* raise a bug report on the Upstart Cookbook project website so that we can improve this work:

- https://bugs.launchpad.net/upstart-cookbook/+filebug

As an incentive you will be credited in the Acknowledgements section.

1.6 Coverage

There are essentially two major versions of Upstart covered by this document:

1.6.1 Upstream Upstart

This is the pure, or "vanilla" version which is designed to work on any Linux system:

- Homepage

  http://launchpad.net/upstart
- Bug Reports

  http://bugs.launchpad.net/upstart
- Questions

  https://answers.launchpad.net/upstart/+addquestion

1.6.2 Ubuntu Version of Upstart

The Ubuntu-packaged version [13].

This is a "debianised" version of Upstart (in other words, a version packaged for Debian and derivatives). It includes a few minor changes specifically for running Upstart on an Ubuntu system, namely:

- Change to the way the console is initialised, to work with Plymouth.
- Initramfs to root filesystem context hand-off changes.

Links:

- Homepage

  http://launchpad.net/ubuntu/+source/upstart
- Bug Reports

  http://bugs.launchpad.net/ubuntu/+source/upstart
- Questions

  https://answers.launchpad.net/ubuntu/+source/upstart/+addquestion

1.6.3 Availability

Upstart is relied upon by millions of systems across a number of different Operating Systems including:

- Google's Chrome OS
- Google's Chromium OS
- Red Hat's RHEL 6 [34]
- Ubuntu

It is also available as an option for other systems such as:

- Debian
- Fedora

1.6.4 Ubuntu-Specific

This document is written with Ubuntu in mind, but will attempt to identify Ubuntu-specific behaviour where appropriate by showing this icon: (displays as "U" on section headings).

1.7 Audience

This document is targeted at:

- Users interested in learning about Upstart.
- System Administrators looking to make the most of the capabilities of Upstart.
- Developers and Packagers who wish to package their application to work with Upstart.

1.8 Document Preparation

This document is written in reStructuredText, a textual markup language. The document was prepared using the following tools:

- Vim editor.
- Emacs editor with Org-Mode for tables.
- Jave for ASCII graphics.

1.9 Document Availability

The source for this document is available here:

- https://code.launchpad.net/~upstart-documenters/upstart-cookbook/trunk

The latest version of this document should always be available from:

- http://upstart.ubuntu.com/cookbook/
- http://upstart.ubuntu.com/cookbook/upstart_cookbook.pdf

1.10 Warning

This document aims to aid understanding of Upstart and identify some hopefully useful "canned" solutions and advice to common problems and questions.

The authors have taken as much care as possible in the preparation of this document. However, you are advised strongly to exercise extreme caution when changing critical system facilities such as the `init` daemon. Most situations are recoverable and advice is provided in this document, but if your system explodes in a ball of fire or becomes unusable as a result of a suggestion from this document, you alone have the intellectual pleasure of fixing your systems.

2 Typographical Conventions

2.1 Commands and configuration stanzas

Throughout this document a fixed-width font such as `this` will be used to denote commands, brief command output and configuration stanzas.

2.2 User Input and Command Output

An indented block will be used to denote user input and command output.

2.2.1 Non-Privileged User

Indented lines starting with a dollar character ('$') are used to denote the shell prompt (followed by optional commands) for a non-privileged user. Command output is shown by indented lines not preceded by the dollar character:

```
$ echo hello
hello
```

2.2.2 Super-User

Indented lines starting with a hash (or "pound") character ('#') are used to denote the shell prompt (followed by optional commands) for the root user. Command output is shown by indented lines not preceded by the hash character [10]:

```
# whoami
root
```

Note that some examples make use of sudo(8) to show the command should be run as root: the example above could thus be written:

```
$ sudo whoami
root
```

This latter approach is clearer in the context where a comment is also specified using the hash character.

2.3 Configuration Examples

An indented block is also used to show examples of job configuration:

```
script
  # a config file
end script
```

3 Introduction

3.1 What is Upstart?

Quoting from http://upstart.ubuntu.com/,

> Upstart is an event-based replacement for the `/sbin/init` daemon which handles starting of tasks and services during boot, stopping them during shutdown and supervising them while the system is running.

The "`init`" or "system initialisation" process on Unix and Linux systems has process ID (PID) "`1`". That is to say, it is the first process to start when the system boots (ignoring the initrd/initramfs). As the quote shows, Upstart is an "`init`" replacement for the traditional Unix "System V" "`init`" system. Upstart provides the same facilities as the traditional "`init`" system, but surpasses it in many ways.

3.1.1 Reliability

Upstart is written using the NIH Utility Library ("`libnih`"). This is a very small, efficient and safe library of generic routines. It is designed for applications that run early in the boot sequence ("plumbing"). Reliability and safety is critically important for an `init` daemon since:

- it runs as the super-user.
- it is responsible for managing critical system services.
- if init exits for any reason, the kernel panics.

To help ensure reliability and avoid regressions, Upstart and the NIH Utility Library both come with comprehensive test suites. See Unit Tests for further information.

3.1.2 Design History

Upstart was created due to fundamental limitations in existing systems. Those systems can be categorized into two types:

- System V init system
- Dependency-based init systems

To understand why Upstart was written and why its revolutionary design was chosen, it is necessary to consider these two classes of init system.

3.1.2.1 Critique of the System V init System

3.1.2.1.1 SysV Benefits

3.1.2.1.1.1 Simplicity

Creating service files is easy with SystemV init since they are simply shell scripts. To enable/disable a service in a particular runlevel, you only need to create/remove a symbolic link in a particular directory or set of directories.

3.1.2.1.1.2 Guaranteed Ordering of Services

This is achieved by init running the scripts pointed to by the symbolic links in sequence. The relative order in which init invokes these scripts is determined by a numeric element in the name: lower numbered services run before higher numbered services.

3.1.2.1.2 SysV Limitations

3.1.2.1.2.1 Non-Optimal Performance

The traditional sequential boot system was appropriate for the time it was invented, but by modern standards it is "slow" in the sense that it makes no use of parallelism.

It was designed to be simple and efficient for Administrators to manage. However, this model does not make full use of modern system resources, particularly once it is recognised that multiple services can often be run simultaneously.

A common "hack" used by Administrators is to circumvent the serialisation by running their service in the background, such that some degree of parallelism is possible. The fact that this hack is required and is common on such systems demonstrates clearly the flaw in that system.

3.1.2.1.2.2 Server-Centric

In the days of colossal Unix systems with hundreds of concurrent users, where reboots were rare, the traditional SysV approach was perfect. If hardware needed replacing, a system shutdown was scheduled, the shutdown performed, the new hardware was installed and the system was brought back on-line.

However, the world has now moved on. From an Ubuntu perspective, a significant proportion of users run the desktop edition on portable devices where they may reboot multiple times a day.

3.1.2.1.2.3 Assumes Static Hardware at all Times

Modern Linux systems can deal with new hardware devices being added and removed dynamically ("hot-plug"). The traditional SysV init system itself is incapable of handling such a dynamically changing system.

3.1.2.1.2.4 Every Service Does Heavy Lifting

Most service files are fairly formulaic. For example, they might:

- perform initial checks, such as:
  - ensuring no other instance of a daemon is running.
  - checking the existence of a directory or file.
  - removing old cache files.
- ensure dependent daemons are running.
- spawn the main service.

The most difficult and time costly operation these services perform is that of handling dependent daemons. The LSB specifies helper utilities that these services can make use of, but arguably each service shouldn't need to be handling this activity *themselves*: the init system itself should do it on behalf of the services it manages.

3.1.2.2 Critique of Dependency-Based init Systems

3.1.2.2.1 Benefits of Dependency-based init

3.1.2.2.1.1 Recognises Services Require Other Services

The recognition that services often need to make use of other services is an important improvement over SystemV init systems. It places a bigger responsibility on the init system itself and reduces the complexity and work that needs to be performed by individual service files.

3.1.2.2.2 Limitations of Dependency-based init

3.1.2.2.2.1 Does Not Recognise Dynamic Nature of Linux

The main problem with dependency-based init systems is that they approach the problem from the "wrong direction". Again, this is due to their not recognising the dynamic nature of modern Linux systems.

For example, if a dependency-based init system wished to start say MySQL, it would *first* start all the dependent services that MySQL needed. This sounds perfectly reasonable.

However, consider how such a system would approach the problem of dealing with a user who plugs in an external monitor. Maybe we'd like our system to display some sort of configuration dialogue so the user can choose how they want to use their new monitor in combination with their existing laptop display. This can only be "hacked" with a dependency-based init system since you do not know when the new screen will be plugged. So, your choices are either:

- Do nothing.

  Corresponds to an inability to handle this scenario.
- Have a daemon that hangs around polling for new hardware being plugged.

  Wasteful and inefficient.

What you really want is a system that detects such asynchronous events and when the conditions are right for a service to run, the service is started.

This can be summarised as:

- Upstart starts a service when its required conditions are met.

  The service (job configuration file) only needs to specify the conditions that allow the service to run, and the executable to run the service itself.

- Dependency-based init systems meet a service's dependencies before starting them.

  Each service generally does this using a brute-force approach of forcing all the dependencies to start.

  Note that the init system itself is not doing the heavy-lifting: that is left up to each service itself (!)

This summary is worth considering carefully as the distinction between the two types of system is subtle but important.

The other problem with dependency-based init systems is that they require a dependency-solver which is often complex and not always optimal.

3.1.2.3 Upstart's Design: Why It Is Revolutionary

It was necessary to outline the limitations of the SysV and dependency-based init systems to appreciate why Upstart is special...

Upstart is revolutionary as it recognises *and was designed specifically for* a dynamic system. It handles asynchronicity by emitting events. This too is revolutionary.

Upstart emits "events" which services can register an interest in. When an event -- or combination of events -- is emitted that satisfies some service's requirements, Upstart will automatically start or stop that service. If multiple jobs have the same "start on" condition, Upstart will start those jobs "in parallel". To be manifest: Upstart handles starting the "dependent" services itself - this is not handled by the service file itself as it is with dependency-based systems.

Further, Upstart is being guided by the ultimate arbiter of hardware devices: the kernel.

In essence, Upstart is an event engine: it creates events, handles the consequences of those events being emitted and starts and stops processes as required. Like the best Unix software, it does this job very well. It is efficient, fast, flexible *and reliable*. It makes use of "helper" daemons (such as the upstart-udev-bridge and the upstart-socket-bridge) to inject new types of events into the system and react to these events. This design is sensible and clean: the init system itself must not be compromised since if it fails, the kernel panics. Therefore, any functionality which is not considered "core" functionality is farmed out to other daemons.

See [31] for further details.

3.1.3 Performance

Upstart was designed with performance in mind. It makes heavy use of the NIH Utility Library which is optimised for efficient early boot environments. Additionally, Upstart's design is lightweight, efficient and elegant. At its heart it is a event-based messaging system that has the ability to control and monitor processes. Upstart is designed to manage services running in parallel. It will only start services when the conditions they have specified are met.

3.1.4 Server

Upstart is used by Ubuntu for the Ubuntu Desktop and for Ubuntu Server (and as a result of this, it is also used in the Ubuntu Cloud). Why is Upstart also compelling in a server environment?

3.1.4.1 Boot Performance

Some say that boot performance is not important on servers, possibly since the time taken to bring RAID arrays on-line is significantly longer than the time it takes to boot the operating system. However, nobody seriously wants their system to take longer than necessary to boot.

Consider also the case for Cloud deployments, which of course run on servers. Here, boot speed is very important as it affects the time taken to deploy a new server instance. The faster you can deploy new services to handle an increasing workload the better the experience for your customers.

3.1.4.2 Failure Modes

It's a fact that systems and software are getting more complex. In the old days of Unix, runlevels encompassed every major mode of operation you might want your system to handle. However, expectations have changed. Nowadays, we expect systems to react to problems (and maybe even "self-heal" the simple ones).

The landscape has changed and Upstart is fully able to accommodate such changes since its design is clean, elegant and abstract. Crucially, Upstart is not tied to the rigid runlevel system. Indeed, Upstart has no knowledge of runlevels internally, but it supports them trivially with events. And since events are so abstract, they are highly flexible building blocks for higher-level constructs. Added to which, since Upstart's events are dynamic, the system can be configured for a myriad of possible system behaviours and failure modes and have it react accordingly.

4 Concepts and Terminology

The main concepts in Upstart are "events" and "jobs". Understanding the difference between the two is crucial.

4.1 Job

A "*unit of work*" - generally either a "*Task*" or a "*Service*". Each Job is defined in a Job configuration file.

4.1.1 Job Types

4.1.1.1 Task Job

A *Task Job* is one which runs a short-running process, that is, a program which might still take a long time to run, but which has a definite lifetime and end state.

For example, deleting a file could be a *Task Job* since the command starts, deletes the file in question (which might take some time if the file is huge) and then the delete command ends.

In this book *Task Jobs* are often referred to as *tasks*.

4.1.1.2 Service Job

A *Service Job* is a long-running (or daemon(3) process). It is the opposite of a *Task Job* since a *Service Job* might never end of its own accord.

Examples of *Service Jobs* are entities such as databases, webservers or ftp servers.

4.1.1.3 Abstract Job

There is one other type of job which has *no* script sections or `exec` stanzas. Such abstract jobs *can* still be started and stopped, but will have no corresponding child process (PID). In fact, starting such a job will result in it "running" perpetually if not stopped by an Administrator. Abstract jobs exist only within Upstart itself but can be very useful. See for example:

- Jobs that "Run Forever"
- Synchronisation

4.1.2 Job States

The table below shows all possible Job States and the legal transitions between them. States are exposed to users via the `status` field in the output of the initctl status command.

Job State Transitions.

Current State	Goal	
	start	stop
waiting	starting	n/a
starting	pre-start	stopping
pre-start	spawned	stopping
spawned	post-start	stopping
post-start	running	stopping
running	stopping	pre-stop or stopping [11]
pre-stop	running	stopping
stopping	killed	killed
killed	post-stop	post-stop
post-stop	starting	waiting

For example, if the job is currently in state `starting`, and its goal is `start`, it will then move to the `pre-start` state.

Note that jobs may change state so quickly that you may not be able to observe all the values above in the `initctl` output. However, you will see the transitions if you raise the log-priority to `debug` or `info`. See initctl log-priority for details.

Details of states:

- `waiting` : initial state.
- `starting` : job is about to start.
- `pre-start` : running pre-start section.
- `spawned` : about to run `script` or `exec` section.
- `post-start` : running post-start section.
- `running` : interim state set after post-start section processed denoting job is running (But it may have no associated PID!)
- `pre-stop` : running pre-stop section.
- `stopping` : interim state set after pre-stop section processed.
- `killed` : job is about to be stopped.
- `post-stop` : running post-stop section.

4.1.2.1 Viewing State Transitions

To view state transitions:

1. Change the log-priority to `debug`
2. "`tail -f`" your system log file
3. start/stop/restart a job or emit an event.

4.2 Job Configuration File

A Job is defined in a *Job Configuration File* (or more simply a *conf file*) which is a plain text file containing one or more *stanzas*. Job configuration files are named:

```
<name>.conf
```

Where "<name>" should reflect the application being run or the service being provided.

Job configuration files can exist in two types of location, depending on whether they are a System Job or a User Job.

Note that it is common to refer to a Job configuration file as a "job", although technically a job is a running instance of a Job configuration file.

4.2.1 System Job

All system jobs by default live in the following directory:

```
/etc/init/
```

This directory *can* be overriden by specifing the `--confdir=<directory>` option to the init daemon, however this is a specialist option which users should not need to use.

4.2.2 User Job

With the advent of Upstart 1.3, non-privileged users are able to create jobs by creating job configuration files in the following directory:

```
$HOME/.init/
```

This feature is not currently enabled in Ubuntu (up to and including 11.10 ("Oneiric Ocelot")).

The syntax for such jobs is identical for "system jobs".

> **Note**
>
> Currently, a user job cannot be created with the same name as a system job: the system job will take precedence.

Controlling user jobs is the same as for system jobs: use initctl, start, stop, *et cetera*.

> **Note**
>
> Stanzas which manipulate resources limits (such as limit, `nice`, and `oom`) may cause a job to fail to start should the value provided to such a stanza attempt to exceed the maximum value the users privilege level allows.

> **Note**
>
> User jobs cannot currently take advantage of job logging. If a user job does specify console log, it is considered to have specified console none. Logging of user jobs is planned for the next release

of Upstart.

4.2.2.1 Enabling

To enable user jobs, the administrator must modify the *D-Bus* configuration file "*Upstart.conf*" to allow non-root users access to all the Upstart D-Bus methods and properties. On an Ubuntu system the file to modify is:

```
/etc/dbus-1/system.d/Upstart.conf
```

The Upstream Upstart 1.3 distribution already includes a "*Upstart.conf*" file containing the required changes.

4.2.3 Odd Jobs

4.2.3.1 Job with `start on`, but no `stop on`

A job does not necessarily need a stop on stanza. If it lacks one, any running instances can still be stopped by an Administrator running either of:

- `initctl stop <job>`
- `stop <job>`

However, if such a job is not stopped, it may be stopped either by another job, or some other facility [28]. Worst case, if nothing else stops it, all processes will obviously be killed when the system is powered off.

4.2.3.2 Job with `stop on`, but no `start on`

If a job has no start on stanza, it can *only* be started manually by an Administrator running either of:

- `initctl start <job>`
- `start <job>`

If any job instances are running at system shutdown time, Upstart will stop them.

4.2.3.3 Job with no `stop on` or `start on`

Such a job can only be controlled by an Administrator. See Job with start on, but no stop on and Job with stop on, but no start on.

4.2.3.4 Minimal Job Configuration

What is the minimum content of a job configuration file? Interestingly enough, to be valid a job configuration file:

- must not be empty
- must be syntactically correct
- must contain at least one legal stanza

Therefore, some examples of minimal job configuration files are:

- Comments only:

  ```
  # this is an abstract job containing only a comment
  ```

- `author` stanza only:

```
author "foo"
```

- `description` stanza only:

```
description "this is an abstract job"
```

As shown, these are all example of Abstract Job configuration files.

4.3 Event

A notification is sent by Upstart to all interested parties (either jobs or other events). They can generally be thought of as "signals", "methods", or "hooks" [21], depending on how they are emitted and/or consumed.

Events are *emitted* (created and then broadcast) to the entire Upstart system. Note that it is not possible to stop any other job or event from seeing an event when it is emitted.

If there are no jobs which have registered an interest in an event in either their start on or stop on conditions, the event has no effect on the system.

Events can be created by an administrator at any time using:

```
# initctl emit <event>
```

Note that some events are "special". See the upstart-events(7) manual page for a list.

Note also that an event name with the same name as a job is allowed.

Jobs are often started or stopped as a result of *other* jobs starting or stopping. Upstart has a special set of events that it emits to announce these job state transitions. You'll probably notice that these events have the same names as some of the job states described in Job States, however it's important to appreciate that these are *not* describing the same thing. Task states are not events, and events are not task states. See Events, not States for details.

These events are as follows:

`starting`
: This event is emitted by Upstart when a job has been scheduled to run and is *about to start* executing.

`started`
: This event is emitted by Upstart when a job is now running. Note that a job does not *have* to have an associated program or script so "running" does not necessarily imply that any additional process is executing.

`stopping`
: This event is emitted by Upstart when a job is *about to be stopped*.

`stopped`
: This event is emitted by Upstart when a job has completed (successfully or otherwise).

See Job Lifecycle for further details.

To help reinforce the difference, consider how Upstart itself starts: See the Startup Process.

1. It performs its internal initialization.
2. Upstart itself emits a single event called startup(7). This event triggers the rest of the system to initialize. Note that there is no "startup" job (and hence no `/etc/init/startup.conf` file).

3. init(8) runs the mountall job (as defined in `/etc/init/mountall.conf`) since the startup(7) event satisfies mountall(8)'s requirement: "`start on startup`".
4. The mountall(8) job in turn emits a number of events (including local-filesystems(7) and all-swaps(7)). See upstart-events(7) for further details.

Upstart provides three different **types** of Events.

4.3.1 Event Types

4.3.1.1 Signals

A *Signal Event* is a *non-blocking (or asynchronous) event*. Emitting an event of this type returns immediately, allowing the caller to continue. Quoting from [22]:

> The announcer of a signal cares not whether anybody cared about it, and doesn't wait around to see whether anything happened. As far as the announcer cares, it's informational only.

Signal Events are created using the `--no-wait` option to the `initctl emit` command like this:

```
# initctl emit --no-wait mysignal
```

The non-blocking behaviour directly affects the emitter by allowing it to continue processing without having to wait for any jobs which make use of the event. Jobs which make *use* of the event (via start on or stop on) are also affected, as they're unable to stop, delay, or in any other way "hold up" the operation of the emitter.

4.3.1.2 Methods

A *Method Event* is a *blocking (or synchronous) event* which is usually coupled with a *task*. It acts like a method or function call in programming languages in that the caller is requesting that some work be done. The caller waits for the work to be done, and if problems were encountered, it expects to be informed of this fact.

Emitting a Method Event is simple:

```
# initctl emit mymethod
```

This is exactly like a *Signal Event*, except the event is being emitted synchronously such that the emitter has to wait until the `initctl` command completes. Once the `initctl` command has completed, there are two possible outcomes for the task that starts on Event `mymethod`:

- The task runs successfully.
- The task failed for some reason.

Assuming we have a job configuration file `/etc/init/myapp.conf` like this:

```
start on mymethod
task
exec /usr/bin/myapp $ACTION
```

You could start the `myapp` job and check if the "method" worked as follows:

```
# initctl emit mymethod ACTION=do_something
[ $? -ne 0 ] && { echo "ERROR: myapp failed"; exit 1; }
```

4.3.1.3 Hooks

A *Hook Event* is a *blocking (or synchronous) event*. Quoting from [23]:

> "A hook is somewhere between a signal and a method. It's a notification that something changed on the system, but unlike a signal, the emitter waits for it to complete before carrying on."

Hooks are therefore used to flag to all interested parties that something is about to happen.

The canonical examples of Hooks are the two job events starting(7) and stopping(7), emitted by Upstart to indicate that a job is *about to start* and *about to stop* respectively.

4.3.2 Events, not States

Although Upstart does use states internally (and these are exposed via the list and status commands in initctl(8)), events are the way that job configuration files specify the desired behaviour of jobs: starting(7), started(7), stopping(7), stopped(7) are events, not states. These events are emitted "just prior" to the particular transition occurring. For example, the starting(7) event is emitted just before the job associated with this event is actually queued for start by Upstart.

4.4 Job Lifecycle

4.4.1 Starting a Job

1. Initially the job is "at rest" with a goal of `stop` and a state of `waiting` (shown as `stop/waiting` by the initctl list and initctl status commands).
2. The goal is changed from `stop` to `start` indicating the job is attempting to start.
3. The state is changed from `waiting` to `starting`.
4. The starting(7) event is emitted denoting the job is "about to start".
5. Any jobs whose start on (or stop on) condition would be satisfied by this job starting are started (or stopped respectively).
6. The starting(7) event completes.
7. The state is changed from `starting` to `pre-start`.
8. If the pre-start stanza exists, the pre-start process is spawned.
9. If the pre-start process fails, the goal is changed from `start` to `stop`, and the stopping(7) and stopped(7) events are emitted with appropriate variables set denoting the error.
10. Assuming the pre-start did not fail or did not call "`stop`", the main process is spawned.
11. The state is changed from `pre-start` to `spawned`.
12. Upstart then ascertains the final PID for the job which may be a descendent of the immediate child process if expect fork or expect daemon has been specified.
13. The state is changed from `spawned` to `post-start`.
14. If the post-start stanza exists, the post-start process is spawned.
15. The state is changed from `post-start` to `running`.
16. The started(7) event is emitted.

    For services, when this event completes the main process will now be fully running. If the job refers to a task, it will now have completed (successfully or otherwise).
17. Any jobs whose start on (or stop on) condition would be satisfied by this job being started are started (or stopped respectively).

4.4.2 Stopping a Job

1. Assuming the job is fully running, it will have a goal of `start` and a state of `running` (shown as `start/running` by the initctl list and initctl status commands).
2. The goal is changed from `start` to `stop` indicating the job is attempting to stop.
3. The state is changed from `running` to `pre-stop`.
4. If the pre-stop stanza exists, the pre-stop process is spawned.
5. The state is changed from `pre-stop` to `stopping`.
6. The stopping(7) event is emitted.

   The `stopping` event has a number of associated environment variables:

   - `JOB`

     The name of the job this event refers to.
   - `INSTANCE`

     The name of the instance of the job this event refers to. This will be empty for single-instance jobs (those jobs that have not specified the instance stanza).
   - `RESULT`

     This variable will have the value "`ok`" if the job exited normally or "`failed`" if the job exited due to failure. Note that Upstart's view of success and failure can be modified using the normal exit stanza.
   - `PROCESS`

     The name of the script section that resulted in the failure. This variable is not set if `RESULT=ok`. If set, the variable will have one of the following values:

     - `pre-start`
     - `post-start`
     - `main` (denoting the `script` or `exec` stanza)
     - `pre-stop`
     - `post-stop`
     - `respawn` (denoting the job attempted to exceed its respawn limit)
   - `EXIT_STATUS` *or* `EXIT_SIGNAL`

     Either `EXIT_STATUS` or `EXIT_SIGNAL` will be set, depending on whether the job exited itself (`EXIT_STATUS`) or was stopped as a result of a signal (`EXIT_SIGNAL`).

     If neither variable is set, the process in question failed to spawn (for example, because the specified command to run was not found).
7. Any jobs whose start on (or stop on) condition would be satisfied by this job stopping are started (or stopped respectively).
8. The main process is stopped:

   - The signal specified by the kill signal stanza is sent to the process group of the main process. (such that all processes belonging to the jobs main process are killed). By default this signal is `SIGTERM`.

     See signal(7) and init(5).
   - Upstart waits for up to kill timeout seconds (default 5 seconds) for the process to end.

   - If the process is still running after the timeout, a `SIGKILL` signal is sent to the process which cannot be ignored and will forcibly stop the processes in the process group.
9. The state is changed from `killed` to `post-stop`.
10. If the post-stop stanza exists, the post-stop process is spawned.
11. The state is changed from `post-stop` to `waiting`.
12. The stopped(7) event is emitted.

    When this event completes, the job is fully stopped.
13. Any jobs whose start on (or stop on) condition would be satisfied by this job being stopped are started (or stopped respectively).

Note: this information is also available in upstart-events(7).

4.5 Ordering

4.5.1 Order in which Events are Emitted

As a general rule, *you cannot rely upon the the order in which events will be emitted*. Your system is dynamic and Upstart responds to changes *as-and-when* they occur (for example hot-plug events).

That said, most systems which use Upstart provide a number of "well-known" events which you *can* rely upon.

For example on Ubuntu, these are documented in the upstart-events(7) man page, which is included within this document for convenience in appendix Ubuntu Well-Known Events (ubuntu-specific).

4.5.2 Order in Which Jobs Which start on the Same Event are Run

Assume you have three jobs like this:

- `/etc/init/X.conf`

  ```
  start on event-A
  ```
- `/etc/init/Y.conf`

  ```
  start on event-A
  ```
- `/etc/init/Z.conf`

  ```
  start on event-A
  ```

Question: If event `event-A` is emitted, which job will run first?

Answer: It is not possible to say, and indeed *you should not make any assumptions about the order in which jobs with the same conditions run in*.

4.5.3 Ordering of Stop/Start Operations

4.5.3.1 Single Job

Imagine a job configuration file `/etc/init/odd.conf` like this:

```
start on event-A
stop  on event-A

script
  sleep 999
end script
```

Would Upstart be happy with this? Actually, yes it would! Upstart *always* handles *stop on* stanzas before handling *start on* stanzas. This means that this strange job would first be stopped (if it's currently running), then it would be started.

We can see what happens when we run this job more clearly when we increase the log priority to debug (see Change the log-priority):

```
# initctl log-priority debug
```

Now, we can watch the state transitions by viewing the system log.

4.5.3.1.1 If Job is Not Currently Running

```
# status odd
odd stop/waiting
# initctl emit event-A
# status odd
odd start/running, process 9474
```

And here is an example from the system log (with annotations) showing what happened:

```
event_new: Pending event-A event                # Upstart emitted the event.
Handling event-A event
event_pending_handle_jobs: New instance odd     # Job instance created.
odd goal changed from stop to start             # Since job not running,
odd state changed from waiting to starting      # change goal to "start".
event_new: Pending starting event
Handling starting event
event_finished: Finished starting event
odd state changed from starting to pre-start
odd state changed from pre-start to spawned
odd main process (9474)                         # Start script section.
odd state changed from spawned to post-start
odd state changed from post-start to running    # Job now fully started.
event_new: Pending started event
Handling started event
event_finished: Finished started event
event_finished: Finished event-A event
```

4.5.3.1.2 If Job is Currently Running

```
# status odd
odd stop/waiting
# start odd
odd start/running, process 11416    # Note this PID!
# status odd
odd start/running, process 11416
# initctl emit event-A
# status odd
odd start/running, process 11428    # Look! It changed!
```

Here is an example from the system log showing what happened in more detail. First the entries relating to starting the job:

```
odd goal changed from stop to start
odd state changed from waiting to starting
event_new: Pending starting event
Handling starting event
event_finished: Finished starting event
odd state changed from starting to pre-start
odd state changed from pre-start to spawned
odd main process (11416)
odd state changed from spawned to post-start
odd state changed from post-start to running
event_new: Pending started event
Handling started event
event_finished: Finished started event
```

Now, the event is emitted:

```
event_new: Pending event-A event
Handling event-A event
odd goal changed from start to stop             # Job already running, so stop it.
odd state changed from running to pre-stop
odd state changed from pre-stop to stopping
event_new: Pending stopping event
event_pending_handle_jobs: New instance odd
odd goal changed from stop to start
Handling stopping event
event_finished: Finished stopping event
odd state changed from stopping to killed
Sending TERM signal to odd main process (11416) # Forcibly stop existing job process.
odd main process (11416) killed by TERM signal  # Successfully stopped it.
odd state changed from killed to post-stop
odd state changed from post-stop to starting
event_new: Pending starting event
Handling starting event
event_finished: Finished starting event
odd state changed from starting to pre-start
odd state changed from pre-start to spawned
odd main process (11428)                        # New instance of job started with new PID.
odd state changed from spawned to post-start
odd state changed from post-start to running
event_new: Pending started event
Handling started event
event_finished: Finished started event
event_finished: Finished event-A event
```

4.5.3.2 Multiple Jobs

Upstart guarantees that jobs which stop on a particular event are processed before jobs that start on *the same event*.

Consider two jobs like this:

- `A.conf`:

```
start on startup
stop on foo
```

- `B.conf`:

```
start on foo
```

Assuming that job "A" is already running, if the "`foo`" event is emitted, Upstart will always stop job "A" before starting job "B".

4.6 Runlevels

A runlevel is a single-byte name for a particular system configuration. Runlevels for Debian and Ubuntu systems are generally as follows [30]:

- `0` : System halt.
- `1` : Single-User mode.
- `2` : Graphical multi-user plus networking (**DEFAULT**)
- `3` : Same as "2", but not used.
- `4` : Same as "2", but not used.
- `5` : Same as "2", but not used.
- `6` : System reboot.

There are also a few pseudo-runlevels:

- `N` : The previous runlevel cannot be determined.
- `S` : Alias for Single-User mode.

4.6.1 Display Runlevel

To display your current and previous runlevels separated by a space character, run the `/sbin/runlevel` command. Note that if this command is unable to determine the system runlevel, it may display simply "`unknown`":

```
$ runlevel
N 2
```

The output above shows that:

- there was no *previous* runlevel (the system was booted and went straight to the *current* runlevel).
- the current runlevel is "2".

4.6.2 Change Runlevel Immediately

To change runlevel immediately, use one of the commands below:

- reboot(8)
- shutdown(8)
- telinit(8)

4.6.3 Changing the Default Runlevel

4.6.3.1 Permanently

To change the default runlevel the system will boot into, modify the variable `DEFAULT_RUNLEVEL` in file `/etc/init/rc-sysinit.conf`. For example, to make the system boot by default to single user mode, set:

```
env DEFAULT_RUNLEVEL=1
```

4.6.3.2 Single Boot

If you want to change the default runlevel for a single boot, rather than making the change permanent by modify the `rc-sysinit.conf` file, simply append the variable to the kernel command line:

```
DEFAULT_RUNLEVEL=1
```

Traditionally, the default runlevel was encoded in file `/etc/inittab`. However, with Upstart, this file is no longer used (it is supported by Upstart, but its use is deprecated).

5 System Phases

The information in this section relates to an Ubuntu system.

To obtain a better understanding of how jobs and events relate at startup and shutdown time, see Visualising Jobs and Events.

5.1 Startup

At boot, after the initramfs system has been run (for setting up RAID, unlocking encrypted file system volumes, *et cetera*), Upstart will be given control. The initramfs environment will exec(3) `/sbin/init` (this is the main Upstart binary) and cause it to run as PID 1.

5.1.1 Startup Process

Note that in this section we assume the default runlevel is "2". See Changing the Default Runlevel for further details.

1. Upstart performs its internal initialization.
2. Upstart itself emits a single *event* called startup(7).

   This event triggers the rest of the system to initialize [29].
3. init(8) runs a small number of *jobs* which specify the startup(7) event in their start on condition.

   The most notable of these is the `mountall` job which mounts your disks and filesystems.
4. The mountall(8) job in turn emits a number of events.

   These include local-filesystems(7), virtual-filesystems(7) and all-swaps(7). See upstart-events(7) for further details.
5. The virtual-filesystems(7) event causes the `udev` job to start.

6. The `udev` job causes the upstart-udev-bridge job to start.
7. The upstart-udev-bridge job will at some point emit the "`net-device-up IFACE=lo`" event signifying the local network (for example, `127.0.0.0` for IPv4) is available.
8. After the last filesystem is mounted, mountall(8) will emit the `filesytem` event.
9. Since the start on condition for the `rc-sysinit` job is:

   ```
   start on filesystem and net-device-up IFACE=lo
   ```

   Upstart will then start the `rc-sysinit` job.
10. The `rc-sysinit` job calls the `telinit` command, passing it the runlevel to move to:

    ```
    telinit 2
    ```

11. The `telinit` command emits the runlevel(7) event as:

    ```
    runlevel RUNLEVEL=2 PREVLEVEL=N
    ```

    Note that this is *all* the `telinit` command does – it runs no commands itself to change runlevel!

    See Runlevels for further information on runlevels.
12. The runlevel(7) event causes many other Upstart jobs to start, including `/etc/init/rc.conf` which starts the legacy SystemV init system.

5.2 Shutdown

5.2.1 *Observations*

There are some important points related to system shutdown:

- Upstart never shuts down itself

  Upstart will "die" when the system is powered off, but if *it* ever exits, that is a bug.
- Upstart never stops a job with no stop on condition.
- Ubuntu employs both Upstart and SysV jobs.

  Ubuntu currently employs a hybrid system where core services are handled by Upstart, but additional services can be run in the legacy SystemV mode. This may seem odd, but consider that there are thousands of packages available in Ubuntu via the Universe and Multiverse repositories and *hundreds* of services. To avoid having to change every package to work with Upstart, Upstart allows packages to utilize their existing SystemV (and thus Debian-compatible) scripts.

5.2.2 *Shutdown Process*

To initiate a shutdown, perform one of the following actions:

- Click "Shut Down..." (or equivalent) in your graphical environment (for example Gnome)
- Run the shutdown(8) command, for example:

  ```
  # shutdown -h now
  ```

The following steps will now be taken:

1. Assuming the current runlevel is "2", either of the actions above will cause Upstart to emit the runlevel(7) event like this:

   ```
   runlevel RUNLEVEL=0 PREVLEVEL=2
   ```

2. The job `/etc/init/rc.conf` will be run.

   This job calls `/etc/init.d/rc` passing it the new runlevel ("`0`").

3. The SystemV system will then invoke the necessary scripts in `/etc/rc0.d/` to stop SystemV services.

4. One of the scripts run is `/etc/init.d/sendsigs`.

   This script will kill any remaining processes not already stopped (including Upstart processes).

5.3 Reboot

To initiate a reboot, perform one of the following actions:

- Click "Restart..." (or equivalent) in your graphical environment (for example Gnome)
- Run the shutdown(8) command specifying the "`-r`" option, for example:

  ```
  # shutdown -r now
  ```

- Run the reboot(8) command:

  ```
  # reboot
  ```

The following will steps will now be taken:

1. Assuming the current runlevel is "2", whichever command is run above will cause Upstart to emit the runlevel(7) event like this:

   ```
   runlevel RUNLEVEL=6 PREVLEVEL=2
   ```

2. The job `/etc/init/rc.conf` will be run.

   This job calls `/etc/init.d/rc` passing it the new runlevel ("6").

3. The SystemV system will then invoke the necessary scripts in `/etc/rc6.d/` to stop SystemV services.

4. One of the scripts run is `/etc/init.d/sendsigs`.

   This script will kill any remaining processes not already stopped (including Upstart processes).

5.4 Single-User Mode

When booting direct into single-user mode, the `runlevel` command will show:

```
# runlevel
N S
```

See Runlevels.

5.5 Recovery Mode ( )

Ubuntu provides a recovery mode in case your system experiences problems. This is handled by the `friendly-recovery` package. If you select a "*recovery mode*" option on the Grub menu. This makes the initramfs pass a flag to Upstart which ensures that the */etc/init/friendly-recovery.conf* Upstart job is the first job run after Upstart starts. As a result, this job has full control over the system and provides a friendly menu that allows users to check disks with fsck(8), repair your package database and so on.

5.6 Failsafe Mode ( )

This is a new phase introduced in Ubuntu 11.10 that borrows an idea from Google's Chrome OS. A new job called *failsafe* has been introduced that checks to ensure the system has reached a particular state. If the expected state is not attained, the job reboots the system automatically.

6 Configuration

This section lists a number of job configuration file stanzas, giving example usage for each. The reference for your specific version of Upstart will be available in the init(5) man page. [14]

6.1 Stanzas by Category

Configuration Stanzas by Category (detail in brackets show version of Upstart stanza added)

Category	Stanzas	Added in Version
Process Definition	exec	
	pre-start	
	post-start	
	pre-stop	
	post-stop	
	script	
Event Definition	manual	0.6.7
	start on	
	stop on	
Job Environment	env	
	export	
Services, tasks and respawning	normal exit	
	respawn	
	respawn limit	
	task	
Instances	instance	

Documentation	author	
	description	
	emits	
	version	
	usage	1.5
Process environment	console none	
	console log	1.4
	console output	
	console owner	
	chdir	
	chroot	
	limit	
	nice	
	oom score	
	setgid	1.4
	setuid	1.4
	umask	
Process Control	expect fork	
	expect daemon	
	expect stop	
	kill signal	1.3
	kill timeout	

6.2 author

Syntax:

```
author <string>
```

Quoted name (and maybe contact details) of author of this Job Configuration File.

Example:

```
author "Scott James Remnant <scott@netsplit.com>"
```

6.3 console

For all versions of Upstart prior to v1.4, the default value for `console` was `console none`. As of Upstart 1.4, the default value is `console log`. If you are using Upstart 1.4 or later and wish to retain the old default, boot specifying the `--no-log` command-line option. An alternative is to boot using the `--default-console <value>` option which allows the default `console` value for jobs to be specified. Using this option it is possible to set the default to `none` but still honour jobs that specify explicitly console log.

6.3.1 `console log`

Only honoured for *System Jobs*: if specified for user jobs, Upstart will treat the job as if it had specified console none.

Connects standard input to `/dev/null`. Standard output and standard error are connected to one end of a pseudo-terminal such that any job output is automatically logged to a file in directory `/var/log/upstart/`. This directory can be changed by specifying the `--logdir <directory>` command-line option.

6.3.2 `console none`

Connects the job's standard input, standard output and standard error file descriptors to `/dev/null`.

6.3.3 `console output`

Connects the job's standard input, standard output and standard error file descriptors to the console device.

6.3.3.1 Example of `console output`

```
console output

pre-start script

  # Perform whatever checks you like here (maybe checking
  # '/etc/default/foo' to see if the service is enabled # or not).
  #
  # if there are no problems detected, simply "exit 0", else do
  # something like this...

  # display an error message to stderr *on the console* and also write
  # the same message to the system log.
  logger -is -t "$UPSTART_JOB" "ERROR: foo!"

  # tell Upstart not to start the main process for the job.
  exit 1
end script

# this service doesn't do much :-)
exec sleep 999
```

See pre-start.

6.3.4 `console owner`

Identical to console output except that additionally it makes the job the owner of the console device. This means it will receive certain signals from the kernel when special key combinations such as Control-C are pressed.

6.4 `chdir`

Syntax:

```
chdir <directory>
```

Runs the job's processes with a working directory in the specified directory instead of the root of the filesystem.

Example:

```
chdir /var/mydaemon
```

6.5 chroot

Syntax:

```
chroot <directory>
```

Runs the job's processes in a chroot(8) environment underneath the specified directory.

Note that the specified directory must have all the necessary system libraries for the process to be run, often including `/bin/sh`.

Example:

```
chroot /srv/chroots/oneiric
```

6.6 description

Syntax:

```
description <string>
```

One line quoted description of Job Configuration File. For example:

```
description "OpenSSH server"
```

6.7 emits

Syntax:

```
emits <values>
```

Specifies the events the job configuration file generates (directly or indirectly via a child process). This stanza can be specified multiple times for each event emitted. This stanza can also use the following shell wildcard meta-characters to simplify the specification:

- asterisk ("`*`")
- question mark ("`?`")
- square brackets ("`[`" and "`]`")

For example, upstart-udev-bridge can emit a large number of events. Rather than having to specify every possible event, since the form of the event names is consistent, a single `emits` stanza can be specified to cover all possible events:

```
emits *-device-*
```

Further Examples:

```
emits foo-event bar-event wibble-event
emits hello
```

6.8 end script

This psuedo-stanza acts as a terminator for script sections:

- script.
- pre-start script.
- post-start script.
- pre-stop script.
- post-start script.

6.9 env

Syntax:

```
env KEY[=VALUE]
```

Allows an environment variable to be set which is accessible in all script sections.

Example:

```
env myvar="hello world"

script
  echo "myvar='$myvar'" > /run/script.log
end script
```

See Environment Variables.

6.10 exec

Syntax:

```
exec COMMAND [ ARG ]...
```

Stanza that allows the specification of a single-line command to run. Note that if this command-line contains any shell meta-characters, it will be passed through a shell prior to being executed. This ensures that shell redirection and variable expansion occur as expected.

Example:

```
exec /usr/bin/my-daemon --option foo -v
```

6.11 expect

> ***Warning***
>
> This stanza is *extremely* important: read this section carefully!

Upstart will keep track of the process ID that it thinks belongs to a job. If a job has specified the instance stanza, Upstart will track the PIDs for each unique instance of that job.

If you do not specify the `expect` stanza, Upstart will track the life cycle of the *first* PID that it executes in the exec or script stanzas. However, most Unix services will "daemonize", meaning that they will create a new process (using fork(2)) which is a child of the initial process. Often services will "double fork" to ensure they have no association whatsoever with the initial process. (Note that no services will fork more than twice initially since there is no additional benefit in doing so).

In this case, Upstart must have a way to track it, so you can use expect fork, or expect daemon which allows Upstart to use ptrace(2) to "count forks".

To allow Upstart to determine the *final* process ID for a job, it needs to know how many times that process will call fork(2). Upstart itself cannot know the answer to this question since once a daemon is running, it could then fork a number of "worker" processes which could themselves fork any number of times. Upstart cannot be expected to know which PID is the "master" in this case, considering it does not know if worker processes will be created at all, let alone how many times, or how many times the process will fork initially. As such, it is necessary to *tell* Upstart which PID is the "master" or parent PID. This is achieved using the `expect` stanza.

The syntax is simple, but you do need to know *how many times your service forks*.

Note that most daemons fork twice.

If your daemon has a "don't daemonize" or "run in the foreground" mode, then it's much simpler to use that and not run with fork following. One issue with that though, is that Upstart will emit the `started JOB=yourjob` event as soon as it has executed your daemon, which may be before it has had time to listen for incoming connections or fully initialize.

A final point: the `expect` stanza *only* applies to exec and script stanzas: it has *no* effect on pre-start and post-start.

It's important to note that the "`expect`" stanza is thus being used for two different but complementary tasks:

- Identifying service readiness.
- PID tracking.

6.11.1 `expect fork`

Upstart will expect the process executed to call fork(2) exactly *once*.

Some daemons fork a new copy of themselves on `SIGHUP`, which means when the Upstart reload command is used, Upstart will lose track of this daemon. In this case, `expect fork` cannot be used. See Daemon Behaviour.

6.11.2 `expect daemon`

Upstart will expect the process executed to call fork(2) exactly *twice*.

6.11.3 `expect stop`

Specifies that the job's main process will raise the SIGSTOP signal to indicate that it is ready. init(8) will wait for this signal before running the job's post-start script, or considering the job to be running.

6.11.4 How to Establish Fork Count

If the application you are attempting to create a Job Configuration File does not document how many times it forks, you can run it with a tool such as strace(1) which will allow you to count the number of forks. For example:

```
# Trace all children of /usr/bin/myapp
$ sudo strace -o /tmp/strace.log -fFv /usr/bin/myapp --arg foo --hello wibble &

# After allowing some "reasonable" time for the app to start, kill it and strace
$ sudo killall -9 strace

# Display the number of forks
#
#   1 => specify "expect fork"
#   2 => specify "expect daemon"
#
$ sudo egrep "\<(fork|clone)\>\(" /tmp/strace.log | wc | awk '{print $1}'
```

6.11.5 Implications of Misspecifying `expect`

The table below summarizes the behaviour resulting for every combination of expect stanza and number of fork(2) calls:

Expect Stanza Behaviour

	Specification of Expect Stanza		
Forks	`no expect`	`expect fork`	`expect daemon`
0	Correct	start hangs	start hangs
1	Wrong pid tracked †	Correct	start hangs
2	Wrong pid tracked †	Wrong pid tracked †	Correct

Key:

'†' - No PID will be displayed.

6.11.6 Recovery on Misspecification of `expect`

6.11.6.1 When `start` hangs

The start command will "hang" if you have misspecified the expect stanza by telling Upstart to expect more fork(2) calls than your application actually makes.

To resolve the situation:

1. Interrupt the start command by using "`CONTROL+c`" (or sending the process the `SIGINT` signal).
2. Run the initctl status command for your job. You will see something like:

   ```
   myjob start/spawned, process 1234
   ```

   You'll notice that the PID shown *is* actually correct since Upstart has tracked the initial PID.

3. Kill(1) the PID of your application.
4. Re-run the initctl status command for your job. You will see something like:

```
myjob stop/waiting
```

5. Correct the expect stanza specification in the job configuration file.

6.11.6.2 When Wrong PID is Tracked

If you have misspecified the expect stanza by telling Upstart to expect fewer fork(2) calls than your application actually makes, Upstart will be unable to manage it since it will be looking at the wrong PID. The start command *will* start your job, but it will show unexpected output (the goal and state will be shown as `stop/waiting`).

To resolve the situation:

1. Run the initctl status command for your job. You will see something like:

```
myjob stop/waiting
```

   Notice that no PID is displayed.

2. Find your jobs PID using ps(1). (If you're struggling to find it, remember that the parent PID will always be "`1`").
3. Kill(1) the PID of your application.
4. Correct the expect stanza specification in the job configuration file.

6.12 `export`

Export variables previously set with env to *all* events that result from this job. See for example Job Lifecycle.

Note that *no* leading dollar sign ($) is specified.

Example:

```
env myvar="hello world"
export myvar
```

6.13 `instance`

Sometimes you want to run the same job, but with different arguments. The variable that defines the unique instance of this job is defined with `instance`.

6.13.1 A Simple Instance Example

Let us start with a simple example which we will call "`foo.conf`":

```
instance $BAR

script
  . /etc/default/myapp-${BAR}

  echo "hello from instance $BAR"
```

```
  sleep 999
end script
```

The example above defines an instance job by specifying the `instance` stanza followed by the *name* of a variable (note that you *MUST* specify the dollar sign ('$').

Note that the **entire** job *is* the instance job: providing the `instance` stanza allows Upstart to make each running version of this job unique.

The job first sources an instance-specific configuration file ("`myapp-${BAR}`") then displays a message. Note again that we're now *using* that instance variable $BAR.

So, let's start an instance of this job:

```
$ sudo start foo
start: Unknown parameter: BAR
```

Oops! We forgot to specify the particular value for the `BAR` variable which makes each instance unique. Lets try again:

```
$ sudo start foo BAR=bar
foo (bar) start/running, process 1234
```

So, we now have one instance running. Let's start another:

```
$ sudo start foo BAR=bar
start: Job is already running: foo (bar)
```

Oops! We tried to run another instance with the same instance name (well, the same value of the `BAR` variable technically). Lets try again:

```
$ sudo start foo BAR=baz
foo (baz) start/running, process 1235
```

Okay. We should now have two instance running, but let us confirm that:

```
$ initctl list | grep ^foo
foo (bar) start/running, process 1234
foo (baz) start/running, process 1235
```

Good - Upstart is running two instances as expected. Notice the instance name in brackets after the job name in the initctl output above.

We will start one more instance:

```
$ sudo start foo BAR="hello world"
$ initctl list | grep ^foo
foo (bar) start/running, process 1234
foo (baz) start/running, process 1235
foo (hello world) start/running, process 1236
```

Let's try to stop the instances:

```
$ sudo stop foo
stop: Unknown parameter: BAR
```

That fails as Upstart needs to know *which* instance to stop and we didn't specify an instance value for the `BAR` instance variable. Rather than stopping each instance in turn, let's script it so that we can stop then all in one go:

```
$ initctl list | grep "^foo " | cut -d\( -f2 | cut -d\) -f1 | while read i
do
  sudo stop foo BAR="$i"
done
foo stop/waiting
foo stop/waiting
foo stop/waiting
$
```

All unique instances of the `foo` job are now stopped.

6.13.2 Another Instance Example

Lets say that once `memcached` is up and running, we want to start a queue worker for each directory in `/var/lib/queues`:

```
# queue-workers

start on started memcached

task

script
  for dir in `ls /var/lib/queues` ; do
    start queue-worker QUEUE=$dir
  done
end script
```

And now:

```
# queue-worker

stop on stopping memcached

respawn

instance $QUEUE

exec /usr/local/bin/queue-worker $QUEUE
```

In this way, Upstart will keep them all running with the specified arguments, and stop them if `memcached` is ever stopped.

The `instance` stanza is designed to make a running job unique.

Notes:

- the stanza isn't restricted to a single value. You can do silly things like the following if you wish:

```
instance ${myvar1}hello${myvar2}-foo/\wibble${var3}{$JOB}
```

  See Multiple Running Job Instances Without PID for another crazy real-life example.

- You *must* include at least one variable and it *must* have a leading dollar sign ($):

```
# GOOD (value can be changed by specifying different values
# for the variable called 'foo')
instance $foo

# BAD (value will always be the string literal "foo")
instance foo
```

- If you attempt to start a job with the `instance` stanza, but forget to provide the required variables, you will get an error since Upstart cannot then guarantee uniqueness. For example, if you have a job configuration file `foo.conf` such as this:

```
instance $bar

script
  sleep 999
end script
```

  Attempting to start it *without* specifying a value for foo will fail:

```
# start foo
start: Unknown parameter: bar
```

  Let's try again:

```
# start foo bar=1
foo (1) start/running, process 30003
```

  And now let's start another instance:

```
# start foo bar="hello 1,2,3"
foo (hello 1,2,3) start/running, process 30008
```

  Finally, let's see the current state of our two job instances:

```
$ initctl list|grep ^foo
foo (1) start/running, process 30003
foo (hello 1,2,3) start/running, process 30008
```

6.13.3 Starting an Instance Job Without Specifying an Instance Value

Note that if you have a job which makes use of `instance` but which may need to be run manually by an administrator, it is possible to "cheat" and allow them to start the job *without* specifying an explicit instance value:

```
# /etc/init/trickery.conf
start on foo

instance $UPSTART_EVENTS
env UPSTART_EVENTS=
```

Now, an Administrator can start this job as follows:

```
# start trickery
```

And this will work even if there is already a running instance of the `trickery` job (assuming the existing instance was started automatically).

This bit of trickery relies upon the fact that Upstart will set the `$UPSTART_EVENTS` environment variable before starting this job as a result of its start on condition becoming true. In this case, Upstart would therefore set `UPSTART_EVENTS='foo'`.

However, since the job sets a null default value for this variable, when an Administrator starts the job, `UPSTART_EVENTS` will be set to a null value. This empty value is enough to make that instance unique (since there are no other instances with a null instance value!)

See Environment Variables for details of `$UPSTART_EVENTS`.

6.14 kill signal

Specifies the stopping signal, `SIGTERM` by default, a job's main process will receive when stopping the running job.

Example:

```
kill signal INT
```

Note that if you are running an older version of Upstart without this feature, and you have an application which breaks with the normal conventions for shutdown signal, you can simulate it to some degree by using start-stop-daemon(8) with the `--signal` option:

```
start on some-event

env cmd=/usr/bin/foo

exec start-stop-daemon --start --exec $cmd

pre-stop exec start-stop-daemon --signal QUIT --stop --exec $cmd
```

6.15 kill timeout

The number of seconds Upstart will wait before killing a process. The default is 5 seconds.

Example:

```
kill timeout 20
```

6.16 `limit`

Provides the ability to specify resource limits for a job.

For example, to allow a job to open any number of files, specify:

```
limit nofile unlimited unlimited
```

> ***Note***
>
> If a user job specifies this stanza, it may fail to start should it specify a value greater than the users privilege level allows.

For further details on the available limits see init(5) and getrlimit(2).

6.17 `manual`

Added in Upstart v0.6.7

This stanza will tell Upstart to ignore the start on / stop on stanzas. It is useful for keeping the logic and capability of a job on the system while not having it automatically start at boot-up.

Example:

```
manual
```

6.18 `nice`

Change the jobs scheduling priority from the default. See nice(1).

Example:

```
# run with lowest priority
nice 19
```

6.19 `normal exit`

Used to change Upstart's idea of what a "normal" exit status is. Conventionally, processes exit with status "*0*" (zero) to denote success and non-zero to denote failure. If your application can exit with exit status "*13*" and you want Upstart to consider this as an normal (successful) exit, then you can specify:

```
normal exit 0 13
```

You can even specify signals. For example, to consider exit codes "*0*", "*13*" as success and also to consider the program to have completed successfully if it exits on signal "*SIGUSR1*" and "*SIGWINCH*", specify:

```
normal exit 0 13 SIGUSR1 SIGWINCH
```

6.20 oom score

Linux has an "Out of Memory" killer facility. This is a feature of the kernel that will detect if a process is consuming increasingly more memory. Once "triggered", the kernel automatically takes action by killing the rogue process to avoid it impacting the system adversely.

Normally the OOM killer regards all processes equally, this stanza advises the kernel to treat this job differently.

The "adjustment" value provided to this stanza may be an integer value from `-999` (very unlikely to be killed by the OOM killer) up to `1000` (very likely to be killed by the OOM killer). It may also be the special value `never` to have the job ignored by the OOM killer entirely (potentially dangerous unless you *really* trust the application in all possible system scenarios).

Example:

```
# this application is a "resource hog"
oom score 1000

expect daemon
respawn
exec /usr/bin/leaky-app
```

6.21 post-start

Syntax:

```
post-start exec|script
```

Script or process to run *after* the main process has been spawned, but before the started(7) event has been emitted.

Use this stanza when a delay (or some arbitrary condition) must be satisfied before an executed job is considered "started". An example is MySQL. After executing it, it may need to perform recovery operations before accepting network traffic. Rather than start dependent services, you can have a post-start like this:

```
post-start script
  while ! mysqladmin ping localhost ; do sleep 1 ; done
end script
```

6.22 post-stop

Syntax:

```
post-stop exec|script
```

There are times where the cleanup done in pre-start is not enough. Ultimately, the cleanup should be done both pre-start and post-stop, to ensure the service starts with a consistent environment, and does not leave behind anything that it shouldn't.

```
exec /some/directory/script
```

If it is possible, you'll want to run your daemon with a simple `exec` line. Something like this:

```
exec /usr/bin/mysqld
```

If you need to do some scripting before starting the daemon, script works fine here. Here is one example of using a script stanza that may be non-obvious:

```
# statd - NSM status monitor

description   "NSM status monitor"
author                "Steve Langasek <steve.langasek@canonical.com>"

start on (started portmap or mounting TYPE=nfs)
stop on stopping portmap

expect fork
respawn

env DEFAULTFILE=/etc/default/nfs-common

pre-start script
    if [ -f "$DEFAULTFILE" ]; then
        . "$DEFAULTFILE"
    fi

    [ "x$NEED_STATD" != xno ] || { stop; exit 0; }

    start portmap || true
    status portmap | grep -q start/running
    exec sm-notify
end script

script
    if [ -f "$DEFAULTFILE" ]; then
        . "$DEFAULTFILE"
    fi

    if [ "x$NEED_STATD" != xno ]; then
        exec rpc.statd -L $STATDOPTS
    fi
end script
```

Because this job is marked respawn, an exit of `0` is "ok" and will not force a respawn (only exiting with a non-`0` exit or being killed by an unexpected signal causes a respawn), this script stanza is used to start the optional daemon `rpc.statd` based on the defaults file. If `NEED_STATD=no` is in `/etc/default/nfs-common`, this job will run this snippet of script, and then the script will exit with `0` as its return code. Upstart will not respawn it, but just gracefully see that it has stopped on its own, and return to `stopped` status. If, however, `rpc.statd` had been run, it would stay in the `start/running` state and be tracked normally.

6.23 `pre-start`

Syntax:

```
pre-start exec|script
```

Use this stanza to prepare the environment for the job. Clearing out cache/tmp dirs is a good idea, but any heavy logic is discouraged, as Upstart job files should read like configuration files, not so much like complicated software.

```
pre-start script
  [ -d "/var/cache/squid" ] || squid -k
end script
```

Another possibility is to cancel the start of the job for some reason. One good reason is that it's clear from the system configuration that a service is not needed:

```
pre-start script
  if ! grep -q 'parent=foo' /etc/bar.conf ; then
    stop ; exit 0
  fi
end script
```

Note that the "`stop`" command did not receive any arguments. This is a shortcut available to jobs where the "`stop`" command will look at the current environment and determine that you mean to stop the current job.

6.23.1 `pre-start` example ( )

On Ubuntu, the common `pre-start` idiom is to use `/etc/default/myapp`, so the example would become:

```
pre-start script

  # stop job from continuing if no config file found for daemon
  [ ! -f /etc/default/myapp ] && { stop; exit 0; }

  # source the config file
  . /etc/default/myapp

  # stop job from continuing if admin has not enabled service in
  # config file.
  [ -z "$ENABLED" ] && { stop; exit 0; }

end script
```

This is safe since the job will not start (technically it won't progress beyond the `pre-start` stage) if:

- the config file does not exist.
- the config file has not been modified to enable the service.

Note that the example above assumes your applications configuration file is shell-compatible (in other words it contains `name="value"` entries). If this is not the case, just use grep(1) or similar:

```
enabled=$(grep ENABLED=1 $CONFIG)
[ -z "$enabled" ] && exit 0
```

Or something like this:

```
if ! grep -q DISABLED=false /etc/default/myapp; then
  stop ; exit 0
fi
```

See Example of console output for another of example where you can display an error message if the job detects it should not be started.

6.24 `pre-stop`

Syntax:

```
pre-stop exec|script
```

The `pre-stop` stanza will be executed *before* the job's stopping(7) event is emitted and **before the main process is killed**.

Stopping a job involves sending `SIGTERM` to it. If there is anything that needs to be done before `SIGTERM`, do it here. Arguably, services should handle `SIGTERM` very gracefully, so this shouldn't be necessary. However, if the service takes more than kill timeout seconds (default, 5 seconds) then it will be sent `SIGKILL`, so if there is anything critical, like a flush to disk, and raising kill timeout is not an option, pre-stop is not a bad place to do it. [16]

You can also use this stanza to cancel the stop, in a similar fashion to the way one can cancel the start in the pre-start.

6.25 `respawn`

> ***Note***
>
> If you are creating a new Job Configuration File, *do not* specify the `respawn` stanza until you are fully satisfied you have specifed the expect stanza correctly. If you *do*, you will find the behaviour potentially very confusing.

Without this stanza, a job that exits quietly transitions into the `stop/waiting` state, no matter how it exited.

With this stanza, whenever the main script/exec exits, without the goal of the job having been changed to `stop`, the job will be started again. This includes running pre-start, post-start and post-stop. Note that pre-stop will not be run.

There are a number of reasons why you may or may not want to use this. For most traditional network services this makes good sense. If the tracked process exits for some reason that wasn't the administrator's intent, you probably want to start it back up again.

Likewise, for tasks, (see below), respawning means that you want that task to be retried until it exits with zero (0) as its exit code.

One situation where it may seem like respawn should be avoided, is when a daemon does not respond well to `SIGTERM` for stopping it. You may believe that you need to send the service its shutdown command without Upstart being involved, and therefore, you don't want to use respawn because Upstart will keep trying to start your service back up when you told it to shutdown.

However, the appropriate way to handle that situation is a pre-stop which runs this shutdown command. Since the job's goal will already be 'stop' when a pre-stop is run, you can shutdown the process through any means, and the process won't be re-spawned (even with the respawn stanza).

6.26 `respawn limit`

Yes, this is *different* to a plain respawn: specifying `respawn limit` *does not* imply `respawn`.

Syntax:

```
respawn limit COUNT INTERVAL
```

Example:

```
# respawn the job up to 10 times within a 5 second period.
# If the job exceeds these values, it will be stopped and
# marked as failed.
respawn
respawn limit 10 5
```

Respawning is subject to a limit. If the job is respawned more than `COUNT` times in `INTERVAL` seconds, it will be considered to be having deeper problems and will be stopped. Default `COUNT` is `10`. Default `INTERVAL` is `5` seconds.

Note that this only applies to automatic respawns and not the restart(8) command.

6.27 `script`

Allows the specification of a multi-line block of shell code to be executed. Block is termined by end script.

6.28 `setgid`

Added in Upstart v1.4

Syntax:

```
setgid <groupname>
```

Changes to the group `<groupname>` before running the job's process.

> ***Warning***
>
> Note that *all* processes (pre-start, post-stop, *et cetera*) will be run with the group specified.

If this stanza is unspecified, the primary group of the user specified in the setuid block is used. If both stanzas are unspecified, the job will run with its group ID set to 0 in the case of system jobs, and as the primary group of the user in the case of User Jobs.

Example:

```
setgid apache
```

6.29 `setuid`

Added in Upstart v1.4

Syntax:

```
setuid <username>
```

Changes to the user `<username>` before running the job's process.

> ### *Warning*
>
> Note that *all* processes (pre-start, post-stop, *et cetera*) will be run as the user specified.

If this stanza is unspecified, the job will run as root in the case of system jobs, and as the user in the case of User Jobs.

Note that System jobs using the setuid stanza are still system jobs, and can not be controlled by an unprivileged user, even if the setuid stanza specifies that user.

6.30 `start on`

This stanza defines the set of Events that will cause the Job to be automatically started.

Syntax:

```
start on EVENT [[KEY=]VALUE]... [and|or...]
```

Each event `EVENT` is given by its name. Multiple events are permitted using the operators "`and`" and "`or`" and complex expressions may be performed with parentheses (within which line breaks are permitted).

You may also match on the environment variables contained within the event by specifying the `KEY` and expected `VALUE`. If you know the order in which the variables are given to the event you may omit the `KEY`.

`VALUE` may contain wildcard matches and globs as permitted by fnmatch(3) and may expand the value of any variable defined with the env stanza.

Negation is permitted by using "`!=`" between the `KEY` and `VALUE`.

Note that if the job is *already running* and is not an instance job, if the `start on` condition becomes true (again), no further action will be taken.

Note that the `start on` stanza expects a token to follow *on the same line*. Thus:

```
# ERROR: invalid
start on
  foo or bar

# OK
start on foo or bar
```

If no environment variables are specified via `KEY` to restrict the match, the condition will match all instances of the specified event.

See Really understanding start on and stop on for further details.

6.30.1 Normal start

If you are just writing an upstart job that needs to start the service after the basic facilities are up, either of these will work:

```
start on (local-filesystems and net-device-up IFACE!=lo)
```

or:

```
start on runlevel [2345]
```

The difference in whether to use the more generic 'runlevel' or the more explicit local-filesystems(7) and `net-device-up` events should be guided by your job's behaviour. If your service will come up without a valid network interface (for instance, it binds to `0.0.0.0`, or uses setsockopt(2) `SO_FREEBIND`), then the `runlevel` event is preferable, as your service will start a bit earlier and start in parallel with other services.

However if your service requires that a non-loopback interface is configured for some reason (i.e., it will not start without broadcasting capabilities), then explicitly saying "once a non loopback device has come up" can help.

In addition, services may be aggregated around an abstract job, such as `network-services`:

```
start on started network-services
```

The network-services job is a generic job that most network services should follow in releases where it is available. [15] This allows the system administrator and/or the distribution maintainers to change the general startup of services that don't need any special case start on criteria.

We use the started(7) event so that anything that must be started before all network services can do "`start on starting network-services`".

6.30.2 Start depends on another service

```
start on started other-service
```

6.30.3 Start must precede another service

```
start on starting other-service
```

Example: your web app needs `memcached` to be started before `apache`:

```
start on starting apache2
stop on stopped apache2
respawn

exec /usr/sbin/memcached
```

6.31 `stop on`

This stanza defines the set of Events that will cause the Job to be automatically stopped if it is already running.

Syntax:

```
stop on EVENT [[KEY=]VALUE]... [and|or...]
```

Like the stop on stanza, `start on` expects a token to follow *on the same line*:

```
# ERROR: invalid
stop on
  foo or bar

# OK
stop on foo or bar
```

See start on for further syntax details.

6.31.1 Normal shutdown

```
stop on runlevel [016]
```

Or if a generic job is available such as `network-services` [15]

```
stop on stopping network-services
```

6.31.2 Stop before depended-upon service

```
stop on stopping other-service
```

Note that this also will stop when `other-service` is restarted, so you will generally want to couple this with the start on condition:

```
start on started other-service
```

6.31.3 Stop after dependent service

```
stop on stopped other-service
```

6.32 `task`

In concept, a task is just a short lived job. In practice, this is accomplished by changing how the transition from a goal of "stop" to "start" is handled.

Without the 'task' keyword, the events that cause the job to start will be unblocked as soon as the job is *started*. This means the job has emitted a starting(7) event, run its pre-start, begun its script/exec, and post-start, and emitted its started(7) event.

With task, the events that lead to this job starting will be blocked until the job has completely transitioned back to *stopped*. This means that the job has run up to the previously mentioned started(7) event, *and* has also completed its post-stop, and emitted its stopped(7) event.

Typically, `task` is for something that you just want to run and finish completely when a certain event happens.

```
# pre-warm-memcache

start on started memcached

task

exec /path/to/pre-warm-memcached
```

So you can have another job that starts your background queue worker once the local memcached is pre-warmed:

```
# queue-worker

start on stopped pre-warm-memcache
stop on stopping memcached

respawn

exec /usr/local/bin/queue-worker
```

The key concept demonstrated above is that we "`start on stopped pre-warm-memcache`". This means that we don't start until the task has completed. If we were to use `started` instead of `stopped`, we would start our queue worker as soon as `/path/to/pre-warm-memcached` had been started running.

We could also accomplish this without mentioning the pre-warm in the queue-worker job by doing this:

```
# queue-worker

start on started memcached
stop on stopping memcached

respawn

exec /usr/local/bin/queue-worker

# pre-warm-memcache

start on starting queue-worker
task
exec /path/to/pre-warm-memcache
```

If we did not use "`task`" in the above example, queue-worker would be allowed to start as soon as we executed `/path/to/pre-warm-memcache`, which means it might potentially start before the cache was warmed.

6.33 umask

Syntax:

```
umask <value>
```

Set the file mode creation mask for the process. `<value>`" should be an octal value for the mask. See umask(2) for more details.

Example:

```
umask 0002
```

6.34 `usage`

Brief message explaining how to start the job in question. Most useful for instance jobs which require environment variable parameters to be specified before they can be started.

Syntax:

```
usage <string>
```

Example:

```
instance $DB
usage "DB - name of database instance"
```

If a job specifies the `usage` stanza, attempting to start the job without specifying the correct variables will display the usage statement. Additionally, the usage can be queried using initctl usage.

6.35 `version`

Syntax:

```
version <string>
```

This stanza may contain version information about the job, such as revision control or package version number. It is not used or interpreted by init(8) in any way.

Example:

```
version "1.0.2a-beta4"
```

7 Command-Line Options

The table below lists the command-line options accepted by the Upstart init daemon.

> ***Warning***
>
> Under normal conditions, you should not need to specify *any* command-line options to Upstart. A number of these options were added specifically for testing Upstart itself and if used without due care can stop your system from booting (for example specifying `--no-startup-event`). Therefore you should be *extremely* careful specifying *any* command-line options to Upstart unless you understand the implications of doing so.

Command-line Options

Option Name	Description	Added in Version
`--confdir=DIR`	Specify alternate configuration directory (default: `/etc/init/`)	1.3
`--debug`	Enable Informational and debug messages	0.1.0
`--default-console=VALUE`	Specify default value for jobs not specifying console (default: `none` (Upstart < 1.4), else `log`)	1.4
`--help`	Show usage statement for init	0.1.0
`--logdir=DIR`	Specify alternate log directory (default: `/var/log/upstart/`)	1.4
`--no-log`	Disable job logging (all job output is discarded)	1.4
`--no-sessions`	Disable user sessions (and chroot support)	1.3
`--no-startup-event`	Disable emitting an event at startup	1.3
`-q` , `--quiet`	Reduce output to errors only	0.1.0
`--session`	Use D-Bus session bus rather than D-Bus system bus	1.3
`--startup-event=NAME`	Specify an alternative initial event (default: `startup` event)	1.3
`-v` , `--verbose`	Increase output to include informational messages	0.1.0
`--version`	Display version information	0.1.0

Notes:

- An alternative to `--debug` and `--verbose` is to modify the message level at runtime by using initctl log-priority.

8 Explanations

8.1 Really understanding `start on` and `stop on`

(Note: This section focuses on start on, but the information also applies to stop on unless explicitly specified).

The `start on` stanza needs careful contemplation. Consider this example:

```
start on started mysql
```

The syntax above is actually a short-hand way of writing:

```
start on started JOB=mysql
```

Remember that started(7) is an event which Upstart emits automatically when the `mysql` job has *started to run*. The whole start on stanza can be summarized as:

```
start on <event> [<vars_to_match_event_on>]
```

Where `<vars_to_match_event_on>` is optional, but if specified comprises one or more variables.

A slight variation of the above:

```
start on started JOB=mydb DBNAME=foobar
```

This example shows that the fictitious job above would only be started when the `mydb` database server brings the `foobar` database on-line. Correspondingly, file `/etc/init/mydb.conf` would need to specify "`export DBNAME`" and be started like this:

```
start mydb DBNAME=foobar
```

Looking at a slightly more complex real-life example:

```
# /etc/init/alsa-mixer-save.conf
start on starting rc RUNLEVEL=[06]
```

This job says,

> **"Run when the `rc` job emits the starting(7) event, but only if the**
>
> environment variable `RUNLEVEL` equals either `0` (halt) or `6` (reboot)".

If we again add in the implicit variable it becomes clearer:

```
# /etc/init/alsa-mixer-save.conf
start on starting JOB=rc RUNLEVEL=[06]
```

But where does the `RUNLEVEL` environment variable come from? Well, variables are exported in a job configuration file to related jobs. Thus, the answer is The rc Job.

If you look at this job configuration file, you will see, as deduced:

```
export RUNLEVEL
```

8.1.1 The `rc` Job

The `rc` job configuration file is well worth considering:

```
# /etc/init/rc.conf
start on runlevel [0123456]
stop on runlevel [!$RUNLEVEL]

export RUNLEVEL
export PREVLEVEL

console output
env INIT_VERBOSE

task

exec /etc/init.d/rc $RUNLEVEL
```

It says in essence,

> "Run the SysV init script as `/etc/init.d/rc $RUNLEVEL` when telinit(8) emits the runlevel(7) event for any runlevel".

However, note the stop on condition:

```
stop on runlevel [!$RUNLEVEL]
```

This requires some explanation. The manual page for runlevel(7) explains that the `runlevel` event specifies two variables in the following order:

- `RUNLEVEL`

  The new "goal" runlevel the system is changing to.

- `PREVLEVEL`

  The previous system runlevel (which may be set to an empty value).

Thus, the stop on condition is saying:

> "Stop the `rc` job when the `runlevel` event is emitted *and* the `RUNLEVEL` variable matches '`[!$RUNLEVEL]`'.

This admittedly does initially appear nonsensical. The way to read the statement above though is:

> "Stop the `rc` job when the `runlevel` event is emitted and the `RUNLEVEL` variable is *not* set to the *current value* of the `RUNLEVEL` variable."

So, if the runlevel is currently "2" (full graphical multi-user under Ubuntu), the `RUNLEVEL` variable will be set to `RUNLEVEL=2`. The condition will thus evaluate to:

```
stop on runlevel [!2]
```

This is just a safety measure. What it is saying is:

- *if* the `rc` job (which is a short-running Task) is still running when the system changes to a *different* runlevel (a runlevel other than "2" here), Upstart will stop it.
- If it is *not* running when the system changes to a different runlevel, no action will be taken to stop the job (since it has already stopped).

However, note that when the system moves to a new runlevel, Upstart will then immediately *re-run* the job at the *new* runlevel since the start on condition specifies that this job should be started in *every* runlevel.

Since this job has specified the `runlevel` event, it automatically gets access to the variables set by this event (`RUNLEVEL` and `PREVLEVEL`). However, note that these two variables are also exported. The reason for this is to allow other jobs which start on or stop on the `rc` job to make use of these variables (which were set by the `runlevel` event).

See runlevel(7) for further details.

8.2 Environment Variables

Upstart allows you to set environment variables which will be accessible to the jobs whose job configuration files they are defined in. Environment variables are set using the env keyword.

For example:

```
# /etc/init/env.conf
env TESTING=123

script
  # prints "TESTING='123'" to system log
  logger -t $0 "TESTING='$TESTING'"
```

```
end script
```

Further, we can pass environment variables defined in *events* to jobs using the env stanza and the export stanza. Assume we have two job configuration files, `A.conf` and `B.conf`:

```
# /etc/init/A.conf
start on wibble
export foo

# /etc/init/B.conf
start on A
script
  logger "value of foo is '$foo'"
end script
```

If we now run the following command, both jobs `A` and `B` will run, causing `B` to write "`value of foo is 'bar'`" to the system log:

```
# initctl emit wibble foo=bar
```

Note that a variables value can always be overridden by specifying a new value on the command-line. For example:

```
start on wibble
env var=hello

script
  logger "value of var is '$var'"
end script
```

When we emit the required event...:

```
# initctl emit wibble var=world
```

... the system log will have recorded:

```
value of var is 'world'
```

Note that a Job Configuration File does *not* have access to a user's environment variables, not even the superuser. This is not possible since all job processes created are children of `init` which does not have a user's environment.

However, using the technique above, it is possible to inject a variable from a user's environment into a job indirectly:

```
# initctl emit wibble foo=bar USER=$USER
```

As another example of environment variables, consider this job configuration file [16]:

```
env var=bar
export var
```

```
pre-start script
  logger "pre-start: before: var=$var"

  var=pre-start
  export var

  logger "pre-start: after: var=$var"
end script

post-start script
  logger "post-start: before: var=$var"

  var=post-start
  export var

  logger "post-start: after: var=$var"
end script

script
  logger "script: before: var=$var"

  var=main
  export var

  logger "script: after: var=$var"
end script

post-stop script
  logger "post-stop: before: var=$var"

  var=post-stop
  export var

  logger "post-stop: after: var=$var"
end script
```

This will generate output in your system log as follows (the timestamp and hostname have been removed, and the output formatted to make it clearer):

```
logger: pre-start:  before: var=bar
logger: pre-start:   after: var=pre-start

logger: post-start: before: var=bar
logger: post-start:  after: var=post-start

logger: script:     before: var=bar
logger: script:      after: var=main

logger: post-stop:  before: var=bar
logger: post-stop:   after: var=post-stop
```

As shown, every script section receives the value of `$var` as `bar`, but if any script section changes the value, it only affects *that* particular script sections copy of the variable. To summarize:

A script section cannot modify the value of a variable defined in a job configuration file *for other script sections*.

8.2.1 Restrictions

Environment variables do not expand in start on or stop on conditions:

```
env FOO=bar
start on $FOO
```

This will start the job in question when the "$FOO" event is emitted, **not** when the event "*bar*" is emitted:

```
# job above *NOT* started
initctl emit bar

# job above started!
initctl emit '$FOO'
```

Similarly, the following will not work:

```
start on starting $FOO
start on starting JOB=$FOO
```

8.2.2 Standard Environment Variables

The table below shows all variables set by Upstart itself. Note that variables prefixed by "UPSTART_" are variables set within a *jobs* environment, whereas the remainder are set within an events environment (see the following table).

Upstart Environment Variables.

Variable	Brief Description	Details
EXIT_SIGNAL	Signal causing job to exit	String such as "HUP" or "TERM", or numeric for unknown signals
EXIT_STATUS	Exit code of job	
INSTANCE	Instance name of $JOB	Variable set but with no value if instance stanza not specified
JOB	Name of job	
PROCESS	Name of Job process type	"main", "pre-start", "post-start", "pre-stop", "post-stop" or "respawn"
RESULT	Whether job was successful	"ok" or "failed"
UPSTART_EVENTS	Events that caused job to start	Space-separated. Event environment not provided
UPSTART_FDS	File descriptor	Number of the file descriptor corresponding to the listening socket-event(7) socket
UPSTART_INSTANCE	Instance name of $UPSTART_JOB	

UPSTART_JOB	Name of current job	
UPSTART_STOP_EVENTS	Events that caused job to stop	Space-separated. Event environment not provided

The following table lists the variables from the table above which are set when job events are emitted, and which are thus available from within a jobs environment.

Environment Variables by Event.

Event	Variables Set in Event Environment
starting(7)	• INSTANCE • JOB
started(7)	• INSTANCE • JOB
stopping(7)	• INSTANCE • JOB • RESULT • PROCESS * • EXIT_STATUS † • EXIT_SIGNAL †
stopped(7)	• INSTANCE • JOB • RESULT • PROCESS * • EXIT_STATUS † • EXIT_SIGNAL †

Notes that some variables (those marked with '*' and '†') are only set when the job fails:

- PROCESS will always be set.
- Either EXIT_STATUS or EXIT_SIGNAL will be set.

Note carefully the distinction between JOB and UPSTART_JOB. If a job "bar.conf" specifies a start on condition of:

```
start on starting foo
```

and does not specify the instance stanza, when job "foo" starts, the environment of the "bar" job will contain:

```
JOB=foo
UPSTART_JOB=bar
UPSTART_EVENTS=starting
INSTANCE=
```

8.3 Job with Multiple Duplicate Stanzas

The way in which Upstart parses the job configuration files means that "the last entry wins". That is to say, every job configuration file must be syntactically correct, but if you had a file such as:

```
start on event-A
start on starting job-B
start on event-C or starting job-D
```

This job will have a start on condition of:

```
start on event-C or starting job-D
```

...since that is the last start on condition specified.

For start on, stop on and `emits` stanzas, you can confirm Upstart's decision, you can use the `initctl show-config` command like this:

```
initctl show-config myjob
```

For the example above, the output would be:

```
start on event-C or starting job-D
```

8.4 Job Specifying Same Condition in `start on` on `stop on`

See Ordering of Stop/Start Operations.

9 Features

9.1 D-Bus Service Activation

As of D-Bus version 1.4.1-0ubuntu2 (in Ubuntu), you can have Upstart start a D-Bus service rather than D-Bus. This is useful because it is then possible to create Upstart jobs that start or stop when D-Bus services start.

See Run a Job When a User Logs in for an example.

10 Tools

Upstart provides a number of additional tools to:

- help manage your system
- create Upstart events from other sources

10.1 Utilities

10.1.1 `reload`

Symbolically linked to initctl, causing the following to be run:

```
initctl reload <job>
```

This will send a running job the `SIGHUP` signal. By convention, daemons receiving this signal reload their configuration or in some way re-initialize themselves (keeping the same PID).

10.1.2 `restart`

Symbolically linked to initctl, causing the following to be run:

```
initctl restart <job>
```

Stops and then starts a job.

10.1.3 `runlevel`

See Runlevels.

10.1.4 `start`

Symbolically linked to initctl, causing the following to be run:

```
initctl start <job>
```

Starts a job.

10.1.4.1 Attempting to Start an Already Running Job

If you try to start a job that is already running and which does *not* specify the instance stanza, you will get the following error:

```
# start myjob
start: Job is already running: myjob
```

10.1.4.2 Attempting to Start a Job that requires an Instance Variable

If you try to start a job that specifies the instance stanza, you will need to specify the appropriate variable. If you do not, you will get an error. For example, assuming `myjob.conf` specified `instance $foo`:

```
# start myjob
start: Unknown parameter: foo
```

To resolve this, specify some value for the variable in question:

```
# start myjob foo="hello, world"
```

10.1.5 `stop`

Symbolically linked to initctl, causing the following to be run:

```
initctl stop <job>
```

Stops a job.

10.1.5.1 Attempting to Stop an Already Stopped Job

If you try to stop a job that is not running, you will get the following error:

```
# stop myjob
stop: unknown instance
```

10.1.5.2 Attempting to Stop a Job that requires an Instance Variable

If you try to stop a job that specifies the instance stanza without specifying the particular instance you wish to stop, you will get an error:

```
# stop myjob
stop: Unknown parameter: foo
```

To resolve this, specify the value for the variable in question:

```
# stop myjob foo=...
```

Where "`...`" must be replaced by a legitimate value for one of the instances as specified in the output of "`initctl status myjob`".

10.1.6 `initctl`

This is the primary command used by users and Administrators to interact with Upstart.

- Run `initctl help` to see the available commands.
- Run `initctl --help` to see the overall options available.
- Run `initctl <command> --help` to see options for the specified command.

Commands to manipulate jobs:

- reload
- restart
- start
- stop

10.1.6.1 `initctl` Commands Summary

Summary of initctl commands

Command	Description	Added in Version
initctl check-config	Check for unreachable jobs/event conditions	1.3
initctl emit	Emit an event	0.3.0
initctl help	Display list of commands	0.3.0
initctl list	List known jobs	0.2.0
initctl log-priority	Change the minimum priority of log messages displayed by the init daemon	0.3.8
initctl notify-disk-writeable	Inform Upstart that disk is now writeable	1.5
initctl reload	Send HUP signal to job	0.6.5
initctl reload-configuration	Reload the configuration	0.6.0

initctl restart	Restart job	0.6.0
initctl show-config	Show emits, start on and stop on details for job(s)	1.3
initctl start	Start job	0.1.0
initctl status	Query status of job	0.1.0
initctl stop	Stop job	0.1.0
initctl usage	Show job usage message if available	1.5
initctl version	Request the version of the init daemon	0.3.8

10.1.6.2 *initctl check-config*

The `initctl check-config` command can be used to check that the events and jobs a job configuration file references are "known" to the system. This is important, since if a System Administrator were to inadvertently force the removal of a package, or inadvertently delete a critical job configuration file, the system may no longer boot. Usage is simple:

```
$ # search all job configuration files for "unreachable" conditions
$ initctl check-config

$ # search specified job configuration file for unreachable conditions
$ initctl check-config <job>
```

Some job configuration files -- such as `plymouth.conf` -- have complex start on conditions which look for any of a number of jobs. As long as one valid set of events can be satisfied, `check-config` will be happy. However, to see if it found any missing jobs or events, specify the `--warn` option. Note that the first invocation returns no output, denoting that no problems have been found:

```
$ initctl check-config plymouth
$ initctl check-config --warn plymouth
plymouth
  start on: unknown job uxlaunch
  start on: unknown job lightdm
  start on: unknown job lxdm
  start on: unknown job xdm
  start on: unknown job kdm
$
```

Note that this is **not** an error condition since although `check-config` cannot satisfy any of these jobs, it **can** satisfy the overall configuration for `plymouth` (by the `gdm` job - see `plymouth.conf` on Ubuntu).

Note that the `check-config` command relies on the emits stanza to be correctly specified for each job configuration file that emits an event (see init(5)). See also [26].

10.1.6.3 *initctl emit*

Generates an arbitrary event.

Example:

```
# initctl emit hello-world
```

> ***Important***
>
> If you attempt to emit an event and it blocks (appears to hang), this is because there are other jobs which have a start on or stop on condition which contains this event. See Event Types for further details.

10.1.6.4 *`initctl help`*

Displays a list of `initctl` commands.

10.1.6.5 *`initctl list`*

The `list` command simply aggregates the status of all job instances. See initctl status.

10.1.6.6 *`initctl log-priority`*

To change the priority with which Upstart logs messages to the system log, you can change the log priority at any time using `log-priority` command as follows:

```
initctl log-priority <priority>
```

Where `<priority>` may be one of:

- `debug`
- `info`
- `message`
- `warn`
- `error`
- `fatal`

For example:

```
# same as "--verbose"
$ sudo initctl log-priority info

# same as "--debug"
$ sudo initctl log-priority debug
```

The default priority is `message`:

```
$ initctl log-priority
message
```

If the log-priority is changed, it can be reverted to the default like this:

```
# return to default value
$ sudo initctl log-priority message
```

Note that you will need to check the configuration for your system logging daemon (generally syslog(3) or rsyslogd(8)) to establish where it logs the output.

the output of these options is handled by your systems look at the particular daemons configuration to know where to find the output.

For a standard Ubuntu Maverick (10.10) system, the output will be sent to file `/var/log/daemon.log`, whilst on newer Ubuntu systems such as Ubuntu Natty (11.04), the output will be directed to file `/var/log/syslog`.

10.1.6.7 *`initctl notify-disk-writeable`*

Command that is used to notify Upstart that the log disk is writeable [7].

This is an indication to Upstart that it can flush the log of job output for jobs that *ended* before the log disk became writeable. If logging is enabled, this command *must* be called once the disks become writeable.

10.1.6.8 *`initctl reload`*

Causes the `SIGHUP` signal to be sent to the main job process since this signal is commonly used to inform an application to re-initialize itself. Note that the jobs associated Job Configuration File is *not* re-read.

10.1.6.9 *`initctl reload-configuration`*

Force the init daemon to reload its configuration files.

It is generally not necessary to call this command since the init daemon watches its configuration directories with inotify(7) and automatically reloads in cases of changes.

Note that no jobs will be started by this command.

10.1.6.10 *`initctl restart`*

Cause the associated job to be killed and respawned. Note that this *does not* cause the job to re-read its Job Configuration File: to force this, stop the job and then start it.

10.1.6.11 *`initctl show-config`*

The `initctl show-config` command can be used to display details of how Upstart has parsed one or more job configuration files. The command displays the start on, stop on and emits stanzas. This might seem rather pointless, but it is extremely useful since:

- The command will fully-bracket all start on and stop on conditions.

  This shows how Upstart has parsed complex conditions. For example, if job `myjob` specified a start on condition:

  ```
  start on starting a or b and stopping c or d
  ```

  The command would return:

  ```
  myjob:
    start on (((starting a or b) and stopping c) or d)
  ```

- The command can produce machine parseable output showing the types of entities by specifying the "`--enumerate`" option.

  For example, the job above would be displayed as:

  ```
  myjob
    start on starting (job: a, env:)
  ```

```
start on b (job:, env:)
start on stopping (job: c, env:)
start on d (job:, env:)
```

Thus,

- a is a job (with triggering event starting(7)).
- b is an event.
- c is a job (with triggering event stopping(7)).
- d is a event.

nd shows the environment for the events.

(ridiculous) start on condition of:

```
-a foo=bar a=b c=22 d="hello world" or stopped job-a e=123 f=blah or hello world=2a or starting foo foo=foo
```

```
l show-config --enumerate myjob

on event-a (job:, env: foo=bar a=b c=22 d=hello world)
on stopped (job: job-a, env: e=123 f=blah)
on hello (job:, env: world=2a)
on starting (job: foo, env: foo=foo)
```

nis makes the condition (slightly!) easier to understand:

-a is an event with 4 environment variables:

`oo=bar`

`=b`

`=22`

`=hello world`

is a job with triggering event stopped(7) and 2 environment variables:

`=123`

`=blah`

is an event with 1 environment variable:

`orld=2a`

a job with triggering event starting(7) and 1 environment variable:

`oo=foo`

See also [25].

10.1.6.12 *`initctl start`*

Start the specified job or job instance.

10.1.6.13 *initctl status*

The status(8) command shows the status *of all running instances* of a particular job.

The format of the output can be summarized as follows:

```
<job> [ (<instance>)]<goal>/<status>[, process <PID>]
        [<section> process <PID>]
```

Considering each field:

- `<job>` is the name of the job

  Essentially, this is the name of the job configuration file, less the path and without the "`.conf`" extension. Thus, `/etc/init/myjob.conf` would display as "`myjob`".

- `<instance>` is the job instance.

  See instance and Determining How to Stop a Job with Multiple Running Instances.

- `<goal>`

  Every job has a goal of either `start` or `stop` where the goal is the *target* the job is *aiming* for. It may not achieve this target, but the goal shows the "direction" the job is heading in: it is either trying to be started, or be stopped.

  - When a Task Job starts, its goal will be `start` and once the task in question has completed, Upstart will change its goal to `stop`.
  - When a Service Job starts, its goal will be `start` and will remain so until either the jobs stop on condition becomes true, or an Administrator manually stops the job using stop.

- `<status>`

  The job instances status. See Job States.

- `<PID>` is the process ID of the running process corresponding to `<job>`.

  See ps(1).

- `<section>` is a `script` or `exec` section (such as `pre-stop`).

Lets look at some examples...

10.1.6.13.1 Single Job Instance Running without PID

Here is the summarised syntax:

```
<job> <goal>/<status>
```

Example:

```
ufw start/running
```

You may be forgiven for thinking this rather curious specimen is an Abstract Job. Although you cannot determine the fact from the output above, this job is *not* an abstract job. If you look at its job configuration file `/etc/init/ufw.conf`, you'll see the following:

```
description     "Uncomplicated firewall"

# Make sure we start before an interface receives traffic
```

```
start on (starting network-interface
          or starting network-manager
          or starting networking)

stop on runlevel [!023456]

console output

pre-start exec /lib/ufw/ufw-init start quiet
post-stop exec /lib/ufw/ufw-init stop
```

Notice the last two lines above. The firewall job configuration file has a pre-start section and a post-stop section, but *no* `script` or `exec` section. So, once Upstart has run the pre-start command and the job is "running", it won't actually have a PID (since the pre-start command will have finished and there is no further command to run until the job stops).

10.1.6.13.2 Single Job Instance Running Job with PID

A single instance of a running job can be summarized like this:

```
<job> <goal>/<status>, process <PID>
```

This is possibly the "most common case" of jobs you will see. For example:

```
cups start/running, process 1733
```

Where:

- `<job>` is "`cups`" (`/etc/init/cups.conf`).
- `<goal>` is "`start`"
- `<status>` is "`running`"
- `<process>` is "`1733`" (as shown by ps(1)).

10.1.6.13.3 Single Job Instance Running with Multiple PIDs

This can be summarized as:

```
<job> <goal>/<status>, process <PID>
  <section> process <PID>
```

For example:

```
ureadahead stop/pre-stop, process 227
      pre-stop process 5579
```

What is going on here? Picking this apart we have:

- `ureadahead` is the job (`/etc/init/ureadahead.conf`).
- `stop` is the goal (job is trying to stop).
- `pre-stop` is the job status (it is running the pre-stop section as PID `5579`).
- the `script` or `exec` stanza is also running under PID `227`. See pre-stop for further details.

10.1.6.13.4 Multiple Running Job Instances Without PID

Summary:

```
<job> (<instance>) <goal>/<status> (<instance>)
<job> (<instance>) <goal>/<status> (<instance>)
```

A job with multiple instances might look a little strange initially. Here is an example:

```
network-interface (lo) start/running
network-interface (eth0) start/running
```

Where:

- `network-interface` is the job (`/etc/init/network-interface.conf`).
- job instances are:
  - `lo`
  - `eth0`
- `start` is the goal (job instances are currently running).
- `running` is the job status (it is running).

A slightly more complex example:

```
network-interface-security (network-manager) start/running
network-interface-security (network-interface/eth0) start/running
network-interface-security (network-interface/lo) start/running
network-interface-security (networking) start/running
```

Where:

- `network-interface-security` is the job (`/etc/init/network-interface-security.conf`).
- job instances are:
  - `network-manager`
  - `network-interface/eth0`
  - `network-interface/lo`
  - `networking`
- `start` is the goal (job instances are currently running).
- `running` is the job status (it is running).

Let's look at the main elements of the corresponding job configuration file:

```
start on (starting network-interface
          or starting network-manager
          or starting networking)

instance $JOB${INTERFACE:+/}${INTERFACE:-}

pre-start script
```

```
  # ...
end script
```

Again, this job has no `script` or `exec` section, but it does have a pre-start script section. Also, note the interesting instance stanza. This explains the rather odd-looking instance names listed above.

10.1.6.13.5 Multiple Running Job Instances With PIDs

Summary:

```
<job> (<instance>) <goal>/<status> (<instance>), process <PID>
```

For example:

```
foo (1) start/running, process 30003
foo (hello 1,2,3) start/running, process 30008
```

Where:

- `foo` is the job (`/etc/init/foo.conf`).
- `start` is the goal (it is not trying to stop).
- `running` is the job status (it is running).
- instances are:
  - `1` (PID 30003)
  - `hello 1,2,3` (PID 30008)

10.1.6.13.6 Multiple Running Job Instances With Multiple PIDs

Summary:

```
<job> (<instance>) <goal>/<status> (<instance>), process <PID>
        <section> process <PID>
```

For example:

```
myjob (foo) stop/pre-stop, process 31677
        pre-stop process 31684
myjob (bar) stop/pre-stop, process 31679
        pre-stop process 31687
myjob (bzr) stop/pre-stop, process 31681
        pre-stop process 31690
```

Where:

- `myjob` is the job (`/etc/init/myjob.conf`).
- `stop` is the goal (job is trying to stop).
- `pre-stop` is the job status (it is running the pre-stop section for each instance).
- instances are:
  - `foo` (PID 31677, with `pre-stop` PID 31684)

- `bar` (PID 31679, with `pre-stop` PID 31687)
- `baz` (PID 31681, with `pre-stop` PID 31690)

It is instructive to see how we got to the output above. Here is the job configuration file:

```
instance $foo

exec sleep 999

pre-stop script
  sleep 999
end script
```

We then started three instances like this:

```
# for i in foo bar baz; do start -n myjob foo=$i; done
```

Note we used the "`-n`" option to start to ensure we didn't have to wait for each instance to complete before starting the next.

Now all three instances are running:

```
# initctl list|grep -A 1 ^inst
myjob start/running (foo), process 31677
myjob start/running (bar), process 31679
myjob start/running (baz), process 31681
```

To trigger the pre-stop, we need to stop the instances:

```
# for i in foo bar baz; do stop -n myjob foo=$i; done
myjob (foo) stop/pre-stop, process 31677
      pre-stop process 31684
myjob (bar) stop/pre-stop, process 31679
      pre-stop process 31687
myjob (baz) stop/pre-stop, process 31681
      pre-stop process 31690
```

Now, running initctl will show the output at the start of this section.

10.1.6.13.7 Stopped Job

Summary:

```
<job> <goal>/<status>
```

A job that is not running (has no instances):

```
rc stop/waiting
```

Where:

- `rc` is the job (`/etc/init/rc.conf`).
- `stop` is the goal (it is not trying to start).

- `waiting` is the job status (it is not running).

10.1.6.14 *initctl stop*

Stop the specified job or job instance.

10.1.6.15 *initctl usage*

This command allows the usage for a job to be queried:

```
$ initctl usage <job>
```

Note that if a job is specified which does not use the usage stanza, no usage will be displayed.

10.1.6.16 *initctl version*

Display the version of the init daemon. To display the version of `initctl` itself, run:

```
initctl --version
```

10.1.7 *init-checkconf*

The `init-checkconf` script performs checks on a job configuration file *prior* to installing it in `/etc/init/`. The script must be run as a non-root user.

To ensure that you haven't misused the Upstart syntax, use the `init-checkconf` command:

```
$ init-checkconf myjob.conf
```

See init-checkconf(8) for further details.

10.1.8 *mountall* ( )

NOTE: mountall(8) is an Ubuntu-specific extension.

The `mountall` daemon is the program that mounts your filesystems during boot on an Ubuntu system. It does this by parsing both `/etc/fstab` and its own fstab file `/lib/init/fstab`, and mounting the filesystems it finds listed. Additionally, it handles running fsck(8).

See fstab(5).

10.1.8.1 *Mountall events*

Mountall also emits a number of useful events. For *every* filesystem it determines needs to be mounted, it will emit up to 2 events:

- `mounting`
- `mounted`

Additional to the couplet above, `mountall` also emits the following "well-known" events. The sections below provide details.

The `mountall` daemon is unusual in emitting such a number of events. However, it does this to provide as much flexibility as possible since making disks and filesystem available is such an important part of the boot process (and a lot of other jobs need to be notified when certain mounts become available).

10.1.8.1.1 *mounting*

Emitted when a particular filesystem is about to be mounted.

See mounting(7).

10.1.8.1.2 *mounted*

Emitted by when a particular filesystem has been mounted successfully.

Note that if a filesystem failed to mount, no corresponding `mounted` event will be emitted.

See mounted(7).

10.1.8.1.3 *all-swaps*

Emitted when all swap devices are mounted.

See all-swaps(7).

10.1.8.1.4 *filesystem*

Emitted after mountall (ubuntu-specific) has mounted (or at least attempted to mount) all filesystems.

See filesystem(7).

10.1.8.1.5 *virtual-filesystems*

Emitted after the last virtual filesystem has been mounted.

See virtual-filesystems(7).

10.1.8.1.6 *local-filesystems*

Emitted after the last local filesystem has been mounted.

See local-filesystems(7).

10.1.8.1.7 *remote-filesystems*

Emitted after the last remote filesystem has been mounted.

See remote-filesystems(7).

10.1.8.2 *Mountall Event Summary*

```
+-------------------------------------------------------------+---------------------+
|mounting MOUNTPOINT=/virtual-1                               |  mounting TYPE=swap |
|mounted  MOUNTPOINT=/virtual-1                               |  mounted  TYPE=swap |
|     :                                                       |  all-swaps          |
|mounting MOUNTPOINT=/virtual-n                               |                     |
|mounted  MOUNTPOINT=/virtual-n                               |                     |
|virtual-filesystems                                          |                     |
+-----------------------------+-------------------------------+                     |
|mounting MOUNTPOINT=/local-1 |mounting MOUNTPOINT=/remote-1 |                     |
|mounted  MOUNTPOINT=/local-1 |mounted  MOUNTPOINT=/remote-1 |                     |
|     :                       |     :                         |                     |
|mounting MOUNTPOINT=/local-n |mounting MOUNTPOINT=/remote-n |                     |
|mounted  MOUNTPOINT=/local-n |mounted  MOUNTPOINT=/remote-n |                     |
|local-filesystems            |remote-filesystems             |                     |
+-----------------------------+-------------------------------+---------------------+
|filesystem                                                                         |
+-----------------------------------------------------------------------------------+
```

The diagram above shows the different event flows when `mountall` runs. Note in particular that columns should be considered as independent "threads" of execution (can happen at any time and independently), and rows are sequential: rows lower down the chart occur at at later time than those higher up the chart.

Notes on `mountall` event emission:

- swap partitions are processed at any time.
- virtual filesystems are processed at any time.
- virtual filesystems are processed before local or remote filesystems (regardless of their ordering in `/etc/fstab`).
- local and remote filesystems are mounted at any time after the last virtual filesystem has been mounted.

See mounting(7) and mounted(7). For a concise summary of all available events generated by `mountall`, see upstart-events(7).

10.1.8.3 `mountall` Examples

The examples which follow were generated using the following job configuration file `/etc/init/get_mountall.conf`:

```
start on (local-filesystems
      or (mounting
      or (mounted
      or (virtual-filesystems
      or (remote-filesystems
      or (all-swaps or filesystem))))))

script
  echo "\n`env`" >> /dev/.initramfs/mountall.log
end script
```

Script output:

```
MOUNTPOINT=/proc
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=nodev,noexec,nosuid
TYPE=proc
UPSTART_EVENTS=mounted
PWD=/
DEVICE=proc

MOUNTPOINT=/sys/fs/fuse/connections
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=optional
TYPE=fusectl
UPSTART_EVENTS=mounted
PWD=/
```

```
DEVICE=fusectl

MOUNTPOINT=/dev/pts
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=noexec,nosuid,gid=tty,mode=0620
TYPE=devpts
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

MOUNTPOINT=/sys/kernel/debug
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=optional
TYPE=debugfs
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

MOUNTPOINT=/sys/kernel/security
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=optional
TYPE=securityfs
UPSTART_EVENTS=mounting
PWD=/
DEVICE=none

MOUNTPOINT=/sys/kernel/security
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=optional
TYPE=securityfs
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

MOUNTPOINT=/dev/shm
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=nosuid,nodev
TYPE=tmpfs
UPSTART_EVENTS=mounting
```

```
PWD=/
DEVICE=none

MOUNTPOINT=/dev/shm
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=nosuid,nodev
TYPE=tmpfs
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

MOUNTPOINT=/var/run
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=mode=0755,nosuid,showthrough
TYPE=tmpfs
UPSTART_EVENTS=mounting
PWD=/
DEVICE=none

MOUNTPOINT=/var/run
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=mode=0755,nosuid,showthrough
TYPE=tmpfs
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

MOUNTPOINT=/var/lock
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=nodev,noexec,nosuid,showthrough
TYPE=tmpfs
UPSTART_EVENTS=mounting
PWD=/
DEVICE=none

MOUNTPOINT=/var/lock
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=nodev,noexec,nosuid,showthrough
TYPE=tmpfs
```

```
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

MOUNTPOINT=/lib/init/rw
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=mode=0755,nosuid,optional
TYPE=tmpfs
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
UPSTART_EVENTS=virtual-filesystems
PWD=/

UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
UPSTART_EVENTS=remote-filesystems
PWD=/

MOUNTPOINT=none
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=sw
TYPE=swap
UPSTART_EVENTS=mounting
PWD=/
DEVICE=/dev/disk/by-uuid/b67802dc-35f9-4153-9957-ef04c7af6a1f

MOUNTPOINT=none
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=sw
TYPE=swap
UPSTART_EVENTS=mounted
PWD=/
DEVICE=/dev/disk/by-uuid/b67802dc-35f9-4153-9957-ef04c7af6a1f

UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
UPSTART_EVENTS=all-swaps
PWD=/

MOUNTPOINT=/
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=errors=remount-ro
TYPE=ext4
UPSTART_EVENTS=mounting
PWD=/
DEVICE=/dev/disk/by-uuid/b68c4bc0-6342-411c-878a-a576b3a255b3

MOUNTPOINT=/
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=errors=remount-ro
TYPE=ext4
UPSTART_EVENTS=mounted
PWD=/
DEVICE=/dev/disk/by-uuid/b68c4bc0-6342-411c-878a-a576b3a255b3

MOUNTPOINT=/tmp
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=defaults
TYPE=none
UPSTART_EVENTS=mounting
PWD=/
DEVICE=none

MOUNTPOINT=/tmp
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
OPTIONS=defaults
TYPE=none
UPSTART_EVENTS=mounted
PWD=/
DEVICE=none

UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
UPSTART_EVENTS=local-filesystems
PWD=/
```

```
UPSTART_INSTANCE=
UPSTART_JOB=get_mountall
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
UPSTART_EVENTS=filesystem
PWD=/
```

10.2 Bridges

Bridges react to events from some other (non-Upstart) source and create corresponding Upstart events.

10.2.1 *`plymouth-upstart-bridge`* ()

The `plymouth-upstart-bridge` is an Ubuntu-specific facility to allow Plymouth to display Upstart state changes on the boot splash screen.

See the Plymouth Ubuntu wiki page for more information on Plymouth.

10.2.2 *`upstart-socket-bridge`*

The Upstart socket bridge is an out-of-process application that "listens" for jobs that announce they "`start on socket`". The bridge arranges for the jobs in question to be started automatically *at the point the first client connection* is made on the socket specified in their start on condition. See socket-event(7).

This is a useful "lazy" facility in that it allows for applications which are expensive to load to be started "*on demand*" rather than simply at some point on every boot: if you have no customers to your web site one day, there is probably no point in starting your database server. The downside to using the bridge being that the first client connection will probably be slower than subsequent connections to allow the application time to start.

10.2.3 *`upstart-udev-bridge`*

The Upstart udev(7) bridge creates Upstart events from udev events. As documented in upstart-udev-bridge(8), Upstart will create events named:

```
<subsystem>-device-<action>
```

Where:

- `<subsystem>` is the udev subsystem.
- `<action>` is the udev action.

Upstart maps the three actions below to new names, but any other actions are left unmolested:

- `add` becomes `added`
- `change` becomes `changed`
- `deleted` becomes `removed`

To see a list of possible Upstart events for your system:

```
for subsystem in /sys/class/*
do
  for action in added changed removed
  do
```

```
    echo "${subsystem}-device-${action}"
  done
done
```

Alternatively, you could parse the following:

```
# udevadm info --export-db
```

To monitor udev events:

```
$ udevadm monitor --environment
```

And now for some examples...

If a job `job-A` specified a `start on` condition of:

```
start on (graphics-device-added or drm-device-added)
```

To see what sort of information is available to this job, we can add the usual debugging information:

```
start on (graphics-device-added or drm-device-added)
script
  echo "`env`" > /dev/.initramfs/job-A.log
end script
```

Here is an example of the log:

```
DEV_LOG=3
DEVNAME=/dev/fb0
UPSTART_INSTANCE=
ACTION=add
SEQNUM=1176
MAJOR=29
KERNEL=fb0
DEVPATH=/devices/platform/efifb.0/graphics/fb0
UPSTART_JOB=job-A
TERM=linux
SUBSYSTEM=graphics
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
MINOR=0
UPSTART_EVENTS=graphics-device-added
PWD=/
PRIMARY_DEVICE_FOR_DISPLAY=1
```

Another example specifying a `start on` containing `net-device-added`:

```
ID_BUS=pci
UDEV_LOG=3
UPSTART_INSTANCE=
ID_VENDOR_FROM_DATABASE=Realtek Semiconductor Co., Ltd.
ACTION=add
SEQNUM=1171
```

```
MATCHADDR=52:54:00:12:34:56
IFINDEX=2
KERNEL=eth0
DEVPATH=/devices/pci0000:00/0000:00:03.0/net/eth0
UPSTART_JOB=job-A
TERM=linux
SUBSYSTEM=net
ID_MODEL_ID=0x8139
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
ID_MM_CANDIDATE=1
ID_MODEL_FROM_DATABASE=RTL-8139/8139C/8139C+
UPSTART_EVENTS=net-device-added
INTERFACE=eth0
PWD=/
MATCHIFTYPE=1
ID_VENDOR_ID=0x10ec
```

Plugging in a USB webcam will generate an `input-device-added` event:

```
DEV_LOG=3
DEVNAME=/dev/input/event12
UPSTART_INSTANCE=
ACTION=add
SEQNUM=2689
XKBLAYOUT=gb
MAJOR=13
ID_INPUT=1
KERNEL=event12
DEVPATH=/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/input/input33/event12
UPSTART_JOB=test_camera
TERM=linux
DEVLINKS=/dev/char/13:76 /dev/input/by-path/pci-0000:00:1d.0-event
SUBSYSTEM=input
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
MINOR=76
DISPLAY=:0.0
ID_INPUT_KEY=1
ID_PATH=pci-0000:00:1d.0
UPSTART_EVENTS=input-device-added
PWD=/
```

Note: you may get additional events if it also includes a microphone or other sensors.

Plugging in a USB headset (headphones plus a microphone) will probably generate *three* events:

- `sound-device-added` (for the headphones):

  ```
  UPSTART_INSTANCE=
  ACTION=add
  SEQNUM=2637
  KERNEL=card2
  DEVPATH=/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/sound/card2
  UPSTART_JOB=test_sound
  TERM=linux
  ```

```
SUBSYSTEM=sound
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
UPSTART_EVENTS=sound-device-added
PWD=/
```

- `usb-device-added` (also for the headphones):

```
UDEV_LOG=3
DEVNAME=/dev/bus/usb/002/027
UPSTART_INSTANCE=
ACTION=add
SEQNUM=2635
BUSNUM=002
MAJOR=189
KERNEL=2-1.2
DEVPATH=/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2
UPSTART_JOB=test_usb
ID_MODEL_ENC=Logitech\x20USB\x20Headset
ID_USB_INTERFACES=:010100:010200:030000:
ID_MODEL=Logitech_USB_Headset
TERM=linux
DEVLINKS=/dev/char/189:154
ID_SERIAL=Logitech_Logitech_USB_Headset
SUBSYSTEM=usb
UPOWER_VENDOR=Logitech, Inc.
ID_MODEL_ID=0a0b
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
MINOR=154
TYPE=0/0/0
UPSTART_EVENTS=usb-device-added
ID_VENDOR_ENC=Logitech
DEVNUM=027
PRODUCT=46d/a0b/1013
PWD=/
ID_VENDOR=Logitech
DEVTYPE=usb_device
ID_VENDOR_ID=046d
ID_REVISION=1013
```

- `input-device-added` (for the microphone):

```
UDEV_LOG=3
UPSTART_INSTANCE=
ACTION=add
PHYS="usb-0000:00:1d.0-1.2/input3"
SEQNUM=2645
EV==13
KERNEL=input31
DEVPATH=/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.3/input/input31
UPSTART_JOB=test_input
MSC==10
NAME="Logitech Logitech USB Headset"
TERM=linux
```

```
SUBSYSTEM=input
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
MODALIAS=input:b0003v046Dp0A0Be0100-e0,1,4,k72,73,ram4,lsfw
KEY==c0000 0 0 0
UPSTART_EVENTS=input-device-added
PRODUCT=3/46d/a0b/100
PWD=/
```

10.2.3.1 Careful Use of udev Events

You need to be careful when using the `upstart-udev-bridge` since certain devices are *NOT* ready at the point the kernel generates the original udev event: in these circumstances, all the kernel is saying is "I have this device", not "I have this device *and it is ready to use*".

The problem is that the kernel does not know when the device is ready and neither can Upstart know this. The kernel is simply signalling that the device has either:

- become available (once the `upstart-udev-bridge` emits the "`*-device-added`" event).
- changed state somehow (once the `upstart-udev-bridge` emits the *one or more* "`*-device-changed`" events).

So, for example, just because you have received a "`usb-device-added`" event for your USB modem does not guarantee that the modem is operational.

Unfortunately, every device acts differently, so you really do need specialist knowledge of the device in question.

However, a general rule of thumb is that a device is ready once Upstart has emitted a "`changed`" event for the device which also includes a "`ID_`" variable in that events environment. This is of particular importance for "`block`" devices and "`sound`" devices.

11 Cookbook and Best Practises

11.1 List All Jobs

To list all jobs on the system along with their states, run:

```
$ initctl list
```

See initctl.

11.2 List All Jobs With No `stop on` Condition

```
# list all jobs (stopped and running instances), and compact down
# to actual job names.
initctl list | awk '{print $1}' | sort -u | while read job
do
  # identify jobs with no "stop on"
  initctl show-config -e $job | grep -q "^  stop on" || echo "$job"
done
```

11.3 List All Events That Jobs Are Interested In On Your System

Here is another example of how `initctl show-config` can be useful:

```
initctl show-config -e | egrep -i "(start|stop) on" | awk '{print $3}' | sort -u
```

11.4 Create an Event

To create, or "emit" an event, use initctl(8) specifying the emit command.

For example, to emit the hello event, you would run:

```
# initctl emit hello
```

This event will be "broadcast" to all Upstart jobs.

If you are creating a job configuration file for a new application, you probably do not need to do this though, since Upstart emits events on behalf of a job whenever the job changes state.

A simple configuration file like that shown below may suffice for your application:

```
# /etc/init/myapp.conf
description "run my app under Upstart"
task
exec /path/to/myapp
```

11.5 Create an Event Alias

Say you have an event, but want to create a different name for it, you can simulate a new name by creating a new *job* which:

- has a start on that matches the event you want to "rename"
- is a task
- emits the new name for the event

For example, if you wanted to create an alias for a particular flavour of the `runlevel` event called "`shutdown`" which would be emitted when the system was shutdown, you could create a job configuration file called `/etc/init/shutdown.conf` containing:

```
start on runlevel RUNLEVEL=0
task
exec initctl emit shutdown
```

Note that this isn't a true alias since:

- there are now *two* events which will be generated when the system is shutting down:
  - `runlevel RUNLEVEL=0`
  - `shutdown`
- the two events will be delivered by Upstart at *slightly* different times (`shutdown` will be emitted just fractionally before `runlevel RUNLEVEL=0`).

However, the overall result might suffice for your purposes such that you could create a job configuration file like the following which will run (and complete) just before your system changes to runlevel *0* (in other words halts):

```
start on shutdown
task
exec backup_my_machine.sh
```

11.5.1 Change the Type of an Event

Note that along with creating a new *name* for an event, you could make your alias be a different *type* of event. See Event Types for further details.

11.6 Synchronisation

Upstart is very careful to ensure when a condition becomes true that it **starts** all relevant jobs *in sequence* (see Order in Which Jobs Which start on the Same Event are Run). However, although Upstart has *started* them one after another *they might still be running at the same time*. For example, assume the following:

- `/etc/init/X.conf`

  ```
  start on event-A
  script
    echo "`date`: $UPSTART_JOB started" >> /tmp/test.log
    sleep 2
    echo "`date`: $UPSTART_JOB stopped" >> /tmp/test.log
  end script
  ```

- `/etc/init/Y.conf`

  ```
  start on event-A

  script
    echo "`date`: $UPSTART_JOB started" >> /tmp/test.log
    sleep 2
    echo "`date`: $UPSTART_JOB stopped" >> /tmp/test.log
  end script
  ```

- `/etc/init/Z.conf`

  ```
  start on event-A

  script
    echo "`date`: $UPSTART_JOB started" >> /tmp/test.log
    sleep 2
    echo "`date`: $UPSTART_JOB stopped" >> /tmp/test.log
  end script
  ```

Running the following will cause all the jobs above to run *in some order*:

```
# initctl emit event-A
```

Here is sample output of `/tmp/test.log`:

```
Thu Mar 31 10:20:44 BST 2011: Y started
Thu Mar 31 10:20:44 BST 2011: X started
Thu Mar 31 10:20:44 BST 2011: Z started
Thu Mar 31 10:20:46 BST 2011: Y stopped
Thu Mar 31 10:20:46 BST 2011: Z stopped
Thu Mar 31 10:20:46 BST 2011: X stopped
```

There are a few points to note about this output:

- All jobs start "around the same time" but are *started* sequentially.
- The order the jobs are initiated by Upstart cannot be predicted.
- *All three jobs are running concurrently*.

It is possible with a bit of thought to create a simple framework for synchronisation. Take the following job configuration file `/etc/init/synchronise.conf`:

```
manual
```

This one-line Abstract Job configuration file is extremely interesting in that:

- Since it includes the manual keyword, a job created from it can only be started manually.
- Only a single instance of a job created from this configuration can exist (since no instance stanza has been specified).

What this means is that we can use a job based on this configuration as a simple synchronisation device.

The astute reader may observe that `synchronise` has similar semantics to a POSIX pthread condition variable.

Now we have our synchronisation primitive, how do we use it? Here is an example which we'll call `/etc/init/test_synchronise.conf`:

```
start on stopped synchronise

# allow multiple instances
instance $N

# this is not a service
task

pre-start script
  # "lock"
  start synchronise || true
end script

script
  # do something here, knowing that you have exclusive access
  # to some resource that you are using the "synchronise"
  # job to protect.
  echo "`date`: $UPSTART_JOB ($N) started" >> /tmp/test.log
  sleep 2
  echo "`date`: $UPSTART_JOB ($N) stopped" >> /tmp/test.log
end script
```

```
post-stop script
  # "unlock"
  stop synchronise || true
end script
```

For example, to run 3 instances of this job, run:

```
for n in $(seq 3)
do
  start test_synchronise N=$n
done
```

Here is sample output of `/tmp/test.log`:

```
Thu Mar 31 10:32:20 BST 2011: test_synchronise (1) started
Thu Mar 31 10:32:22 BST 2011: test_synchronise (1) stopped
Thu Mar 31 10:32:22 BST 2011: test_synchronise (2) started
Thu Mar 31 10:32:24 BST 2011: test_synchronise (2) stopped
Thu Mar 31 10:32:25 BST 2011: test_synchronise (3) started
Thu Mar 31 10:32:27 BST 2011: test_synchronise (3) stopped
```

The main observation here:

- Each instance of the job *started* **and stopped** before any other instance ran.

Like condition variables, this technique require collaboration from all parties. Note that you cannot know the order in which each instance of the `test_synchronise` job will run.

Note too that it is not necessary to use instances here. All that is required is that your chosen set of jobs all collaborate in their handling of the "lock". Instances make this simple since you can spawn any number of jobs from a single "template" job configuration file.

11.7 Determine if Job was Started by an Event or by "`start`"

A job that specifies a start on condition can be started in two ways:

- by Upstart itself when the start on condition becomes true.
- by running, "`start <job>`".

Interestingly, it is possible for a job to establish how it was started by considering the `UPSTART_EVENTS` variable:

- If the `UPSTART_EVENTS` variable is set in the job environment, the job was started by an event.
- If the `UPSTART_EVENTS` variable is *not* set in the job environment, the job was started by the `start` command.

Note that this technique does not allow you to determine definitively if the job was started *manually* by an Administrator since it is possible that if the `UPSTART_EVENTS` variable is *not* set that the job was started by *another job* calling `start` inside a `script` section.

11.8 Stop a Job from Running if A `pre-start` Condition Fails

If you wish a job to not be run if a `pre-start` condition fails:

```
pre-start script
  # main process will not be run if /some/file does not exist
  test -f /some/file || { stop ; exit 0; }
end script

script
  # main process is run here
end script
```

11.9 Run a Job Only When an Event Variable Matches Some Value

By default, Upstart will run your job if the start on condition matches the events listed:

```
start on event-A
```

But if `event-A` provides a number of environment variables, you can restrict your job to starting *only* when one or more of these variables matches some value. For example:

```
start on event-A FOO=hello BAR=wibble
```

Now, Upstart will only run your job *if all of the following are true*:

- the `event-A` is emitted
- the value of the `$FOO` variable in `event-A`'s environment is "`hello`".
- the value of the `$BAR` variable in `event-A`'s environment is "`wibble`".

11.10 Run a Job when an Event Variable Does Not Match Some Value

Upstart supports negation of environment variable values such that you can say:

```
start on event-A FOO=hello BAR!=wibble
```

Now, Upstart will only run your job *if all of the following are true*:

- the `event-A` is emitted
- the value of the `$FOO` variable in `event-A`'s environment is "`hello`".
- the value of the `$BAR` variable in `event-A`'s environment is **not** "`wibble`".

11.11 Run a Job as Soon as Possible After Boot

(Note: we ignore the initramfs in this section).

To start a job as early as possible, simply "`start on`" the `startup` event. This is the first event Upstart emits and all other events and jobs follow from this:

```
start on startup
```

11.12 Run a Job When a User Logs in Graphically ( )

Assuming a graphical login, this can be achieved using a `start on` condition of:

```
start on desktop-session-start
```

This requires the display manager emit the event in question. See the upstart-events(7) man page on an Ubuntu system for the 2 events a Display Manager is expected to emit. If your Display Manager does not emit these event, check its documentation to see if it allows scripts to be called at appropriate points and then you can easily conform to the reference implementations behaviour:

```
# A user has logged in
/sbin/initctl -q emit desktop-session-start \
  DISPLAY_MANAGER=some_name USER=$USER

# Display Manager has initialized and displayed a login screen
# (if appropriate)
/sbin/initctl -q emit login-session-start \
  DISPLAY_MANAGER=some_name
```

11.13 Run a Job When a User Logs in

This makes use of D-Bus Service Activation.

1. Add "`UpstartJob=true`" to file "`/usr/share/dbus-1/system-services/org.freedesktop.ConsoleKit.service`".
2. Create a job configuration file corresponding to the D-Bus service, say `/etc/init/user-login.conf` [12]:

   ```
   start on dbus-activation org.freedesktop.ConsoleKit
   exec /usr/sbin/console-kit-daemon --no-daemon
   ```

3. Ensure that the D-Bus daemon ("`dbus-daemon`") is started with the `--activation=upstart` option (see `/etc/init/dbus.conf`).

Now, when a user logs in, D-Bus will emit the `dbus-activation` event, specifying the D-Bus service started. You can now create other jobs that `start on user-login`.

11.13.1 Environment

Below is an example of the environment such an Upstart D-Bus job runs in:

```
UPSTART_INSTANCE=
DBUS_STARTER_BUS_TYPE=system
UPSTART_JOB=user-login
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
SERVICE=org.freedesktop.ConsoleKit
DBUS_SYSTEM_BUS_ADDRESS=unix:path=/var/run/dbus/system_bus_socket,guid=e86f5a01fbb7f5f1c22131090000000a
UPSTART_EVENTS=dbus-activation
PWD=/
DBUS_STARTER_ADDRESS=unix:path=/var/run/dbus/system_bus_socket,guid=e86f5a01fbb7f5f1c22131090000000a
```

11.14 Run a Job For All of a Number of Conditions

If you have a job configuration file like this:

```
start on (event-A or (event-B or event-C))

script
  echo "`date`: ran in environment: `env`" >> /tmp/myjob.log
end script
```

Upstart will run this job *when any of the following events is emitted*:

- `event-A`
- `event-B`
- `event-C`

You cannot know the order in which the events will arrive in, but the specified start on condition has told Upstart that any of them will suffice for your purposes. So, if `event-B` is emitted first, Upstart will run the job and only consider re-running the job if and when the job has finished running. If `event-B` is emitted and the job is running and *then* (before the job finishes running) `event-A` is emitted, *the job will not be re-run*.

However, what if you wanted to run the script for all the events? If you know that all of these events will be emitted at some point, you could change the start on to be:

```
start on (event-A and (event-B and event-C))
```

Here, the job will only run at the time when the last of the three events is received.

Is it possible to run this job for each event *as soon as each event arrives*? Yes it is:

```
start on (event-A or (event-B or event-C))

instance $UPSTART_EVENTS

script
  echo "`date`: ran in environment: `env`" >> /tmp/myjob.log
end script
```

By adding the instance keyword, you ensure that whenever *any* of the events listed in your start on condition is emitted, *an instance of* the job will be run. Therefore, if all three events are emitted very close together in time, three jobs *instances* will now be run.

See the Instance section for further details.

11.15 Run a Job Before Another Job

If you wish to run a particular job before some other job, simply make your jobs `start on` condition specify the starting(7) event. Since the starting(7) event is emitted *just before* the job in question starts, this provides the behaviour you want since your job will be run first.

For example, assuming your job is called `job-B` and you want it to start before `job-A`, in `/etc/init/job-B.conf` you would specify:

```
start on starting job-A
```

11.16 Run a Job After Another Job

If you have a job you wish to run after job "`job-A`", your `start on` condition would need to make use of the stopped(7) event like this:

```
start on stopped job-A
```

11.17 Run a Job Once After Some Other Job Ends

Imagine a job configuration file `myjob.conf` such as the following which might result in a job which is restarted a number of times:

```
start on event-A

script
  # do something
end script
```

Is it possible to run a job *only once* **after** job `myjob` ends? Yes if you create a job configuration file `myjob-sync.conf` such as:

```
start on stopped myjob and event-B

script
  # do something
end script
```

Now, when `event-A` is emitted, job `myjob` will start and if and when job `myjob` finishes *and* event `event-B` is emitted, job `myjob-sync` will be run.

However, crucially, even if job `myjob` is restarted, the `myjob-sync` job will *not* be restarted.

11.18 Run a Job Before Another Job and Stop it After that Job Stops

If you have a job you wish to be running before job "`job-A`" starts, but which you want to stop as soon as `job-A` stops:

```
start on starting job-A
stop on stopped job-A
```

11.19 Run a Job Only If Another Job Succeeds

To have a job start only when `job-A` succeeds, use the `$RESULT` variable from the stopped(7) event like this:

```
start on stopped job-A RESULT=ok
```

11.20 Run a Job Only If Another Job Fails

To have a job start only when `job-A` fails, use the `$RESULT` variable from the stopped(7) event like this:

```
start on stopped job-A RESULT=failed
```

Note that you could also specify this condition as:

```
start on stopped job-A RESULT!=ok
```

11.21 Run a Job Only If One Job Succeeds and Another Fails

This would be a strange scenario to want, but it is quite easy to specify. Assuming we want a job to start only if `job-A` succeeds and if `job-B` fails:

```
start on stopped job-A RESULT=ok and stopped job-B RESULT=failed
```

11.22 Run a Job If Another Job Exits with a particular Exit Code

Imagine you have a database server process that exits with a particular exit code (say 7) to denote that it needs some sort of cleanup process to be run before it can be re-started. To handle this you could create `/etc/init/mydb-cleanup.conf` with a `start on` condition like this:

```
start on stopped mydb EXIT_STATUS=7

script
  # handle cleanup...

  # assuming the cleanup was successful, restart the server
  start mydb
end script
```

11.23 Detect if Any Job Fails

To "monitor" all jobs for failures, you could either create a job that checks specifically for a *single* job failure (see Run a Job If Another Job Exits with a particular Exit Code), but you could just as easily detect if *any* job has failed as follows:

```
start on stopped RESULT=failed
```

Since this start on condition does not specify the Job to match against, it will match all jobs. You can then perform condition processing:

```
script
    if [ -n "$EXIT_STATUS" ];
    then
        str="with exit status $EXIT_STATUS"
    else
        str="due to signal $EXIT_SIGNAL"
    fi

    logger "Upstart Job $JOB (instance '$INSTANCE', process $PROCESS) failed $str"

    case "$JOB" in
        myjob1)
```

```
        ;;

        myjob2)
        ;;

        etc)
        ;;
    esac

end script
```

Note that `$PROCESS` above is *not* the PID, it is the name of the job process type (such as `main` or `pre-start`). See stopped(7) for further details.

11.24 Use Details of a Failed Job from Another Job

Although you cannot see the exact environment another job ran in, you can access some details. For example, if your job specified `/etc/init/job-B.conf` as:

```
start on stopped job-A RESULT=fail

script
  exec 1>>/tmp/log.file
  echo "Environment of job $JOB was:"
  env
  echo
end script
```

The file `/tmp/log.file` might contain something like this:

```
UPSTART_INSTANCE=
EXIT_STATUS=7
INSTANCE=
UPSTART_JOB=B
TERM=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin
PROCESS=main
UPSTART_EVENTS=stopped
PWD=/
RESULT=failed
JOB=A
```

Here, `job-B` can see that:

- `job-A` exited in its "main" process. This is a special name for the `script` section. All other script sections are named as expected. For example, if the `pre-start` section had failed, the `PROCESS` variable would be set to `pre-start`, and if in `post-stop`, the variable would have been set to `post-stop`.
- `job-A` exited with exit code `7`.
- `job-A` only had 1 instance (since the `INSTANCE` variable is set to the null value.
- `job-A` ran in the root ("`/`") directory.
- `UPSTART_JOB` is the name of the job running the script (ie `job-B`).

- `JOB` is the name of the job that we are starting on (here `job-A`).
- `UPSTART_EVENTS` is a list of the events that caused `UPSTART_JOB` (ie `job-B`) to start. Here, the event is starting(7) showing that `job-B` started as a result of `job-A` being sent the stopped(7) event.

11.25 Stop a Job when Another Job Starts

If we wish `job-A` to stop when `job-B` starts, specify the following in `/etc/init/job-A.conf`:

```
stop on starting job-B
```

11.25.1 Simple Mutual Exclusion

It is possible to create two jobs which will be "toggled" such that when `job-A` is running, `job-B` will be stopped and *vice versa*. This provides a simple mutually exclusive environment. Here is the job configuration file for `job-A`:

```
# /etc/init/job-A.conf
start on stopped job-B

script
  # do something when job-B is stopped
end script
```

And `job-B`:

```
# /etc/init/job-B.conf
start on stopped job-A

script
  # do something when job-A is stopped
end script
```

Finally, start one of the jobs:

```
# start job-A
```

Now:

- when `job-A` is running, `job-B` will be stopped.
- when `job-B` is running, `job-A` will be stopped.

Note though that attempting to have more than two jobs using such a scheme *will not work*. However, you can use the technique described in the Synchronisation section to achieve the same goal.

11.26 Run a Job Periodically

This cannot currently be handled by Upstart directly. However, the "Temporal Events" feature is being worked on now will address this.

Until Temporal Events are available you should either use cron(8), or something like:

```
# /etc/init/timer.conf

instance $JOB_TO_RUN

script
  for var in SLEEP JOB_TO_RUN
  do
    eval val=\${$var}
    if [ -z "$val" ]
    then
      logger -t $0 "ERROR: variable $var not specified"
      exit 1
    fi
  done

  eval _sleep=\${SLEEP}
  eval _job=\${JOB_TO_RUN}

  while [ 1 ]
  do
    stop  $_job || true
    sleep $_sleep
    start $_job || true
  done
end script
```

Note well the contents of the `while` loop. We ensure that the commands that might fail are converted into expressions guaranteed to pass. If we did not do this, `timer.conf` would fail, which would be undesirable. Note too the use of `instance` to allow more than one instance of the `timer` job to be running at any one time.

11.27 Restart a job on a Particular Event

To restart a job when a particular event is emitted requires two jobs. First the main job:

```
start on something

exec /sbin/some-command
```

Then a helper job to perform the restart:

```
start on my-special-event

exec restart main-job
```

Now, when the `my-special-event` event is emitted, the main job will be restarted.

11.28 Migration from System V initialization scripts

With SysV init scripts, the Administrator decides the order that jobs are started in by assigning numeric values to each service. Such a system is simple, but non-optimal since:

- The SysV init system runs each job sequentially.

This disallows running jobs in parallel, to make full use of system resources. Due to the limited nature of the SysV system, many SysV services put services that take a long time to start into the background to give the illusion that the boot is progressing quickly. However, this makes it difficult for Administrators to know if a required service is running by the time their later service starts.

- The Administrator cannot know the best order to run jobs in.

  Since the only meta information encoded for services is a numeric value used purely for ordering jobs, the system cannot optimize the services since it knows nothing about the requirements for each job.

In summary, the SysV init system is designed to be easy for the Administrator to use, not easy for the system to optimize.

In order to migrate a service from SysV to Upstart, it is necessary to change your mindset somewhat. Rather than trying to decide which two services to "slot" your service between, you need to consider the conditions that your service needs before it can legitimately be started.

So, if you wished to add a new service that traditionally started before cron(8) or atd(8) you do not need to change the configuration files `cron.conf` or `atd.conf`. You can "insert" your new service by specifying a simple:

```
# /etc/init/my-service.conf
start on (starting cron or starting atd)
```

In English, this says,

> "start the "`my-service`" service *just before* **either** the `cron` or the `atd` services start".

Whether `crond` or `atd` actually start first is not a concern for my-service: Upstart ensures that the `my-service` service will be started before either of them. Even if `cron` normally starts before `atd` but for some reason one day atd starts first, Upstart will ensure that `my-service` will be started before `atd`.

Note therefore that introducing a new service should not generally require existing job configuration files to be updated.

11.29 How to Establish a Jobs `start on` and `stop on` Conditions

How do you establish what values you should specify for a jobs `start on` and `stop on` conditions?

11.29.1 Determining the `start on` Condition ( )

So you have created a Job Configuration File for your Service Job. You have checked the expect stanza is correct and you've even enabled respawn.

But how do you determine the *correct* "*start on*" condition? Actually, this is almost a trick question since there are potentially many "correct" answers; it depends on the application and how sensitive it is to the environment it runs in. There are *many* potential `start on` conditions - it is your job to determine the most efficient and effective one. This section attempts to give some advice and guidelines on chosing a suitable condition, and explaining how to test your choice for correctness. However, note that each job requires a specific and possibly unique set of conditions to run.

11.29.1.1 Standard Idioms

If your application isn't particularly needy, you may be able to use one of the standard idioms below:

- To start your job *as soon as possible*:

  See Run a Job as Soon as Possible After Boot.

- To start your job "*as late as possible*":

  See Run a Job When a User Logs in Graphically (ubuntu-specific).

- If you want the job to start "around the time" (actually just after) the equivalent System-V job would run, specify:

```
start on stopped rc
```

- If you want your job to start after all filesystems are mounted, specify:

```
start on filesystem
```

- If you want your job to start when all network devices are active, specify:

```
start on stopped networking
```

  Note that as of Ubuntu Oneiric, you could also say:

```
start on static-network-up
```

- If you want your job to start when a runlevel begins, specify:

```
start on runlevel [2345]
```

  This is used by a lot of standard jobs and is a good starting place.

11.29.1.2 More Exotic start on Conditions

If your job more precise control over when your job starts, read carefully the upstart-events(7) manual page which summarizes all the "well-known" events you can rely upon on an Ubuntu system. These events provide a set of "hook points" which your job can make use of to simplify the job of specifying the start on condition.

The main question to ask yourself is, "*what are the exact requirements for the job?*". To help answer that question consider the following questions:

- Does your application live in a standard local directory?
- Does the application write any files to disk? (data files, log files, lock files, named sockets?) If so, which partition(s) does it need to write to?
- Does the application read any files from disk? If so, which partitions do they live in? `/etc`? `/var`?
- Do you want the application to start as early as possible, or as late as possible?
- Does the application need to start before or after a service which might *not* be installed?
- If the application needs access to a disk (it probably will), which partitions or mounts does it need? `/etc`? `/var`? `/mnt/remote-system`? Can it wait until all *local* partitions are mounted? Or does it need to wait for a particular *remote* filesystem to be mounted?
- Should a particular set of services already be running when your job starts?
- Should a particular set of services *not* be running when your job starts?
- What runlevel (or runlevels) should your job run in?

- Does your application require a network?
    - Does it need a local network (127.0.0.1?)
    - Does it need IPv6?
    - Does it require a bridge network interface?
- Should your service only start when a client network connection is initiated? If so, use the `socket` event (emitted by the upstart-socket-bridge). See the socket-event(7) man page for details.
- Does your job require the services of some other system server?
- Does your job access files over the network?
- Does your application provide a D-Bus service which you want to start when some sequence of Upstart events are emitted?

  If so, use the D-Bus service activation facility.

This list can be summarized as:

> What are the precise conditions your job needs before it can be started successfully?

And yes, you really do need to be able to answer all the questions above before you can know that you have chosen the correct start on condition. This might sound daunting, but consider:

- Upstart needs to know this information to allow your application to run at the correct point.
- By devoting some time to understanding your applications requirements, you will allow the system to run as efficiently as possible.

11.29.1.2.1 udev conditions

To identify a start on condition making use of udev events, first you need to know which udev subsystem is appropriate. See upstart-udev-bridge for details.

Having identified the subsystem, follow the steps below:

1. Create a job that displays all udev variables set for a particular udev subsystem.

   In the example below, we're consider at the `tty` subsystem, so modify to taste:

   ```
   start on tty-device-added
   exec env
   ```

2. Boot your system and look at the relevant log file for the job.

   For example look at `/var/log/upstart/myjob.log` to see which udev variables are set for your chosen udev subsystem.

   If your version of Upstart does not have job logging, you'll need to redirect the output of `env` somewhere - refer to section See the Environment a Job Runs In.

3. Refine your start on condition accordingly.

   For example, you might change it to be something like:

   ```
   start on tty-device-added DEVNAME=*ttyS1
   ```

   to start the job when the `/dev/ttS1` serial device becomes available.

11.29.2 *Determining the `stop on` Condition ( )*

Recall from the Shutdown section that if no `stop on` condition is stopped, your job will be killed *at some (random) point* at system shutdown. If you need your job to stop at a pariticular point in the shutdown sequence, you *must* specify a suitable `stop on` condition.

Shut down is not as event rich as startup. A common idiom is to specify your `stop on` as:

```
stop on runlevel [016]
```

This ensures the job will be stopped on shutdown, when switching to single-user mode and on reboot.

The next most common is to stop your job either before or after some other job stops:

- To stop a job just before a particular job has *started* to stop:

  ```
  # stop your job "just before" job 'some-job' ends
  stop on stopping some-job
  ```

  See also Run a Job Before Another Job.

- To stop a job immediately after a particular job has stopped:

  ```
  # stop your job "just after" job 'some-job' has ended
  stop on stopped some-job
  ```

  See also Run a Job After Another Job.

Other questions relating to other stanzas:

- What should happen if your job fails to start?
- What should happen if your job fails after some period of time?
- Do you want Upstart to restart the job if it exits? If so, use the `respawn` stanza.
- Does your job use non-standard exit codes to denote success and failure? If so, use the `normal exit` stanza.
- Is your job a daemon? If so, how many times does it call fork(2)?

11.29.3 *Final Words of Advice*

If your start on or stop on conditions are becoming complex (referencing more than 2 or maybe 3 events), you should consider your strategy carefully since there is probably an easier way to achieve your goal by specifying some more appropriate event. See the upstart-events(7) manual page for ideas.

Also, review the conditions from standard job configuration files on your system. However, it is inadvisable to make use of conditions you do not *fully* understand.

11.30 Guarantee that a job will only run once

If you have a job which must only be run once, but which depends on multiple conditions, the naive approach won't necessarily work:

```
task
start on (A or B)
```

If event 'A' is emitted, the task will run. But assuming the task has completed and event 'B' is *then* emitted, the task will run *again*.

11.30.1 Method 1

A better approach is as follows:

1. Create separate job configuration files for each condition you want your job to start on:

```
# /etc/init/got-A.conf
# job that will "run forever" when event A is emitted
start on A

# /etc/init/got-B.conf
# job that will "run forever" when event B is emitted
start on B
```

2. Create a job which starts on either of the `got-A` or `got-B` jobs starting:

```
# /etc/init/only-run-once.conf
start on (starting got-A or starting got-B)
```

Now, job "`only-run-once`" will start only once since jobs "`got-A`" and "`got-B`" can only be started once themselves since:

- they do not specify the instance stanza to allow multiple instances of the jobs.
- if either job starts, that job will run forever.
- none of the jobs have a stop on stanza.

11.30.2 Method 2

Change your `start on` condition to include the `startup` event:

```
task
start on startup and (A or B)
```

11.31 Stop a Job That is About to Start

Upstart will start a job when its "`start on`" condition becomes true.

Although somewhat unusual, it is quite possible to stop a job from starting when Upstart tries to start it:

```
start on starting job-A

script
  stop $JOB
end script
```

11.32 Stop a Job That is About to Start From Within That Job

You can in fact stop a job that Upstart has decided it needs to start *from within that job*:

```
pre-start script
  stop
end script
```

This is actually just an alias for:

```
pre-start script
  stop $UPSTART_JOB
end script
```

Of course, you could set the pre-start using the Override Files facility.

11.33 Stop a Job from Running if its Configuration file has not been Created/Modified

Use a pre-start stanza to check for required application conditions. If these are not met, call:

```
stop
exit 0
```

This will cause the job to stop successfully before the main script or exec stanza (which would run your application/daemon) is started.

In particular, see the Ubuntu-specific example

11.34 Stop a Job When Some Other Job is about to Start

Here, we create `/etc/init/job-C.conf` which will stop `job-B` when `job-A` is *about to start*:

```
start on starting job-A

script
  stop job-B
end script
```

11.35 Start a Job when a Particular Filesystem is About to be Mounted

Here, we start a job when the `/apps` mountpoint is mounted read-only as an NFS-v4 filesystem:

```
start on mounting TYPE=nfs4 MOUNTPOINT=/apps OPTION=ro
```

Here's another example:

```
start on mounted MOUNTPOINT=/var/run TYPE=tmpfs
```

Another example where a job would be started when any non-virtual filesystem is mounted:

```
start on mounted DEVICE=[/UL]*
```

The use of the `$DEVICE` variable is interesting. It is used here to specify succinctly any device that:

- is a real device (starts with "/" (to denote a normal "/dev/..." mount)).
- is a device specified by its filesystem:
    - label (starts with "L" (to denote a "LABEL=" mount)).
    - UUID (starts with "U" (to denote a "UUID=" mount)).

Another example where a job is started when a non-root filesystem is mounted:

```
start on mounting MOUNTPOINT!=/ TYPE!=swap
```

11.36 Start a Job when a Device is Hot-Plugged

Hot-plug kernel events create udev(7) events under Linux and Upstart events are created from udev events by the upstart-udev-bridge(8).

Added to this the `ifup` and `ifdown` commands are run at boot when network devices are available for use.

11.36.1 To start a job when eth0 is added to the system

Note that the device is *not* yet be available for use):

```
start on net-device-added INTERFACE=eth0
```

See upstart-udev-bridge for more examples.

On an Ubuntu system, you can see which devices have been added by udev (which the `upstart-udev-bridge` is using) with this snippet:

```
$ awk 'BEGIN {RS=""; ORS="\n\n"}; /ACTION=add/ && /SUBSYSTEM=net/ { print; }' \
    /var/log/udev | grep ^INTERFACE= | cut -d= -f2 | sort -u
eth0
lo
wlan0
$
```

11.36.2 To start a job when eth0 is available

Here, the device is available for use:

```
start on net-device-up IFACE=eth0
```

Notes:

- It does not matter whether the `eth0` interface has been configured statically, or if it is handled via DHCP, this event will always be emitted.

  See upstart-events(7) and file `/var/log/udev` for further details.
- The "`net-device-up`" event sets the "`IFACE`" variable whereas the net-device-added event sets the "`INTERFACE`" variable!

11.37 Stopping a Job if it Runs for Too Long

To stop a running job after a certain period of time, create a generic job configuration file like this:

```
# /etc/init/timeout.conf
stop on stopping JOB=$JOB_TO_WAIT_FOR
kill timeout 1
manual

export JOB_TO_WAIT_FOR
export TIMEOUT

script
  sleep $TIMEOUT
  initctl stop $JOB_TO_WAIT_FOR
end script
```

Now, you can control a job using a timeout:

```
start myjob
start timeout JOB_TO_WAIT_FOR=myjob TIMEOUT=5
```

This will start job `myjob` running and then wait for 5 seconds. If job "`myjob`" is still running after this period of time, the job will be stopped using the initctl(8) command. Note the `stop on` stanza which will cause the `timeout` job not to run if the job being waited for has already started to stop.

11.38 Run a Job When a File or Directory is Created/Deleted

If you need to start a Job only when a certain file is created, you could create a generic job configuration file such as the following:

```
# /etc/init/wait_for_file.conf
instance FILE_PATH
export   FILE_PATH

script
  while [ ! -e "$FILE_PATH" ]
  do
    sleep 1
  done

  initctl emit file FILE_PATH="$FILE_PATH"
end script
```

Having done this, you can now make use of it. To have another job start if say file `/var/run/foo.dat` gets created, you first need to create a job configuration file stating this:

```
# /etc/init/myapp.conf
start on file FILE_PATH=/var/run/foo.dat

script
  # ...
end script
```

Lastly, kick of the process by starting an instance of `wait_for_file`:

```
start wait_for_file FILE_PATH=/var/run/foo.dat
```

Now, when file `/var/run/foo.dat` is created, the following will happen:

1. The `myapp` job will emit the `file` event, passing the path of the file which you just specified in that events environment.
2. Upstart will see that the start on condition for the `myapp` job configuration file is satisfied.
3. Upstart will create a `myapp` job, and start it.

You can modify this strategy slightly to run a job when a file is:

- modified
- deleted
- contains certain content
- *et cetera*

See test(1), or your shells documentation for available file tests.

Note that this is very simplistic. A better approach would be to use inotify(7).

11.39 Run a Job Each Time a Condition is True

This is the default way Upstart works when you have defined a task:

```
# /etc/init/myjob.conf
task
exec /some/program
start on (A or B)
```

Job "myjob" will run every time either event 'A' or event 'B' are emitted. However, there is a corner condition: if event 'A' has been emitted and the task is *currently running* when event 'B' is emitted, job "myjob" will *not* be run. To avoid this situation, use instances:

```
# /etc/init/myjob2.conf
task
instance $SOME_VARIABLE
exec /some/program
start on (A or B)
```

Now, as long variable `$SOME_VARIABLE` is defined with a unique value each time either event 'A' or 'B' is emitted, Upstart will run job "`myjob2`" multiple times.

11.40 Run a Job When a Particular Runlevel is Entered and Left

To run a job when a particular runlevel is entered and also run it when that same runlevel is left, you could specify:

```
start on runlevel RUNLEVEL=5 or runlevel PREVLEVEL=5
```

See runlevel(7) and the Runlevels section for more details.

11.41 Pass State From a Script Section to its Job Configuration File

Assume you have a job configuration file like this:

```
script
   # ...
end script

exec /bin/some-program $ARG
```

How can you get the script section to set `$ARG` and have the job configuration file use that value in the "`exec`" stanza? This isn't as easy as you might imagine for the simple reason that Upstart runs the `script` section in a new process. As such, by the time Upstart gets to the `exec` stanza the process spawned to handle the `script` section has now ended. This implies they cannot communicate directly.

A way to achieve the required goal is as follows:

```
# set a variable which is the name of a file this job will use
# to pass information between script sections.
env ARG_FILE="/var/myapp/myapp.dat"

# make the variable accessible to all script sections (ie sub-shells)
export ARG

pre-start script
   # decide upon arguments and write them to
   # $ARG_FILE, which is available in this sub-shell.
end script

script
  # read back the contents of the arguments file
  # and pass the values to the program to run.
  ARGS="$(cat $ARG_FILE)"
  exec /bin/some-program $ARGS
end script
```

11.42 Pass State From Job Configuration File to a Script Section

To pass a value from a job configuration file to one of its script sections, simply use the `env` stanza:

```
env CONF_FILE=/etc/myapp/myapp.cfg

script
  exec /bin/myapp -c $CONF_FILE
end script
```

This example is a little pointless, but the following slightly modified example is much more useful:

```
start on an-event
export CONF_FILE
```

```
script
  exec /bin/myapp -c $CONF_FILE
end script
```

By dropping the use of the `env` stanza we can now pass the value in via an event:

```
# initctl emit an-event CONF_FILE=/etc/myapp/myapp.cfg
```

This is potentially much more useful since the value passed into `myapp.conf` can be varied without having to modify the job configuration file.

11.43 Run a Job as a Different User

11.43.1 Running a User Job

See User Job.

11.43.2 Changing User

Some daemons start running as the super-user and then internally arrange to drop their privilege level to some other (less privileged) user. However, some daemons do not need to do this: they never need root privileges so can be invoked as a non-root user.

How do you run a "system job" but have it run as a non-root user then? As of Upstart 1.4, Upstart has the ability to run a System Job as a specified user using the setuid and setgid stanzas.

However, if you are not using Upstart 1.4, it is easy to accomplish the required goal. There are a couple of methods you can use. The recommended method for Debian and Ubuntu systems is to use the helper utility start-stop-daemon(8) like this:

```
exec start-stop-daemon --start -c myuser --exec command
```

The advantage of using start-stop-daemon(8) is that it simply changes the user and group the command is run as. This also has an advantage over su(1) in that su(1) must fork to be able to hold its PAM session open, and so is harder for upstart to track, whereas start-stop-daemon(8) will simply exec the given command after changing the uid/gid.

Another potential issue to be aware of is that `start-stop-daemon` does *not* impose PAM ("Pluggable Authentication Module") limits to the process it starts. Such limits can be set using the appropriate Upstart stanzas, you just cannot specify the limits via PAMs limits.conf(5).

Of course, you may *want* PAM restrictions in place, in which case you should either use su(1) or sudo(8), both of which are linked to the PAM libraries.

The general advice is *NOT* to use su(1) or sudo(8) though since PAM restrictions really not appropriate for system services. For example, PAM will make a wtmp(5) entry every time su(1) or sudo(8) are called and those records are not appropriate for system services.

If you want to use su(1) or sudo(8), the examples below show you how.

Using su(1):

```
exec su -s /bin/sh -c command $user
```

Note that although you *could* simplify the above to the following, it is not recommended since if user "`$user`" is a system account with a shell specified as `/bin/false`, the job will *not* run the specified command: it will fail due to `/bin/false` returning "1":

```
exec su -c command $user
```

The job will silently fail if user "`$user`" is a system account with a shell specified as `/bin/false`.

To avoid the fork(2) caused by the shell being spawned, you could instead specify:

```
exec su -s /bin/sh -c 'exec "$0" "$@"' $user -- /path/to/command --arg1=foo -b wibble
```

This technique is particularly useful if your job is a Service Job that makes use of expect.

A basic example using sudo(8):

```
exec sudo -u $user command
```

11.44 Disabling a Job from Automatically Starting

With Upstart 0.6.7, to stop Upstart automatically starting a job, you can either:

- Rename the job configuration file such that it does not end with "`.conf`".
- Edit the job configuration file and comment out the "`start on`" stanza using a leading '#'.

To re-enable the job, just undo the change.

11.44.1 Override Files

With Upstart 1.3, you can make use of override files and the `manual` stanza to achieve the same result in a simpler manner [27]:

```
# echo "manual" >> /etc/init/myjob.override
```

Note that you *could* achieve the same effect by doing this:

```
# echo "manual" >> /etc/init/myjob.conf
```

However, using the override facility means you can leave the original job configuration file untouched.

To revert to the original behaviour, either delete or rename the override file (or remove the `manual` stanza from your "`.conf`" file).

11.45 Jobs that "Run Forever"

To create a job that runs continuously from the time it is manually started(7) until the time it is manually stopped(7), create a job configuration file without any process definition (`exec` and `script`) or event definition (`start on` for example) stanzas:

```
# /etc/init/runforever.conf
description "job that runs until stopped manually"
```

This job can only be started by the administrator running:

```
# start runforever
```

The status of this job will now be "`start/running`" until the administrator subsequently runs:

```
# stop runforever
```

These "Abstract Job" types have other uses as covered in other parts of this document. See for example Synchronisation.

11.46 Run a Java Application

Running a Java application is no different to any other, but Java suffers from the inability to switch users without extra helper classes.

If your Java daemon needs to run as a different user and you are running Upstart 1.4, you can use the setuid and setgid stanzas.

However, if you are using an older version, you will have to use a facility such as su(1). Also, you may wish to define some variables to simplify the invocation:

```
env ROOT_DIR=/apps/myapp
env HTTP_PORT=8080
env USER=java_user
env JAVA_HOME=/usr/lib/jvm/java-6-openjdk
env JVM_OPTIONS="-Xms64m -Xmx256m"
env APP_OPTIONS="--httpPort=$HTTP_PORT"
env LOGFILE=/var/log/myapp.log

script
  exec su -c "$JAVA_HOME/bin/java $JVM_OPTIONS \
    -jar $ROOT_DIR/myjar.jar $APP_OPTIONS > $LOGFILE 2>&1" $USER
end script
```

You should read the Changing User section section before using this technique though.

11.46.1 Alternative Method

Here is how you might run a Java application which calls fork(2) some number of times:

```
exec start-stop-daemon --start --exec $JAVA_HOME/bin/java \
  -- $JAVA_OPTS -jar $SOMEWHERE/file.war
```

Again, you should read the Changing User section section before using this technique.

11.47 Ensure a Directory Exists Before Starting a Job

This is a good use of the `pre-start` stanza:

```
env DIR=/var/run/myapp
env USER=myuser
env GROUP=mygroup
env PERMS=0755

pre-start script
  mkdir $DIR              || true
  chmod $PERMS $DIR       || true
```

```
  chown $USER:$GROUP DIR || true
end script
```

11.48 Run a GUI Application

To have Upstart start a GUI application, you first need to ensure that the user who will be running it has access to the X display. This is achieved using the `xhost` command.

Once the user has access, the method is the same as usual:

```
env DISPLAY=:0.0
exec xclock -update 1
```

11.49 Run an Application through GNU Screen

If you want Upstart to create a GNU Screen (or Byobu) session to run your application in, this is equally simple:

```
exec su myuser -c "screen -D -m -S MYAPP java -jar MyApp.jar"
```

11.50 Run Upstart in a chroot Environment

11.50.1 chroot Workaround for Older Versions of Upstart

Older versions of Upstart jobs cannot be started in a chroot(2) environment [17] because Upstart acts as a service supervisor, and processes within the chroot are unable to communicate with the Upstart running outside of the chroot. This will cause some packages that have been converted to use Upstart jobs instead of init scripts to fail to upgrade within a chroot.

Users are advised to configure their chroots with `/sbin/initctl` pointing to `/bin/true`, with the following commands *run within the chroot*:

```
dpkg-divert --local --rename --add /sbin/initctl
ln -s /bin/true /sbin/initctl
```

11.50.2 chroots in Ubuntu Natty

The version of Upstart in Ubuntu Natty now has full chroot(2) support. This means that if initctl is run as user `root` from within a chroot the Upstart init daemon (outside the chroot) will honour requests from within the chroot to manipulate jobs within the chroot.

What all this means is that you no longer need to use `dpkg-divert` and can control chroot jobs from within the chroot environment exactly as you would control jobs outside a chroot environment. There are a number of caveats and notes to consider though:

- Within the chroot, only jobs within the chroot are visible
- Within the chroot, only jobs within the chroot can be manipulated.
- It is only possible to view and control such chroot jobs from within the chroot.

  That is to say, the "outer" system cannot manipulate jobs within the chroot.
- Due to the design of this feature, Upstart will not be able to detect changes to job configuration files within the chroot until a process within the chroot has either manipulated a job, or listed one or more

jobs.

- Chroot support can be disabled at boot by passing the "`--no-sessions`" option on the Grub kernel command-line.

  See Add --verbose or --debug to the kernel command-line for details of how to add values to the grub kernel command-line.

  If chroots are disabled, running Upstart commands within a chroot will affect jobs outside the chroot only.

- If a job is run in a chroot environment (such as provided by schroot(1)), exiting the chroot will kill the job.

11.51 Record all Jobs and Events which Emit an Event

For example, if you want to record all jobs which emit a started event:

```
# /etc/init/debug.conf
start on started
script
  exec 1>>/tmp/log.file
  echo "$0:$$:`date`:got called. Environment of job $JOB was:"
  env
  echo
end script
```

You could also log details of all jobs (except the debug job itself) which are affected by the main events:

```
# /etc/init/debug.conf
start on ( starting JOB!=debug \
  or started JOB!=debug \
  or stopping JOB!=debug \
  or stopped JOB!=debug )
script
  exec 1>>/tmp/log.file
  echo -n "$UPSTART_JOB/$UPSTART_INSTANCE ($0):$$:`date`:"
  echo    "Job $JOB/$INSTANCE $UPSTART_EVENTS. Environment was:"
  env
  echo
end script
```

Note that the `$UPSTART_JOB` and `$UPSTART_INSTANCE` environment variables refer to *the debug job itself*, whereas `$JOB` and `$INSTANCE` refer to *the job which the debug job is triggered by*.

11.52 Integrating your New Application with Upstart

Integrating your application into Upstart is actually very simple. However, you need to remember that Upstart is *NOT* "System V" (aka "SysV"), so you need to think in a different way.

With SysV you slot your service script between other service scripts by specifying a startup number. The SysV init system then runs each script in numerical order. This is very simple to understand and use, but highly inefficient in practical terms since it means the boot cannot be parallelised and thus cannot be optimized.

11.53 Block Another Job Until Yours has Started

It is common that a particular piece of software, when installed, will need to be started before another. The logical conclusion is to use the 'starting' event of the other job:

```
start on starting foo
```

This will indeed, block foo from starting until our job has started.

But what if we have multiple events that we need to delay:

```
start on starting foo or starting network-services
```

This would seem to make sense. However, if we have a time-line like this:

```
starting foo
starting our job
starting network-services
started network-services
```

Network-services will actually NOT be blocked. This is because upstart only blocks an event if that event causes change in the *goal* of the service. So, we need to make sure upstart waits every time. This can be done by using a "wait job":

```
# myjob-wait
start on starting foo or starting network-services
stop on started myjob or stopped myjob
instance $JOB
normal exit 2
task
script
  status myjob | grep -q 'start/running' && exit 0
  start myjob || :
  sleep 3600
end script
```

This is a bit of a hack to get around the lack of state awareness in Upstart. Eventually this should be built in to upstart. The job above will create an instance for each JOB that causes it to start. It will try and check to see if it's already running, and if so, let the blocked job go with exit 0. If it's not running, it will set the ball in motion for it to start. By doing this, we make it very likely that the stopped or started event for myjob will be emitted (the only thing that will prevent this, is a script line in 'myjob' that runs 'stop'). Because we know we will get one of those start or stopped events, we can just sleep for an hour waiting for upstart to kill us when the event happens.

11.54 Controlling Upstart using D-Bus

Upstart contains its own D-Bus server which means that initctl and any other D-Bus application can control Upstart. The examples below use `dbus-send`, but any of the D-Bus bindings could be used.

11.54.1 Query Version of Upstart

To emulate `initctl version`, run:

```
$ dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart org.freedesktop.DBus.Properties.Get string:com.ubuntu.Upstart0_6 string:version
```

Note: this is querying the version of `/sbin/init`, *not* the version of `initctl`. For the latter, see initctl version.

11.54.2 Query Log Priority

To emulate `initctl log_priority` and show the current log priority, run:

```
$ dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart org.freedesktop.DBus.Properties.Get string:com.ubuntu.Upstart0_6 string:log_priority
```

11.54.3 Set Log Priority

To emulate `initctl log_priority <value>` and *set* a new log priority, run:

```
$ priority=debug
$ sudo dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart org.freedesktop.DBus.Properties.Set string:com.ubuntu.Upstart0_6 string:log_priority variant:string:$priority
```

11.54.4 List all Jobs via D-Bus

To emulate `initctl list`, run:

```
$ dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart com.ubuntu.Upstart0_6.GetAllJobs
```

11.54.5 Get Status of Job via D-Bus

To emulate `initctl status myjob`, run:

```
$ job=myjob
$ dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart/jobs/${job}/_ org.freedesktop.DBus.Properties.GetAll string:''
```

Note that this will return information on all running job instances of `myjob`.

11.54.6 Emit an Event

To emulate `initctl emit <event>`, run:

```
$ event=foo
$ sudo dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart com.ubuntu.Upstart0_6.EmitEvent string:$event array:string: boolean:true
```

To emulate `initctl emit --no-wait <event> A=B c='hello world' D=123.456`, run:

```
$ event=foo
$ sudo dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart com.ubuntu.Upstart0_6.EmitEvent string:$event array:string:'A=B',"C='hello world'",D=123.456 boolean:false
```

11.54.7 Get Jobs start on and stop on Conditions via D-Bus

To show a jobs start on condition:

```
$ job=cron
$ for condition in start_on stop_on
> do
>     dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart/jobs/$job org.freedesktop.DBus.Properties.Get string:com.ubuntu.Upstart0_6.Job string:$condition
> done
```

If you have a job with a start on condition like this:

```
start on (starting foo A=B or (stopping bar C=D and (stopped baz E=F G=H I=J or foo)))
```

... a dbus-send(1) query like the one above for start on will return an "array of arrays of strings":

```
method return sender=:1.629 -> dest=:1.630 reply_serial=2
   variant       array [
         array [
            string "starting"
```

```
            string "foo"
            string "A=B"
         ]
         array [
            string "stopping"
            string "bar"
            string "C=D"
         ]
         array [
            string "stopped"
            string "baz"
            string "E=F"
            string "G=H"
            string "I=J"
         ]
         array [
            string "foo"
         ]
         array [
            string "/OR"
         ]
         array [
            string "/AND"
         ]
         array [
            string "/OR"
         ]
      ]
```

This will require a little massaging. Every inner array entry represents one of the following:

- an Event
- an operator ("`and`" or "`or`")

For event arrays, the first element is the event name and subsequent elements represent the events environment variables.

Note too that the entire start on expression has been encoded using Reverse Polish Notation (RPN) since this is a convenient format to represent the condition (particularly when you consider that they are represented internally as trees).

Normally, you don't need to get involved with RPN since initctl show-config converts the RPN back into the original form as specified in the Job Configuration file.

11.54.8 To Start a Job via D-Bus

To emulate `initctl start myjob`, run:

```
# job=myjob
# dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart/jobs/${job} com.ubuntu.Upstart0_6.Job.Start array:string: boolean:true
```

Note that you must be `root` to manipulate system jobs.

11.54.9 To Stop a Job via D-Bus

To emulate `initctl stop myjob`, run:

```
# job=myjob
# dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart/jobs/${job} com.ubuntu.Upstart0_6.Job.Stop array:string: boolean:true
```

Note that you must be `root` to manipulate system jobs.

11.54.10 To Restart a Job via D-Bus

To emulate `initctl restart myjob`, run:

```
# job=myjob
# dbus-send --system --print-reply --dest=com.ubuntu.Upstart /com/ubuntu/Upstart/jobs/${job} com.ubuntu.Upstart0_6.Job.Restart array:string: boolean:true
```

Note that you must be `root` to manipulate system jobs.

11.55 Establish Blocking Job

Image you have just run the following command and it has "blocked" (appeared to hang):

```
# initctl emit event-A
```

The reason for the block is that the `event-A` event changes the goal of "some job", and until the goal has changed, the `initctl` command will block.

But *which* job is being slow to change goal? It is now possible to hone in on the problem using `initctl show-config` in a script such as this:

```
#!/bin/sh
# find_blocked_job.sh

[ $# -ne 1 ] && { echo "ERROR: usage: $0 <event>"; exit 1; }
event="$1"

# obtain a list of jobs (removing instances)
initctl list | awk '{print $1}' | sort -u | while read job
do
  initctl show-config -e "$job" |\
    egrep "(start|stop) on \<event\>" >/dev/null 2>&1
  [ $? -eq 0 ] && echo $job
done
```

This will return a list of jobs, one per line. One of these will be the culprit. Having identified the problematic job, you can debug using techniques from the Debugging section.

11.56 Determine if a Job is Disabled

To determine if a job has been disabled from starting automatically:

```
$ job=foo
$ initctl show-config $job | grep -q "^  start on" && echo enabled || echo disabled
```

11.57 Visualising Jobs and Events

Use the initctl2dot(8) facility. See [25] for further details and examples.

11.58 Sourcing Files

You need to take care when "sourcing" a script or configuration file into a script section for a number of reasons. Suppose we have the following:

```
script
. /etc/default/myapp.cfg
. /etc/myapp/myapp.cfg
echo hello > /tmp/myapp.log
end script
```

Assume that file `/etc/myapp/myapp.cfg` does *NOT* exist.

11.58.1 Develop Scripts Using `/bin/sh`

Firstly, if you developed this script using the bash(1) shell, before you put it into a job configuration file), all would be well. However, as noted, Upstart runs all jobs with `/bin/sh -e`. What you will find is that if you run the script above under `/bin/sh`, in all likelihood the file will *never* be created since regardless of whether you specify "`-e`" or not, the dash(1) shell (which `/bin/sh` is linked to on Ubuntu systems) has different semantics when it comes to sourcing compared with `/bin/bash`.

Therefore, to avoid surprises later on:

- Always develop your scripts using "`/bin/sh -e`".
- Always code defensively.

  For example, it would be better to write the script above as:

  ```
  script
    [ -f /etc/default/myapp.cfg ] && . /etc/default/myapp.cfg
    [ -f /etc/myapp/myapp.cfg ]   && . /etc/myapp/myapp.cfg
    echo hello > /tmp/myapp.log
  end script
  ```

Or maybe even like this to minimise mistakes:

```
script
  files="\
  /etc/default/myapp.cfg
  /etc/myapp/myapp.cfg
  "

  for file in $files
  do
    [ -f "$file" ] && . "$file"
  done
  echo hello > /tmp/myapp.log
end script
```

11.58.2 `ureadahead`

Most modern Linux systems attempt to optimise the boot experience by pre-loading files early on in the boot sequence. This allows hard disks can minimise expensive (slow) seek operations.

On Ubuntu, this job is accomplished using ureadahead(8), which was designed with *both* spinning hard disk and SSD drives in mind. However, if your job configuration files start reading files from all over the disk, you will be potentially slowing down the boot as the disk is then forced to seek across the filesystem, looking for your files.

The general advice is therefore to put your configuration variables *inside* the job configuration file itself where possible.

11.59 Determining How to Stop a Job with Multiple Running Instances

As explained in the initctl status section, a job that has multiple running instances will show the specific (unique) instance value within brackets:

```
$ initctl list | grep ^network-interface-security
network-interface-security (network-manager) start/running
network-interface-security (network-interface/eth0) start/running
network-interface-security (network-interface/lo) start/running
network-interface-security (networking) start/running
```

In the example output above there are four instances of the `network-interface-security` job running with the unique instances values of:

- "`network-manager`"
- "`network-interface/eth0`"
- "`network-interface/lo`"
- "`networking`"

So how do we stop one of these jobs? Lets try to work this out without looking at initctl(8) manual page:

```
# stop network-interface-security network-interface/eth0
stop: Env must be KEY=VALUE pairs
```

That clearly doesn't work. The problem is that we have provided the *value* to the instance variable, but we haven't named the instance variable that the given value corresponds to. But how do we establish the instance variable *name*?

There are 2 options:

- look at the corresponding Job Configuration File.

  `/etc/init/network-interface-security.conf` in this example.
- Use a trick to get Upstart to tell you the name:

  ```
  $ status network-interface-security
  status: Unknown parameter: JOB
  ```

  This shows us the name of the instance variable is "`JOB`".

We are now in a position to stop a particular instance of this job:

```
# stop network-interface-security JOB=network-interface/eth0
network-interface-security stop/waiting
```

The job instance has now been stopped. To prove it:

```
# status network-interface-security JOB=network-interface/eth0
status: Unknown instance: network-interface/eth0
# initctl list | grep ^network-interface-security | grep network-interface/eth0
#
```

11.60 Logging Boot and Shutdown Times

If you want to create a log of when your system starts and stops, you could do something like this:

```
start on filesystem or runlevel [06]

env log=/var/log/boot-times.log

script
  action=$(echo "$UPSTART_EVENTS" | grep -q filesystem && echo boot || echo shutdown)
  echo "`date`: $action" >> $log
end script
```

Note that you do *not* need to specify a stop on condition: you want this job to *start* both "around" the time of system startup (when the disks are writeable, hence the use of the `fileysystem` event) and shutdown.

If you want a more accurate method, you would need to have a job start on startup. The slight issue here is that when Upstart emits that first event, there is no guarantee of writeable disks. However, this can be overcome using a bit of thought...

First, create a "`record-boot-time.conf`" job configuration file to record the time of the "boot" (initial Upstart event):

```
start on startup

exec initctl emit boot-time TIME=$(date '+%s')
```

This job emits an event containing a variable specifying the time in seconds since the Epoch.

Now, create a second "`log-boot-time.conf`" job configuration file to actually log the boot time:

```
start on boot-time and filesystem

log=/var/log/boot-times.log

script
  echo "system booted at $TIME" >>$log
end script
```

Since the "`log-boot-time`" job specifies the "`booted`" event emitted by the "`record-boot-time`" job, Upstart will retain knowledge of this event until it is able to run the second job. The "`record-boot-time`" job can then simply make use of the "`TIME`" variable set by the first job.

11.61 Running an Alternative Job on a tty

Here's a silly example of how to run a custom job on a particular tty. It asks the user to guess a random number. If after 3 attempts they fail to guess the correct number, the job ends. However, if they guess successfully, the are allowed to login. This won't win any scripting competitions, but you get the idea.

WARNING - *DO NOT USE THIS ON A REAL SYSTEM* unless you want to get hacked, or fired or both!:

```
# Get the user to guess the number. If they get it right, let them
# login.

start on runlevel [23]
stop on runlevel [!23]

env tty=tty9

# XXX: Ensure job is connected to the terminal device
console output

script
  # XXX: Ensure all standard streams are connected to the console
  exec 0</dev/$tty >/dev/$tty 2>&1
  clear
  trap '' INT TERM HUP
  RANDOM=$(dd if=/dev/urandom count=1 2>/dev/null|cksum|cut -f1 -d' ')
  answer=$(((RANDOM % 100) + 1))
  attempt=0
  max=3
  got=0

  while [ $attempt -lt $max ]
  do
    attempt=$((attempt+1))
    echo -n "Guess the number (1-100, attempt $attempt of $max): "
    read guess
    if [ "$guess" -eq "$answer" ]
    then
      got=1
      break
    else
      echo "Wrong"
    fi
  done

  [ "$got" = 0 ] && stop

  exec /sbin/getty -8 38400 $tty
end script
```

The important lines are:

```
console output
```

... and:

```
exec 0</dev/$tty >/dev/$tty 2>&1
```

11.62 Creating a SystemV Service that Communicates with Upstart

There are occasions when you want to have a SystemV service start an Upstart job. However, you must take care as shown in the example below...

Image we create a SysV service as `/etc/init.d/myservice`. This service needs another service to be running but that other service is actually an Upstart job (`/etc/init/myjob.conf`).

The Upstart job specifies a start on condition of:

```
start on filesystem and static-network-up and myservice-server-running
```

So, job `myjob` will only start once all three of the events specified are emitted and the `myservice-server-running` event is being emitted by `/etc/init.d/myservice` like this:

```
initctl emit myservice-server-running
```

This all looks perfectly reasonable and in fact it is... generally.

However, consider what would happen if the package containing `/etc/init.d/myservice` happened to attempt to restart that service having installed it (to make sure it is running immediately after installation)...

1. `/etc/init.d/myservice` is run.
2. `/etc/init.d/myservice` calls "`initctl emit myservice-server-running`".
3. Upstart emits the `myservice-server-running` event.

Nothing magical here yet. Or is there? Since job `myjob` will only be started when all three of the events specified in its start on condition are true, this job cannot yet be started. Why? Because the `filesystem` and `static-network-up` events have *already* been emitted early in the boot (see Ubuntu Well-Known Events (ubuntu-specific)).

What this means is that the job `myjob` will *never* start *post boot* if those two events it cares about *have already been emitted*. Any yet, the SysV job and the Upstart event combinations are perfectly valid *on boot*. Note too that because those two events will not be re-emitted, the initctl emit will block (appear to hang) since Upstart is waiting for those two events to be emitted.

The solution to this is very simple: make the SysV job only emit the event in question *on boot*:

```
# Only emit the event 'on boot' to ensure the SysV service
# does not "hang" (block) due to events the ``myjob`` job requires
# never being re-emitted post-boot. We do this by checking for one of
# Upstarts  standard environment variables which will only be run when
# the Upstart SysV compatibility system is running the SysV service in
# question.
[ -n "$UPSTART_JOB" ] && initctl emit myservice-server-running
```

A slightly different method is to emit a signal by running initctl with the `--no-wait` option like this:

```
[ -n "$UPSTART_JOB" ] && initctl emit --no-wait myservice-server-running
```

See Signals and Standard Environment Variables.

12 Test Your Knowledge

12.1 Questions about *start on*

Consider the following start on condition:

```
start on startup or starting stopped or stopping started
```

Questions (answers provided in footnote links):

Question: Is this a legal condition?
Answer: [1]

Question: What standard Upstart tool could you use to help explain the expression?
Answer: [2]

Question: Explain the condition.
Answer: [3]

Question: How many times could this job be run assuming all other jobs on the system run exactly once?
Answer: [4]

12.2 General Questions

What is wrong with the following job configuration file?:

```
start on startup

script
  echo hello > /tmp/foo.log
end script
```

Answer: [5]

Whas is wrong with the following job configuration file?:

```
start on runlevel [2345]

env CONFIG=/etc/default/myapp

expect fork
respawn

script
  enabled=$(grep ENABLED=1 $CONFIG)
  [ -z "$enabled" ] && exit 0
  /usr/bin/myapp
end script
```

Answer: [6]

13 Common Problems

13.1 Cannot Start a Job

If you have just created or modified a job configuration file such as `/etc/init/myjob.conf`, but `start` gives the following error when you attempt to start it:

```
start: Unknown job: myjob
```

The likelihood is that the file contains a syntax error. The easiest way to establish if this is true is by running the init-checkconf command.

If you are wondering why the original error couldn't be more helpful, it is important to remember that the job control commands (`start`, `stop` and `restart`) and `initctl` communicate with Upstart over D-Bus. The problem here is that Upstart rejected the invalid `myjob.conf`, so attempting to control that job over D-Bus is nonsensical - the job does not exist.

13.2 Cannot stop a job

If start is hanging or seems to be behaving oddly, the chances are you have misspecified the expect stanza. See expect and How to Establish Fork Count.

13.3 Strange Error When Running `start/stop/restart` or `initctl emit`

If you attempt to run a job command, or emit an event and you get a D-Bus error like this:

```
$ start myjob
start: Rejected send message, 1 matched rules; type="method_call", sender=":1.58" (uid=1000 pid=5696 comm="start) interface="com.ubuntu.Upstart0_6.Job" member="Start" error name="(unset)" requested_reply=0 destination="com.ubuntu.Upstart" (uid=0 pid=1 comm="/sbin/init"))
```

The problem is caused by not running the command as `root`. To resolve it, either "`su -`" to `root` or use a facility such as sudo(8):

```
# start myjob
myjob start/running, process 1234
```

The reason for the very cryptic error is that the job control commands (`start`, `stop` and `restart`) and `initctl` communicate with Upstart over D-Bus.

13.4 The `initctl` command shows "the wrong PID"

The likelihood is that you have mis-specified the type of application you are running in the job configuration file. Since Upstart traces or follows fork(2) calls, it needs to know how many forks to expect. If your application forks *once*, specify the following in the job configuration file:

```
expect fork
```

However, if your application forks *twice* (which all daemon processes *should* do), specify:

```
expect daemon
```

See also Alternative Method.

13.5 Symbolic Links don't work in `/etc/init`

Upstart does not monitor files which are symbolic links since it needs to be able to guarantee behaviour and if a link is broken or cannot be followed (it might refer to a filesystem that hasn't yet been mounted for example), behaviour would be unexpected, and thus undesirable. As such, all system job configuration files must live in or below `/etc/init` (although user jobs can live in other locations).

13.6 Sometimes `status` shows PID, but other times does not

You may have noticed that when you start certain jobs manually using start, sometimes the output will show the PID of the process associated with that job. However, other times, no PID is shown. Why?

This behaviour is observed when the job runs to completion very quickly. If your system has minimal load the job will start *and finish* before the initctl status command has a chance to query its PID from Upstart. Whereas if your system is busy you may well see a PID displayed since Upstart was able to return the PID details to `status` before the job finished.

The behaviour is similar to the following shell code:

```
(sleep 0.01 &) ; ps -fU $USER | grep sleep | grep -v grep
```

It is unlikely that you will get any output from this command (since the `sleep 0.01` command will run to completion before the grep(1) calls get a chance filter the ps(1) output. However, change the time for that subshell to run, and you will see the PID:

```
(sleep 5 &) ; ps -fU $USER | grep sleep | grep -v grep
```

See initctl status.

14 Testing

Before embarking on rewriting your systems job configuration files, think *very*, **very** carefully.

We would advise strongly that before you make your production server unbootable that you consider the following advice:

1. Version control any job configuration files you intend to change.

   You could employ the version stanza to help in this regard.

2. Test your changes in a Virtual Machine.
3. Test your changes on a number of non-critical systems.
4. Backup all your job configuration files to *both*:

    - An alternate location on the local system

      (Allowing them to be recovered quickly if required).

    - At least one other suitable alternate backup location.

15 Daemon Behaviour

Upstart manages the running of jobs. Most of these jobs are so-called "daemons", or programs that:

- run detached from a terminal device.
- require no user input.

- generate no output to the standard output streams "`stdout`" and "`stderr`".

To manage such daemons, Upstart expects a daemon to adhere to the following rules:

- The daemon should advertise if it forks once, or if it double-forks.

  This allows the Administrator to establish the correct value for the important expect stanza.

- The daemon should not install a `SIGCHLD` handler of its own.

  This is a problem when the job incorrectly specifies expect fork for a daemon (that should have been specified as expect daemon) since Upstart waits for a single fork but the daemon double forks however Upstart never gets notification of the first process exiting since a `SIGCHLD` signal is never generated for that process.

  This leads to a "stuck job (see Implications of Misspecifying expect).

  this could stop Upstart from determining when the process has finished if the expect stanza is mis-specified as expect fork.

- The daemon should ensure that when it completes the second fork that it is fully initialized, since Upstart uses the fork count to determine service readiness (see expect).

- When sent a `SIGHUP` signal, Upstart will expect the daemon to:

  - do whatever is necessary to re-initialize itself, for example by re-reading its configuration file.

    This behaviour ensures that "`initctl reload <job>`" will work as expected.

  - retain its current PID: if the daemon calls fork(2) on receiving this signal. See expect.

    This behaviour ensures that Upstart can continue to manage the PID.

- When sent a `SIGTERM` signal, Upstart expects the daemon to shut down cleanly.

  If a daemon does not shut down on receipt of this signal in a timely fashion, Upstart will send it the unblockable `SIGKILL` signal.

- Signalling "readiness": Since Upstart tracks forks, it can only assume that once the *final* fork(2) call has been made (as indicated by the expect stanza specification), that the job is "ready" to accept work from other parts of the system.

  This generally works very well, but can be an issue for daemons which start relatively quickly, but which are not considered "ready" to service requests until some arbitrary future time.

  A good example of this scenario would be a database server which starts but which can only be considered "ready" or "online" once it has finished replaying some transaction logs (which take some time to process). In this scenario, there are two approaches:

  1. Create a post-start section that performs some check and only returns once the service is "ready".

  2. If the service accepts incoming network connnections, modify it to make use of the upstart-socket-bridge.

- The daemon should not make use of the ptrace(2) system call (atleast not until it has initialized itself fully).

  This ensures that Upstart is able to track the daemons pid. See expect.

The following are recommendations if you are writing a new daemon:

- If the daemon does not *need* to run as root, it should drop its privilege level (using setuid(2) and setgid(2)).

- If a daemon is able to drop its privilege level to any non-root user, it should provide a documented way (such as command-line options) for the invoker to specify the user and group to have the daemon eventually run as.

16 Precepts for Creating a Job Configuration File

16.1 Determining the value of `expect`

The Expect section explains how to determine the value of the `expect` stanza. Note that you *should not* introduce the respawn stanza until you are fully satisfied you have specifed the expect stanza correctly.

16.2 `start on` and `stop on` condition

See How to Establish a Jobs start on and stop on Conditions.

16.3 Services

- If your job is a service, identify the correct value for the expect stanza.

  Once you have decided on the correct value:

  1. start the job:

     ```
     $ sudo start myjob
     ```

  2. Check the PID of the job matches the expected PID:

     ```
     $ actual_pid=$(pidof myapp)
     $ upstart_pid=$(status myjob | awk '{print $NF}')
     $ [ "$actual_pid" = "$upstart_pid" ] || echo "ERROR: pid "
     ```

  3. Stop the job:

     ```
     $ sudo stop myjob
     ```

  4. Ensure the PID no longer exists:

     ```
     $ [ -z "$(pidof myapp)" ] || echo "ERROR: myapp still running"
     ```

- *Only* once you have specified the correct expect stanza should you introduce the respawn stanza since if you introduce it at the outset, this will just confuse your understanding, particulary if the expect stanza has been misspecified.

16.4 Ubuntu Rules ( )

On Ubuntu, the following rules should be adhered to:

16.4.1 Console attributes

Jobs that specify console output or console owner should **NOT** modify the attributes of the console (`/dev/console`), for example by using tcsetattr(3).

The reason for this being that Plymouth, the graphical boot splash application, needs full control over the console on boot and shutdown.

17 Debugging

17.1 Obtaining a List of Events

To obtain a list of events that have been generated by your system, do one of the following:

17.1.1 Add `--verbose` or `--debug` to the kernel command-line

By adding `--verbose` or `--debug` to the kernel command-line, you inform Upstart to enter either verbose or debug mode. In these modes, Upstart generates extra messages which can be viewed in the system log. See initctl log-priority.

Assuming an standard Ubuntu Natty system, you could view the output like this:

```
grep init: /var/log/syslog
```

Note that until Upstart 1.3 it was difficult to get a complete log of events for the simple reason that when Upstart starts, there is no system logger running to record messages from Upstart (since Upstart hasn't started it yet!) However, Upstart 1.3 writes these "early messages" to the kernel ring buffer (see dmesg(1)) such that by considering the kernel log *and* the system log, you can obtain a complete list of events from the initial "`startup`". So, for a standard Ubuntu Oneiric system, you would do:

```
grep init: /var/log/kern.log /var/log/syslog
```

The mechanism for adding say the `--debug` option to the kernel command-line is as follows:

1. Hold down SHIFT key before the splash screen appears (this will then display the grub menu).
2. Type, "e" to edit the default kernel command-line.
3. Use the arrow keys to go to the end of the line which starts "`linux /boot/vmlinuz ...`".
4. Press the `END` key (or use arrows) to go to end of the line.
5. Add a space followed by "`--debug`" (note the two dashes).
6. Press `CONTROL+x` to boot with this modified kernel command line.

17.1.2 Change the log-priority

If you want to see event messages or debug messages "post boot", change the log priority to `debug` or `verbose`. See initctl log-priority.

17.2 See the Environment a Job Runs In

To get a log of the environment variables set when Upstart ran a job you can add simple debug to the appropriate `script` section. For example:

```
script
  echo "DEBUG: `set`" >> /tmp/myjob.log

  # rest of script follows...
end script
```

Alternatively you could always have the script log to the system log:

```
script
  logger -t "$0" "DEBUG: `set`"

  # rest of script follows...
end script
```

Or, have it pop up a GUI window for you:

```
env DISPLAY=:0.0

script
  env | zenity --title="got event $UPSTART_EVENTS" --text-info &
end script
```

For the full details, install the procenv(1) utility and run this as a job. On a Debian Sid or Ubuntu Raring (or newer) system:

```
$ sudo apt-get -y install procenv
$ cat <<EOT | sudo tee /etc/init/procenv.conf
exec /usr/bin/procenv
EOT
$ sudo start procenv
$ sudo cat /var/log/upstart/procenv.log
```

17.3 Checking How a Service Might React When Run as a Job

You may find that your service runs fine when executed from the command-line, but does not work initially when you start testing it with Upstart. This is because the environment the service is run in when started by Upstart is potentially radically different to your interactive user (or even root user) environment.

To discover exactly what sort environment Upstart provides, see the `procenv` example in See the Environment a Job Runs In.

Before you even put your service into a Job Configuration File, try the following test which simulates an Upstart-like environment.

Assuming your service is `/usr/bin/mydaemon` and you want to run it as user `root`:

```
$ user=root
$ cmd=/usr/bin/mydaemon
$ su -c 'nohup env -i $cmd </dev/null >/dev/null 2>&1 &' $user
```

That command will run `/usr/bin/mydaemon`:

- as user `$user` (`root` here, but maybe not for you if you've used setuid)
- with no associated terminal
- parented to init
- with no environment

Or, if you want to set a user *and* a group, use sudo(8) (or maybe su(1) and newgrp(1)):

```
$ user=user1
$ group=group2
```

```
$ cmd=/usr/bin/mydaemon
$ ( sudo -u $user -g $group nohup env -i $cmd < /dev/null > /dev/null 2>&1 ) &
```

For the `sudo` example, you should first check that `$user` is able to run `$cmd`.

If your service is unable to run in one of these environments, it is also likely to fail when run as a Job.

17.4 Obtaining a log of a Script Section

17.4.1 Upstart 1.4 (and above)

Upstart 1.4 provides automatic logging of all job output.

See console log for further details.

17.4.2 Versions of Upstart older than 1.4

This technique relies on a trick relating to the early boot process on an Ubuntu system. On the first line below `script` stanza, add:

```
exec >>/dev/.initramfs/myjob.log 2>&1
set -x
```

This will ensure that `/bin/sh` will log its progress to the file named `/dev/.initramfs/myjob.log`.

The location of this file is special in that `/dev/.initramfs/` will be available early on in the boot sequence (before the root filesystem has been mounted read-write).

Note that newer releases of Ubuntu mount `/run/` read-writeable very early on in the boot process too.

17.5 Log Script Section Output to Syslog

There are two techniques you can use to do this:

Use the same technique as shown in Obtaining a log of a Script Section, but change the file to */dev/kmsg*. This will send the data to the kernels ring buffer. Once the syslog(3) daemon starts, this data will be redirected to the system log file:

```
script
  exec >/dev/kmsg 2>&1
  echo "this data will be sent to the system log"
end script
```

17.6 Checking a Job Configuration File for Syntax Errors

See init-checkconf.

17.7 Check a Script Section for Errors

Upstart runs your job using `/bin/sh -e` for safety reasons: scripts running as the `root` user need to be well-written! But how can you check to ensure that your script sections contain valid (syntactically correct at least) shell fragments? Simply run the init-checkconf script, which performs these checks automatically.

17.7.1 Older versions of Upstart

To check that you haven't made a (shell) syntax error in your `script` section, you can use `sed` like this:

```
$ /bin/sh -n <(sed -n '/^script/,/^end script/p' myjob.conf)
```

Or for a `pre-start script` section:

```
$ /bin/sh -n <(sed -n '/^pre-start script/,/^end script/p' myjob.conf)
```

No output indicates no syntax errors.

Alternatively, you could wrap this into a script like this:

```
#!/bin/sh
# check-upstart-script-sections.sh

[ $# -ne 1 ] && { echo "ERROR: usage: $0 <conf_file>"; exit 1; }
file="$1"

[ ! -f "$file" ] && { echo "ERROR: file $file does not exist" >&2; exit 1; }

for v in pre-start post-start script pre-stop post-stop
do
  if egrep -q "\<${v}\>" $file
  then
    sed -n "/^ *${v}/,/^ *end script/p" $file | \
      sh -n || echo "ERROR in $v section"
  fi
done
```

And run it like this to check all possible script sections for errors:

```
$ check-upstart-script-sections.sh myjob.conf
```

17.8 Debugging a Script Which Appears to be Behaving Oddly

If a `script` section appears to be behaving in an odd fashion, the chances are that one of the commands is failing. Remember that Upstart runs every `script` section using `/bin/sh -e`. This means that if *any* simple command fails, the shell will exit. For example, if file */etc/does-not-exist.cfg* does not exist in the example below **the script will exit before the shell runs the `if` test**:

```
script
  grep foo /etc/does-not-exist.cfg >/dev/null 2>&1
  if [ $? -eq 0 ]
  then
    echo ok
  else
    echo bad
  fi
end script
```

In other words, you will get *no* output from this script if the file grep is attempting to operate on does not exist.

The common idiom to handle possible errors of this type is to convert the simple expression into an expression guaranteed to return true:

```
script
  # ensure this statement always evaluates to true
  command-that-might-fail || true

  # ditto
  another-command || :
end script
```

See `man sh` for further details.

18 Recovery

If you do something really bad or if for some reason Upstart fails, you might need to boot to recovery mode and revert your job configuration file changes. In Ubuntu, you can therefore either:

18.1 Boot into Recovery Mode

Select the "recovery" option in the Grub boot menu

This assumes that Upstart (init(8) itself) is usable.

Note that you need to hold down the `SHIFT` key to see the Grub boot menu.

18.2 Boot to a shell directly

If Upstart (init(8)) itself has broken, you'll need to follow the steps below. By specifying an alternate "initial process" (here a shell) it is possible to repair the system.

1. Hold down SHIFT key before the splash screen appears (this will then display the grub menu).
2. Type, "e" to edit the default kernel command-line.
3. Use the arrow keys to go to the end of the line which starts "`linux /boot/vmlinuz ...`".
4. Press the `END` key (or use arrows) to go to end of the line.
5. Add a space followed by "`init=/bin/sh`".
6. If the line you are editing contains "*quiet*" and/or "*splash*", remove them.
7. Press `CONTROL+x` to boot with this modified kernel command line.
8. When the shell appears you will need to remount the root filesystem read-write like this:

   ```
   # mount -oremount,rw /
   ```

You can now make changes to your system as necessary.

19 Advanced Topics

19.1 Changing the Default Shell

By default, Upstart uses "`/bin/sh`" to execute script sections. If you wish to change this behaviour, you have the following options:

- Link `/bin/sh` to your chosen shell [9].
- Copy your chosen shell to `/bin/sh`.
- Recompile Upstart specifying an alternative shell as follows:

```
# XXX: Note the careful quoting to retain double-quotes around the shell!
export CFLAGS=-DSHELL='\"/bin/bash\"'
./configure && make && sudo make install
```

  Note that you should consider such a change carefully since Upstart has to rely upon the shell. Remember too that **Upstart runs all script sections as the root user**.

- Use a "here document" (assuming your chosen shell supports them) within the Job Configuration Files you wish to run with a different shell:

```
script
/bin/bash <<EOT

echo "Hi - I am running under the bash shell"

date

echo "and so am I :)"

EOT
end script
```

  Note that currently, this technique is the only way (without modifying the Upstart source code) to run a shell *without* specifying the "`-e`" option (see dash(1) or bash(1) for details).

19.2 Running a script Section with Python

To run a script section with Python:

```
script

python - <<END

from datetime import datetime

today = datetime.now().strftime("%A")

fh = open("/tmp/file.txt", "w")
print >>fh, "Today is %s" % today
fh.close()
```

```
END

end script
```

19.3 Running a script Section with Perl

To run a script section with Perl:

```
script

perl - <<END

use strict;
use warnings;
use POSIX;

my $fh;
my $today = POSIX::strftime("%A", localtime);

open($fh, ">/tmp/file.txt");
printf $fh "Today is %s\n", $today;
close($fh);

END

end script
```

20 Development and Testing

20.1 Warnings

- Upstart runs as `root` so has full system privileges.
- If Upstart crashes...:

  ```
  Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000100
  [ 2.745566]
  [ 2.751931] Pid: 1, comm: false Not tainted 3.5.0-15-generic #22-Ubuntu
  [ 2.755489] Call Trace:
  [ 2.757068]  [<c15be842>] panic+0x81/0x17b
  [ 2.759206]  [<c104a6a5>] do_exit+0x745/0x7a0
  [ 2.761602]  [<c104a9a4>] do_group_exit+0x34/0xa0
  [ 2.764162]  [<c104aa28>] sys_exit_group+0x18/0x20
  [ 2.765231]  [<c15c8a94>] syscall_call+0x7/0xb
  ```

  ... your kernel panics!
- Unlike the kernel, if a new version of Upstart fails to work at all, there is no easy fix.

20.2 Precautions and Practises

Precautions and Practises:

- Every function asserts its arguments. This allows simple programming bugs to be found very quickly ("fail fast").
- Every function that returns a value must be checked and exceptions handled. GCC helps in this respect with the `__attribute__ ((warn_unused_result))` function attribute.
- Every function that returns newly-allocated memory has its prototype decorated using `__attribute__ ((malloc))`.
- No compiler warnings are allowed (`-Wall -Werror`).
- Every function or logical unit of functionality *must* have an associated set of tests.
- Every build of Upstart *must* pass all NIH and Upstart tests before being made available to users.
- The code is very well tested(using physical and virtual hardware, all architectures and containers).
- *All* code is peer-reviewed.
- All changes to the main `lp:upstart` code branch in Launchpad now automatically generate a mail to the Upstart mailing list.
- All bzr merge proposals raised on Upstart also result in a mail to the Upstart mailing list.
- Where possible, all new features add a `--no--<feature>` command-line option (allowing the feature to be disabled to provide a fall-back mechanism).

20.3 Code Style

- Use tabs.
- Every function, macro, structure, typedef and variable must be documented.
- Every function must specify what is returned on success and failure.
- Every function must check all possible parameters using .

See file: `upstart:HACKING`

20.4 Development Advice

- KISS and KIRS ("keep it readable silly")

  "Clever" code often outwits the author.

  Prefer to keep it simple, elegant and most of all readable. Bit-twiddlers and IOCCC champions need not apply.
- Do not use system calls or library calls if NIH already provides an alternative. That means:
  - No `malloc()`, `calloc()`, `strtok()`, `sprintf()`, *et cetera*.
  - You really need to familiarise yourself with NIH by reading the NIH source and the Upstart source.
- Don't just read the NIH and Upstart source, read the test code - it has comments too! ;-)
- Write code to be *testable*.
- Always consider security and performance.
  - DoS possibility?
  - Get the code security-reviewed.

- If you plan to work on some huge feature that will take you 6 months of effort, *PLEASE* alert the developers via the mailing list *BEFORE* you start since:
    - We may already be working on such a feature.
    - If your design doesn't fit in with the project, you're potentially facing a lot of avoidable rework.
- Always test on a range of hardware:
    - Physical and virtual.
    - 32-bit and 64-bit.
    - Intel/ARM/etc.

20.5 Setting up an Upstart Development Environment

```
$ sudo apt-get install build-dep upstart # cheat :)
$ bzr branch lp:upstart
$ cd upstart
$ ./configure --disable-silent-rules --enable-compiler-warnings \
    --disable-compiler-optimisations --disable-linker-optimisations \
    --enable-compiler-coverage
$ export CFLAGS="-fstack-protector --param=ssp-buffer-size=4 -Wformat -Werror=format-security"
$ make
$ cscope -Rbq && ctags
```

- You *need* to build the code before indexing since D-Bus bindings are auto-generated (using `nih-dbus-tool`).
- You *need* to use all those flags to enable all the compiler checks.

20.6 Setting up an Upstart+NIH Development Environment

Since Upstart makes such heavy use of NIH, it is often useful to build both Upstart and link it to a debug symbols build of NIH:

```
$ sudo apt-get install build-dep upstart libnih1 # cheat :)
$ prefix=/testing
$ mkdir $prefix
$ export PKG_CONFIG_PATH=${prefix}/lib/pkgconfig:$PKG_CONFIG_PATH
$ export ACDIR=${prefix}/share/aclocal:$ACDIR
$ export CFLAGS="-fstack-protector --param=ssp-buffer-size=4 -Wformat \
                 -Werror=format-security -ggdb3 -fno-inline"
$ bzr branch lp:libnih
$ cd libnih
$ ./configure --disable-silent-rules --enable-compiler-warnings \
    --disable-compiler-optimisations --disable-linker-optimisations \
    --enable-compiler-coverage && make && make install
$ cd -
$ bzr branch lp:upstart
$ cd upstart
$ ./configure --disable-silent-rules --enable-compiler-warnings \
    --disable-compiler-optimisations --disable-linker-optimisations \
    --enable-compiler-coverage && make && make install
```

20.7 Upstart Objects

- `Event` represents an event.
- **`ConfSource` represents a type of configuration source (file or directory) and**

  includes `inotify` watches.
- **`ConfFile` represents the jobs "`.conf`" file *name*, but also**

  has a pointer to its contents (see below).
- `JobClass` represents the jobs "`.conf`" file *contents*.
- `Job` represents a running instance of job.
- `Session` represents a user session for user jobs, or a chroot.
- `Log` represents job log data (data that a single job process has produced on its standard output and standard error).
- **`Blocked` is used to handle `hook` and `method` events types.**

  (See hooks and methods).

See upstart-objects-diagram.

20.8 Unit Tests

Every major feature in Upstart needs to be accompanied with comprehensive unit tests. To run the tests:

```
$ autoreconf -fi
$ ./configure --enable-compiler-coverage ...
$ make check 2>&1|tee make-check.log
```

Note that as of Upstart 1.3, some of these tests cannot be run from within a chroot(2) environment unless D-Bus is installed and configured *within* the chroot. This scenario is detected, a warning about bug 728988 is logged and those tests are automatically skipped. Hence, to run *all* the tests, please ensure you run "`make check`" *outside* of a chroot(2) environment.

20.8.1 Building Within a Chroot

Some of the unit tests assume a full environment, including a controlling terminal. If you wish to build an Upstart package on a Debian or Ubuntu system, note that although the pbuilder(8) tool will work as expected, currently sbuild(1) does not provide a controlling terminal which causes tests to fail. See [19] and [20].

20.8.2 Statistics

At the time of writing, the number of Upstart tests, and tests for the NIH Utility Library used by Upstart are:

Unit Test Statistics.

Application	Test Count
Upstart unit tests	1068
Upstart user tests	80
NIH Utility Library	2863
Total	**4011**

importance of the test-suite cannot be overstated: it's one of the main "safety-nets" to ensure the behaviour of NIH and Upstart is assured.

To run the test suite for NIH or Upstart, simply run the following *as a non-privileged user*:

```
make check
```

20.8.3 Test Coverage

To check the test coverage after running the tests, look at each file using gcov(1):

```
$ cd init
$ gcov -bf event.c
```

20.9 Enable Full Compiler Warnings

If you want to start submitting changes to Upstart, you need to ensure you build it as follows to catch any warnings and errors the compiler can flag:

```
./configure --disable-silent-rules --enable-compiler-warnings --disable-compiler-optimisations --disable-linker-optimisations --enable-compiler-coverage
```

20.10 Running Upstart as a Non-Privileged User

Upstart 1.3 introduced a number of options to help with testing. The "`--session`" command-line option allows you to run Upstart as a non-privileged user since it makes Upstart connect to the D-Bus *session* bus for which each user has their own:

```
$ /sbin/init --session --debug --confdir $HOME/conf/ --no-sessions
```

This is useful since you can now try out new features, debug with GDB, *et cetera* without having to install Upstart and run it as `root`. Once you've got your second instance of Upstart running, you can then use the same option on initctl to manipulate jobs:

```
$ initctl --session emit foo
```

The caveat here is that running Upstart as a non-privileged user with a PID other than `1` changes its behaviour slightly. So, only use this technique for unit/functional testing and remember that any changes you post for inclusion should have been tested in a real scenario where Upstart is run as `root` and used to boot a system.

20.11 Useful tools for Debugging with D-Bus

If you are debugging initctl(8), you'll need to understand D-Bus. These tools are invaluable:

- dbus-send(1)
- D-Feet

20.12 Debugging a Job

There is a magic stanza called `debug` which will start the job via fork(2) and then pause it. This can be useful. Assuming you have a job "`debug.conf`" such as:

```
# XXX: magic stanza!
debug
```

```
script
  /bin/true
end script
```

You could now trace the job process like this:

```
# start debug
debug start/running, process 12345
# strace -p 12345 -o /tmp/debug.log -Ff -s 1024 -v
status debug debug stop/waiting
```

After the call to start, the job process will be "running", but paused. The *strace(1)* will resume the job and you will then have a log of what happened in file "`/tmp/debug.log`".

20.13 Debugging Another Instance of Upstart Running as root with PID 1

20.13.1 Method 1 (crazy)

Caveat Emptor: this is somewhat crazy, but if you really want to do this:

```
$ sudo \
  gdb --args \
  clone -e DBUS_SYSTEM_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
  -f CLONE_NEWPID,SIGCHLD,CLONE_PTRACE -- \
  init/init --debug --confdir /my/conf/dir --no-startup-event
    --no-sessions
```

This uses the Clone tool, which is very similar to unshare(1) but allows you to put a process into a new PID namespace.

20.13.2 Method 2 (saner)

Use a container technology such as LXC, that simplifies the access to namespaces. For example [8]:

```
$ sudo lxc-start -n natty
$ upstart_pid=$(pgrep -f /sbin/init|grep -v '^1$')
$ sudo gdb /sbin/init $upstart_pid
```

Like the example above, here we use gdb to debug Upstart running as root with PID 1, but with thanks to LXC, the container is fully isolated from the host system using namespaces. See lxc(7) for details of LXC on Ubuntu.

20.14 NIH

Grab the code from the NIH Utility Library page.

The NIH documentation is *with* the code:

- Header files provide introductory details.
- Every function, macro and variable is documented immediately above it.

References in the sections below give locations of file in the NIH source.

20.14.1 Memory Handling

Do not use `malloc()`, `calloc()`, `realloc()` or `free()` when working with Upstart. Rely instead on the NIH memory routines:

- Low-level memory allocation is handled using `nih_alloc()` and `nih_realloc()`.
- It is more normal to use `nih_new(parent, type)` though.
- To free memory, use `nih_free()`:

```
typedef struct foo {
    int i;
} Foo;

Foo *foo = nih_new (NULL, Foo);
foo->i = 123;
    /* time passes... */
nih_free (foo);
```

> **Warning**
>
> *NEVER* free memory using `nih_free()` that NIH did not allocate!

See: `nih/alloc.[ch]`

Like C++, NIH can perform automatic cleanup when objects go out of scope. The most magical part of NIH is `nih_local`.

Question: is the following code leaking memory?

```
void foo (void)
{
    nih_local char *string = nih_strdup (NULL, "hello, world");

    nih_message ("%s", string);
}
```

Answer: No!

- `nih_local` is syntactic sugar to tell the compiler that the memory that the variable it applies to ("`string`") should be freed when the last reference to it is dropped. This happens when the variable goes out of scope at the end of `foo()`.

> **Warning**
>
> *ALWAYS* assign `nih_local` variables to `NULL` to avoid memory corruption issues if the variable is not assigned for some code path!

20.14.2 The NIH Parent Pointer

Most NIH routines take a `void *parent` as their first parameter.

This parent pointer can be `NULL` as shown below:

```
nih_strdup (NULL, "hello, world");
```

If the parent is *not* `NULL`, NIH will automatically add an appropriate reference such that when the parent is freed, so are its child objects.

Consider this example:

```
void bar (void)
{
    typedef struct thing {
        char *str;
    } Thing;

    nih_local Thing *thing = nih_new (NULL, Thing);

    /* XXX: note that we specify the parent as 'thing' */
    thing->str = nih_strdup (thing, "first string");
}
```

Two memory allocations have been performed:

- `thing`
- `thing->str`

And yet when `bar()` exits, there is *no* leak because NIH knows that `thing->str` is a "child" of `thing` and will *do-the-right-thing (TM)* and free both chunks of memory!

Here is another subtle example:

```
void bar (void)
{
    typedef struct thing {
        char *str;
    } Thing;

    nih_local Thing *thing = nih_new (NULL, Thing);

    /* XXX: note that we specify the parent as 'thing' */
    thing->str = nih_strdup (thing, "a string value");

    /* now, let's reassign the pointer */
    thing->str = nih_strdup (thing, "another string value");
}
```

Surely, there must be a leak *now* since we've re-assigned `thing->str`?

In fact, *there is no leak* because both the strings that we've assigned to `thing->str` have specified the *same* parent: `thing`. So, the reference to `a string value` has not been lost and both string values will be freed correctly when `thing` goes out of scope!

20.14.3 `nih_free()`

However, sometimes using `nih_local` is not appropriate. In the example below, we manually free the memory using `nih_free()`:

```
void bar (void)
{
    typedef struct thing {
        char *str;
    } Thing;

    Thing *thing = nih_new (NULL, Thing);

    /* XXX: note that we specify the parent as 'foo' */
    thing->str = nih_strdup (thing, "first string");

    /* "manually" free thing _and_ thing->str */
    nih_free (thing);
}
```

Here we use `nih_free()` to force NIH to free up memory.

> **Warning**
>
> *NEVER* call `nih_free()` on an `nih_local` variable!

20.14.4 `NIH_MUST()`

The example so far have not checked for error conditions. Here's how we *could* handle an out-of-memory scenario:

```
nih_local char *string = NULL;

string = nih_strdup (thing, "first string");

if (! string) {
  /* handle the error */
}
```

However, this tends to lead to code littered with error checking. There is a common NIH idiom that avoids such problems:

```
nih_local char *string = NULL;

string = NIH_MUST (nih_strdup (thing, "first string"));

  /* string is now guaranteed to have the expected error */
}
```

See file: `nih/macros.h`

`NIH_MUST()` will evaluate its argument until it returns a value.

> **Warning**
>
> `NIH_MUST()` will try *forever* to grab the memory required.
>
> That *could* lead to Upstart going into a tight loop and effectively killing your machine.

However, realistically, Upstart only ever allocates small chunks of memory and if `/sbin/init`, running as `root` is unable to allocate a few bytes of memory, you machine has big problems.

20.14.5 Error Handling

If a function detects a failure, it must return a suitable error value. However, it may be appropriate to raise an exception. You'll know if a function raises an exception since it will be documented like this:

```
Returns: zero on success, negative value on raised error.
```

A "raised error" refers to an `NihError` object being raised when the function detects an error.

Therefore, it is the callers responsibility to:

- Check the return code of every function that returns a value.
- Handle raised errors appropriately (and immediately!)

See: `nih/error.[ch]`

Let's look at an example:

```
char *
num_to_str (int i)
{
    if (i % 2)
        return NIH_MUST (nih_sprintf (NULL, "%d", i));

    nih_error_raise_no_memory ();
    return NULL;
}

int
main (int argc, char *argv[])
{
    nih_local char *s = NULL;

    s = num_to_str (1);
    nih_message ("got: '%s'", s);

    /* force error scenario */
    s = num_to_str (2);
    if (! s) {
        /* retrieve the error */
        err = nih_error_get ();
```

```
        /* display it */
        nih_message ("%s:%d:%s:%d:%s",
                err->filename,
                err->line,
                err->function,
                err->number,
                err->message);
        /* clear the error */
        nih_free (err);
    }

    exit (EXIT_SUCCESS);
}
```

20.14.5.1 Impact of Ignoring a Raised Error

An example of code that ignores a raised error:

```
char *
num_to_str (int i)
{
    if (i % 2)
        return NIH_MUST (nih_sprintf (NULL, "%d", i));

    nih_error_raise_no_memory ();
    return NULL;
}

int
main (int argc, char *argv[])
{
    nih_local char *s = NULL;

    /* ok */
    s = num_to_str (1);
    nih_message ("got: '%s'", s);

    /* force error scenario */
    s = num_to_str (2);

    /* Oops - forgot to check return! */
    nih_message ("got: '%s'", s);

    exit (EXIT_SUCCESS);
}
```

Output:

```
got: '1'
got: '(null)'
(null):test_nih_error.c:38: Unhandled error from num_to_str: Cannot
allocate memory
[1]    20476 abort (core dumped)  bin/test_nih_error
```

The reason this crashes is that NIH installs an atexit(3) handler which checks for any `NihError` errors that have not been handedled on exit.

Of course, in the case of Upstart, *it* never exits so failing to handle an error will result in an assertion failure the *next* time an error object is raised.

- To raise an exception when `ERRNO` gets set, use:
  - `nih_error_raise_system()`
  - `nih_return_system_error()`
- To raise an arbitrary exception, use:
  - `nih_error_raise(number, message)`
  - `nih_error_raise_printf(number, format, ...)`
  - `nih_return_error(retval)`

See file: `nih/error.h`

20.14.6 Output

NIH has a rich set of output routines:

- `nih_debug()`
- `nih_info()`
- `nih_message()`
- `nih_warn()`
- `nih_fatal()`
- `nih_fatal()`

All routines take a format string and arguments like printf(3):

```
int   i = 123;
char *s = "hello, world";
nih_debug ("s='%s', i=%d", s, i);
```

Like syslog(3), NIH will only display message made with the above calls if the log priority is appropriate.

To change the priority, use `--verbose`, `--debug`, or programatically call `nih_set_priority()`.

By default, output goes to *standard output*, but early in its initialisation, it redirects output to the kernel ring buffer using:

```
nih_log_set_logger (logger_kmsg);
```

See file: `nih/logging.[ch]`

20.15 Creating a New Object

20.15.1 Template for a new "foo"

```
/**
 * foo:
 * @entry: list header,
 * @name: name of foo,
 * @value: value of foo.
 *
 * Structure to hold a foo.
 * << XXX: more details here >>.
 **/
 typedef struct foo {
     NihList   entry;
     char     *name;
     int       value;
 } Foo;

 /**
  * foos:
  * List of all foos. << XXX: more details here >>
  **/
 NihList *foos;

 /**
  * Initilise the foos list.
  */
 void foo_init (void)
 {
     if (! foos)
         foos = NIH_MUST (nih_list_new (NULL));
 }

 Foo * foo_new (void *parent, const char *name, int value)
         __attribute__ ((warn_unused_result, malloc));

 /**
  * foo_new:
  * @parent: parent of new foo,
  * @name: name of foo,
  * @value: value of foo.
  *
  * Returns: Newly allocated foo, or NULL on insufficient memory.
  **/
  Foo *
  foo_new (void *parent, const char *name, int value)
  {
      Foo *foo;

      assert (name); /* check all args possible */
      foo_init (); /* initialise the subsystem */
```

```
    /* create the object */
    foo = NIH_MUST (nih_new (parent, Foo));

    /* initialise the embedded list */
    nih_list_init (&foo->entry);

    /* save values */
    foo->name = NIH_MUST (nih_strdup (foo, name));
    foo->value = value;

    /* Add object to list of known foos */
    nih_list_add (foos, &source->entry);

    /* explain how objects should be disposed of */
    nih_alloc_set_destructor (foo, nih_list_destroy);

    return foo;

error:
    nih_free (foo);
    return NULL;
}
```

20.15.2 Basic Test Example for a New "foo"

```
Foo  *foo;
char *str;

TEST_FEATURE ("with parent");

foo_init ();

TEST_LIST_EMPTY (foos);

str = nih_strdup (NULL, "hello");
TEST_NE_P (str, NULL);

foo = foo_new (str, "foo", 123);
TEST_NE_P (foo, NULL);

TEST_ALLOC_PARENT (foo, str);
TEST_ALLOC_SIZE (foo, sizeof (Foo));

TEST_FREE_TAG (foo->name);
TEST_LIST_NOT_EMPTY (foos);

TEST_EQ (foo->value, 123);
TEST_EQ_STR (foo->name, "foo");
TEST_ALLOC_PARENT (foo->name, foo);

nih_free (foo);
TEST_LIST_EMPTY (foos);
TEST_FREE (foo->name);

nih_free (str);
```

20.16 Adding a new `initctl` command

20.16.1 Adding a New non-Job Command

1. Add a new function called "`<name>_action()`" to `util/initctl.c` where "<name>" is the name of the new command the user will type on the command-line ("`initctl <name>`") with all hyphens ("`-`") converted to underscores ("`_`").

   Example: "`reload_configuration_action()`" for the "`reload-configuration`" command-line command.

2. Make "`<name>_action()`" call "`upstart_<name>_sync()`", which will be an auto-generated function (see below).

   Example: "`reload_configuration_action()`" calls "`upstart_reload_configuration_sync()`".

3. Add a new D-Bus method corresponding to "<name>" *in "camel-case"* to:

   ```
   dbus/com.ubuntu.Upstart.xml
   ```

   Example: Add the following for the "`reload-configuration`" command:

```
<method name="ReloadConfiguration">
</method>
```

4. Add implementation to "`init/control.c`" as "`control_<name>()`".

   Example: add "`control_reload_configuration()`".

20.16.2 Adding a New Job Class Command

Process is as per Adding a new non-Job Command, but rather than modifying file "`dbus/com.ubuntu.Upstart.xml`", you must modify file:

```
dbus/com.ubuntu.Upstart.Job.xml
```

... and then add a function to "`init/job_class.c`".

20.16.3 Adding a New Job Command

Process is as per Adding a new non-Job Command, but rather than modifying file "`dbus/com.ubuntu.Upstart.xml`", you must modify file:

```
dbus/com.ubuntu.Upstart.Instance.xml
```

... and then add a function to "`init/job.c`".

20.16.4 Generating the D-Bus Bindings

After following the steps above to add a new `initctl` command, run "`make`" and observe the the `nih-dbus-tool` utility gets calls to convert your XML definitions into auto-generated code:

```
/usr/bin/nih-dbus-tool \
                --package=upstart \
                --mode=object --prefix=control \
                --default-interface=com.ubuntu.Upstart0_6 \
                --output=com.ubuntu.Upstart.c
../dbus/com.ubuntu.Upstart.xml
/usr/bin/nih-dbus-tool \
                --package=upstart \
                --mode=object --prefix=job_class \
                --default-interface=com.ubuntu.Upstart0_6.Job \
                --output=com.ubuntu.Upstart.Job.c
../dbus/com.ubuntu.Upstart.Job.xml
/usr/bin/nih-dbus-tool \
                --package=upstart \
                --mode=object --prefix=job \
                --default-interface=com.ubuntu.Upstart0_6.Instance \
                --output=com.ubuntu.Upstart.Instance.c
../dbus/com.ubuntu.Upstart.Instance.xml
```

20.17 `TEST_ALLOC_FAIL`

NIH provides a rather clever macro called `TEST_ALLOC_FAILED`; it accepts a code block and will execute that block 1 + *N* times where *N* is the number of NIH memory allocation calls made within the block.

- The first time through, the macro counts the number of NIH allocation calls.
- Each subsequent time through, it causes the *N*th call to an NIH memory allocation routine to fail.

This exercises fully for example a function which returns a newly-allocated object (and which may make any number of calls to the NIH memory allocation routines).

Essentially, it ensures your handling of memory allocation failures are correct.

20.17.1 Improved Test Example for a New "foo" (with a bug)

We can now modify our previous example to also use `TEST_ALLOC_FAIL`. Note that this version contains a bug! Can you spot it?:

```
Foo  *foo;
char *str;

TEST_FEATURE ("put text here");

foo_init ();

TEST_ALLOC_FAIL {
    TEST_LIST_EMPTY (foos);

    str = nih_strdup (NULL, "hello");
    TEST_NE_P (str, NULL);

    foo = foo_new (str, "foo", 123);
    if (test_alloc_failed) {
        TEST_EQ_P (foo, NULL);
        continue;
    }

    TEST_LIST_NOT_EMPTY (foos);

    TEST_ALLOC_SIZE (foo, sizeof (Foo));
    TEST_EQ (foo->value, 123);
    TEST_EQ_STR (foo->name, "foo");

    nih_free (str);
}
```

20.18 TEST_ALLOC_SAFE

If you need to guarantee that particular memory allocations within the *do not* fail, wrap those in a call to `TEST_ALLOC_SAFE`:

```
TEST_ALLOC_FAIL {
    TEST_ALLOC_SAFE {
        /* Memory allocations will work here */
    }

    /* Memory allocations will be sequentially FAILED here */
}
```

20.18.1 Final Test Example for a New "foo"

Using `TEST_ALLOC_FAIL`, we can now fix the example to be:

```
Foo  *foo;
char *str;

TEST_FEATURE ("put text here");

foo_init ();

TEST_ALLOC_FAIL {
    TEST_ALLOC_SAFE {
        TEST_LIST_EMPTY (foos);

        str = nih_strdup (NULL, "hello");
        TEST_NE_P (str, NULL);
    }

    foo = foo_new (str, "foo", 123);
    if (test_alloc_failed) {
        TEST_EQ_P (foo, NULL);
        continue;
    }

    TEST_LIST_NOT_EMPTY (foos);

    TEST_ALLOC_SIZE (foo, sizeof (Foo));
    TEST_EQ (foo->value, 123);
    TEST_EQ_STR (foo->name, "foo");

    nih_free (str);
}
```

20.19 Basic Debugging

Don't underestimate the usefulness of two very simple techniques:

- `sudo strace -p 1 -fFv -s 1024`
- `nih_fatal("\%s:\%d", __func__, __LINE__);`

20.20 Debugging Upstart as a Non-Privileged User

With the right command-line options, it's possible to run Upstart as a normal non-privileged user:

```
$ make
$ mkdir /tmp/conf /tmp/log
$ cp *.conf /tmp/conf
$ gdb init/init --confdir /tmp/conf --logdir /tmp/log --no-sessions --session --debug
```

This is a useful technique but be aware that the behaviour of Upstart running as a non-privileged user is slightly different to running it as `root` with PID 1.

20.21 Debugging Upstart as `root`

It *is* in fact possible to debug `/sbin/init` using `gdb` as user `root` on a running system!

- Build upstart with `-ggdb3` and install to `/sbin/init.foo` for example.
- `sudo gdb /sbin/init.foo 1`

20.22 Debug Tip Using Destructors

If can be useful to register a custom destructor for your object as a debug aid:

```
int foo_destructor(void *ignored)
{
    /* Do something */
    return 1;
}

Foo *
foo_new (void *parent)
{
    Foo *foo = NIH_MUST (nih_new (parent, Foo));

    /* ... */

    /* Call foo_destructor when object is destroyed */
    nih_alloc_set_destructor (foo, foo_destructor);

    return foo;
}
```

Now, whenever a `Foo` is freed, `foo_destructor()` will be called.

Note that child objects of the `Foo` object that `foo_destructor()` is being called for and the parent referneces and the object itself *will* be freed - the destructor is for very specialist operations, such as degugging.

20.22.1 Lists

Here's an example of using NIH lists:

```
typedef struct bar {
    NihList  entry;
    char    *str;
} Bar;

int
main (int argc, char *argv[])
{
    int i;
    nih_local NihList *args = NULL;

    args = NIH_MUST (nih_list_new (NULL));

    /* store all arguments in a list */
```

```
    for (i = 1; i < argc; ++i) {
        Bar *bar = NIH_MUST (nih_new (args, Bar));
        nih_list_init (&bar->entry);
        bar->str = NIH_MUST (nih_strdup (bar, argv[i]));
        nih_list_add (args, &bar->entry);
    }
    i = 1;

    /* display all arguments by iterating over list */
    NIH_LIST_FOREACH (args, iter) {
        Bar *bar = (Bar *)iter;
        nih_message ("argument %d='%s'", i, bar->str);
        ++i;
    }

    return (0);
}
```

- NIH lists are designed to be *embedded* within some other structure.
- Create a list dynamically using `nih_list_new()`.
- Initialize a static list using `nih_list_init()`.
- Add one list to another using `nih_list_add()`.
- Iterate a list using `NIH_LIST_FOREACH()`.

See file: `nih/list.[ch]`

20.22.1.1 Removing Elements from a List

An example showing how to remove an element from a list:

```
NihList      *entry_list;
NihListEntry *entry;

entry_list = NIH_MUST (nih_list_new (NULL));

entry = NIH_MUST (nih_list_entry_new (entry_list));
entry->str = NIH_MUST (nih_strdup (entry, "hello"));
nih_list_add (entry_list, &entry->entry);

entry = NIH_MUST (nih_list_entry_new (entry_list));
entry->str = NIH_MUST (nih_strdup (entry, "world"));
nih_list_add (entry_list, &entry->entry);

entry = (NihListEntry *)nih_list_remove (entry_list);
nih_free (entry_list);
```

Freeing `entry_list` frees the "`hello`" *and* the "`world`" entries since although the "``world" entry was removed from its containing list, we did *NOT* break the reference between that entry and its parent (`entry_list`).

If we had wanted to break the reference, we could have used `nih_ref()` and `nih_unref()` to:

- Add a reference for this entry to a new parent.

- Remove the existing reference between `entry_list` and the entry.

Another method for removing an entry from a list is whilst iterating it:

```
NIH_LIST_FOREACH_SAFE (entry_list, iter) {
    NihListEntry *entry = (NihListEntry *)iter;
    nih_free (entry);
}
```

- Note that we are now using . Do *NOT* attempt to remove a list entry whilst iterating a list using .
- It is *NOT* allowed to iterate a list *whilst it is already being iterated*. Therefore, you need to be *very* careful that your function is not being called from within a foreach-loop.

20.22.1.2 Moving an Element Between Lists

An example showing moving an element from one list to another:

```
NihList      *list1;
NihList      *list2;
NihListEntry *entry;

list1 = NIH_MUST (nih_list_new (NULL));
list2 = NIH_MUST (nih_list_new (NULL));

/* Create entry and add to list1 */
entry = NIH_MUST (nih_list_entry_new (list1));
nih_list_add (list1, &entry->entry);

/* Fully move entry to list2 */
nih_list_add (list2, &entry->entry);
nih_ref (entry, list2);
nih_unref (entry, list1);

/* Frees list1, but not entry */
nih_free (list1);

/* Frees list2 AND entry */
nih_free (list2);
```

20.22.2 Hashes

NIH Hashes are actually "hashed lists" (essentially arrays of lists):

```
NihHash *
nih_hash_new (const void      *parent,
              size_t           entries,
              NihKeyFunction   key_function,
              NihHashFunction  hash_function,
              NihCmpFunction   cmp_function);
```

However, the more common way to create a hash is via:

```
typedef struct foo {
    NihList  entry;
    char    *name;
} Foo;

/**
 * foos:
 * List of all foos. << XXX: more details here >>
 **/
NihHash *foos;

/**
 * Initilise the foos hash.
 */
void foo_init (void)
{
    if (! foos)
        foos = NIH_MUST (nih_hash_string_new (NULL, 0));
}

Foo *
foo_new (void *parent, const char *name)
{
    Foo *foo;

    assert (name);
    foo_init (); /* initialise the subsystem */

    /* create the object */
    foo = NIH_MUST (nih_new (parent, Foo));

    /* initialise the embedded _list_ */
    nih_list_init (&foo->entry);

    nih_hash_add (foos, &foo->entry);

    return foo;
}
```

20.22.2.1 Using Hashes

To iterate a hash, use `NIH_HASH_FOREACH()`:

```
NIH_HASH_FOREACH (foos, iter) {
    Foo *foo = (Foo *)iter;

    /* do something with foo */
}
```

To find an entry in a hash, use `nih_hash_lookup()`:

```
Foo *foo;
```

```
foo = (Foo *)nih_hash_lookup (foos, "hello");
if (foo) {
    /* ... */
}
```

Alternatively, if there are *multiple* entries for a particular "hash bucket", use `nih_hash_search()`.

See: `nih/hash.[ch]`

20.22.2.2 nih_hash_string_new()

nih_hash_string_new()``is "magic" *BUT* to use it *the first structure element **after** the element **must** be a "``char *" that will uniquely represent that hash entry*.

If a simple string is not sufficient for your purposes, you will need to use `nih_hash_new()` and will also have to specify the `NihKeyFunction`, `NihHashFunction` and `NihCmpFunction`.

Analogous to `NIH_LIST_FOREACH_SAFE`, there is also a `NIH_HASH_FOREACH_SAFE` facility for removing hash entries whilst iterating the hash.

20.22.3 Trees

A basic example of NIH trees:

```
typedef struct foo {
    NihTree node;
    int     value;
} Foo;

NihTree *tree;
Foo     *foo;

tree = NIH_MUST (nih_tree_new (NULL));
foo = NIH_MUST (nih_new (tree, Foo));
nih_tree_init (&foo->node);
foo->value = 123;

nih_tree_add (tree, &foo->entry, NIH_TREE_LEFT);
```

To iterate a tree:

- `NIH_TREE_FOREACH()` (in-order traversal)
- `NIH_TREE_FOREACH_PRE()` (pre-order traversal)
- `NIH_TREE_FOREACH_POST()` (post-order traversal)

See: `nih/tree.[ch]`

Example of iterating a tree using in-order traversal:

```
NIH_TREE_FOREACH (tree, iter) {
    Foo *foo = (Foo *)iter;
    /* ... */
}
```

20.22.4 Avoiding Problems

What's wrong with this code?:

```
/* XXX: this code is incorrect! */
void foo (const char *string)
{
    nih_local char *str;
    nih_assert (string);

    if (! strcmp ("foo", string)) {
        str = NIH_MUST (nih_strdup (NULL, "bar"));
        bar (str);
    }
}
```

The problem here is that `str` is not always assigned a value, so if `string` is not `foo`, the results of this function are undefined - it could result in a crash!!

The example below contains two memory leaks:

```
NihList      *entry_list;
NihListEntry *entry;

entry_list = NIH_MUST (nih_list_new (NULL));

entry = NIH_MUST (nih_list_entry_new (NULL));
entry->str = NIH_MUST (nih_strdup (NULL, "hello"));
nih_list_add (entry_list, &entry->entry);

nih_free (entry_list);
```

- `entry` is *not* freed. To resolve, either:
    - Make its parent pointer non-`NULL` (recommended).
    - Call `nih_free()`.
- `entry->str` is *not* freed. To resolve, either:
    - Set its parent pointer to `entry` (recommended).
    - Call `nih_free (entry->str)`.

20.23 Debugger Magic

Debugging in gdb initially seems rather difficult, but you just need to know the right tricks. The complication comes from the fact that Upstart uses the NIH Utility Library, which uses macros (such as `NIH_LIST_FOREACH` and `NIH_HASH_FOREACH`) for performance.

However, how do you access a data structure such as an NihList whose only method of iteration is a macro? Like this:

20.23.1 *NihList*

```
# first entry
(gdb) print *(JobClass *)job_classes->next

# 2nd entry
(gdb) print *(JobClass *)job_classes->next->next

# 3rd entry
(gdb) print *(JobClass *)job_classes->next->next->next

# ConfSource NihWatch for 1st entry in conf_sources list
(gdb) print *((ConfSource *)conf_sources->next)->watch
```

20.23.2 *NihHash*

```
# size of JobClass->instances hash list
# XXX: this is the capacity, *NOT* the number of entries!
print class->instances->size

# first entry in job_classes global hash
print *(JobClass *)job_classes->bins->next
```

20.23.3 *nih_iterators*

Alternatively, you can make use of the "unofficial" NIH Iterators which provide functional versions of the standard NIH macros and a few extras. Note that these are *ONLY* for testing and debugging!

- `nih_list_foreach()`:

```
/**
 * nih_list_foreach:
 *
 * @list: list,
 * @len: optional output parameter that will contain length of list,
 * @handler: optional function called for each list entry,
 * @data: optional data to pass to handler along with list entry.
 *
 * Iterate over specified list.
 *
 * One of @len or @handler may be NULL.
 * If @handler is NULL, list length will still be returned in @len.
 * If @handler returns 1, @len will be set to the number of list entries
 * processed successfully up to that point.
 *
 * Returns: 0 on success, or -1 if handler returns an error.
 **/
int
nih_list_foreach (const NihList *list, size_t *len, NihListHandler handler, void *data);
```

  - `nih_hash_foreach()`:

```
/**
 * nih_hash_foreach:
 *
 * @hash: hash,
```

```
 * @len: optional output parameter that will contain count of hash entries,
 * @handler: optional function called for each hash entry,
 * @data: optional data to pass to handler along with hash entry.
 *
 * Iterate over specified hash.
 *
 * One of @len or @handler may be NULL.
 * If @handler is NULL, count of hash entries will still be returned in @len.
 * If @handler returns 1, @len will be set to the number of hash entries
 * processed successfully up to that point.
 *
 * Returns: 0 on success, or -1 if handler returns an error.
 **/
int
nih_hash_foreach (const NihHash *hash, size_t *len,
            NihListHandler handler, void *data);
```

- nih_tree_foreach():

```
/**
 * nih_tree_foreach:
 *
 * @tree: tree,
 * @len: optional output parameter that will contain count of tree nodes,
 * @handler: optional function called for each tree node,
 * @data: optional data to pass to handler along with tree node.
 *
 * Iterate over specified tree.
 *
 * One of @len or @handler may be NULL.
 * If @handler is NULL and @len is non-NULL, count of tree nodes will
 * still be returned in @len.
 * If @handler returns 1, @len will be set to the number of tree nodes
 * processed successfully up to that point.
 *
 * Returns: 0 on success, or -1 if handler returns an error.
 **/
int
nih_tree_foreach (NihTree *tree, size_t *len,
            NihTreeFilter handler, void *data);
```

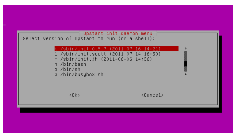
These routines allow us to also provide trivial implementations of the following convenience functions:

- nih_list_count()
- nih_hash_count()
- nih_tree_count()

20.24 Development Utilities

20.24.1 upstart_menu.sh

The upstart-menu utility allows /sbin/init versions you wish to boot with to be selected using a friendly menu. You can also select a shell. upstart_menu.sh scans /sbin/ for init version and presents a list, most recently modified version first:



Main screen of `upstart_menu.sh`.

The utility also allows you to specify options (it automatically shows you a list of available options for the version of the program you have selected):

`upstart_menu.sh` showing the options screen.

20.24.1.1 Enabling `upstart_menu.sh`

To enable `upstart_menu.sh`:

1. Copy file to `/sbin/upstart_menu.sh`.
2. Make the file executable.
3. Update `/etc/default/grub` such that `GRUB_CMDLINE_LINUX` is modified to:
    - Remove "`quiet`" and "`splash`".
    - Add "`init=/sbin/upstart_menu.sh`".
4. Update grub: "`sudo update-grub`".
5. Reboot!

20.25 Gotchas

- Passing `NULL` to `nih_free()`: unlike `free(3)`, `nih_free()` **does not** allow a `NULL` parameter.
- Running `make check` as `root` (tests *will* fail).
- Debugging a failing memory-checking test by littering test code with calls to `nih_debug()`... which calls `nih_alloc()`.
- Forgetting to install either `/sbin/init` or `/sbin/initctl` when you modify the D-Bus interface to Upstart (if you're lucky, you'll get a crash, else very odd behaviour! :-)
- Not checking for existing `init` and `test_*` processes still running from a previous failed test run when you run `make check`.

21 Known Issues

21.1 Restarting Jobs with Complex Conditions

The `and` and `or` operators allowed with `start on` and `stop on` do not work intuitively: operands to the right of either operator are only evaluated when the specified event is emitted. This can lead to jobs with complex `start on` or `stop on` conditions not behaving as expected when restarted. For example, if a job specifies the following condition:

```
start on A and (B or C)
```

When the events "A" and "B" are emitted, the condition is satisfied so the job will be run. If the job fails to start, or is stopped later, there is no guarantee that "A" will be emitted again, and the fact that it happened before **is no longer known to Upstart**. Meanwhile, events "C" or "B" may occur, but the job will *not* be transitioned back to a start goal, until event "A" is emitted again.

21.1.1 Advice

To minimise the risk of being affected by this issue, avoid using complex conditions with jobs which need to be restarted.

21.2 Using `expect` with `script` sections

Using the expect stanza with a job that uses a `script` section will lead to trouble if your script spawns any processes (likely!). Consider:

```
expect fork
respawn
script
  ARGS=$(cat /etc/default/grub)
  exec echo "ARGS=$ARGS" > /tmp/myjob.log
end script
```

This job configuration file is somewhat nonsensical, but it does demonstrate the problem. The main issue here is that by specifying expect fork, Upstart will attempt to follow *only* the **first** fork(2) call. The first process that this job will spawn is... cat(1), *NOT* `echo`. As such, starting the job will show something like this:

```
# start myjob
myjob start/running, process 12345
# status myjob
myjob start/running, process 12345
# ps --no-headers -p 12345
# kill 12345
-su: kill: (12345) - No such process
```

As the ps(1) call shows, the (`cat`) process is no longer running, but Upstart thinks it is.

Unfortunately, since Upstart will wait forever until it is able to stop the pid (which no longer exits). A manual attempt to either "`stop myjob`" or "`start myjob`" will also hang.

The only solution to clear this "stuck job" is to reboot. See [18] and Recovery on Misspecification of expect. Note that this "zombie job" isn't actually causing any problems for Upstart, but it is annoying and potentially confusing seeing it listed in initctl output. It will of course also be consuming a very small amount of memory.

Note however, that if you are working on a development system (hopefully you *are* whilst developing your job configuration file!), what you *can* do to keep working is to copy the problematic job configuration file to a new name, ignore the old job entirely and keep working using the new job!

21.3 Bugs

Upstart bugs

https://bugs.launchpad.net/upstart

Ubuntu-specific Upstart bugs

https://launchpad.net/ubuntu/+source/upstart/+bugs

22 Support

The primary sources of support are:

- The IRC Channel `#upstart` on IRC server `freenode.net`.

  If you don't get a response, consider posting to the Mailing List.
- The Mailing List

  If you don't get a response, consider raising a bug. See Coverage to determine how to report bugs and ask questions.

23 References

23.1 Manual Pages

man 5 init
: Configuration syntax reference.

man 8 init
: Options for running the Upstart init daemon.

man 8 initctl
: Explanation of the Upstart control command.

man 7 upstart-events
: Comprehensive summary of all "well-known" Upstart system events on Ubuntu.

23.2 Web Sites

http://upstart.ubuntu.com/
: Main Ubuntu page for Upstart.

http://launchpad.net/upstart
: The main Upstart Bazaar project page.

http://upstart.at
: The New Upstart Blog site.

http://netsplit.com/category/tech/upstart/
: Scotts Original Upstart blog with useful overviews of features and Concepts.

https://wiki.ubuntu.com/ReplacementInit
: Original Specification.

23.3 Mailing List

- https://lists.ubuntu.com/mailman/listinfo/upstart-devel

24 Answers to Test

25 Footnotes

26 Colophon

Document generated from reStructuredText plaintext markup source on 2013-01-25 at 08:12:20 from Bazaar branch branch. See Upstart Documenters and Upstart Cookbook. (revision $Revision-Id).

Organization: Canonical Ltd.

Status: Drafting

27 Appendices

27.1 Ubuntu Well-Known Events ( )

The information in this section is taken from the upstart-events(7) manual page.

```
Name

   upstart-events — Well-known Upstart events summary

Event Summary

   This manual page summarizes well-known events generated by the Upstart init(8) daemon. It is
   not an exhaustive list of all possible events, but rather details a standard set of events
   expected to be generated on any Ubuntu system running Upstart.

   The primary table, Table 1, encodes the well-known events, along with the type of each event
   (listed in Table 2), the emitter of the event (see Table 3) and the approximate time at which
   the event could be generated. Additionally, the Note column indexes into Table 4 for further
   details on a particular event.

   The Ref (Reference) column is used to refer to individual events succinctly in the Time
   column.

   Note that the '<' and '>' characters in the Time column denote that the event in the Event
   column occurs respectively before or after the event specified in the Time column (for
   example, the mounting(7) event occurs "at some time" after the startup(7) event, and the
   virtual-filesystems(7) event occurs after the last mounted(7) event relating to a virtual
   filesystem has been emitted).

   For further details on events, consult the manual pages and the job configuration files,
   usually located in /etc/init.

   Table 1. Table 1: Well-Known Event Summary.

   +-----------------------------------------------------------------------------------------+
   | Ref |            Event              | Type | Emit |             Time            | Note |
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | all-swaps                     |  S   |  M   | > (5)                       |      |
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | control-alt-delete(7)         |  S   |  A   | > (5)                       |  A   |
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | container                     |  S   |  C   | > /run mounted              |  Q   |
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | dbus-activation               |  S   |  B   | > D-Bus client request      |      |
```

```
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | deconfiguring-networking      |  H   |  V   | < non-local IFs down        |  P   |
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | desktop-session-start         |  H   |  D   | > X(7) session created      |  B   |
   |-----+-------------------------------+------+------+-----------------------------+------|
   |     | desktop-shutdown              |  H   |  D   | > X(7) session ended        |  O   |
```

```
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | device-not-ready              |   H   |   M   | > (2)                       |   N   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | drm-device-added              |   S   |   U   | > (5)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | failsafe-boot                 |   S   |   X   | > (7) and local IF          |   S   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  7  | filesystem                    |   S   |   M   | After last (1)              |   D   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | graphics-device-added         |   S   |   U   | > (5)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | keyboard-request(7)           |   S   |   A   | > (5)                       |   E   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | local-filesystems(7)          |   S   |   M   | > (6)                       |       |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | login-session-start           |   H   |   D   | < DM running                |   F   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  1  | mounted(7)                    |   H   |   M   | > associated (2)            |   G   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  2  | mounting(7)                   |   H   |   M   | > (5)                       |   H   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  3  | net-device-added              |   S   |   U   | > (5)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | net-device-changed            |   S   |   U   | > (5)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | net-device-down               |   S   |   F   | < (4)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  4  | net-device-removed            |   S   |   U   | > (5)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | net-device-up                 |   S   |  F,N  | > (3)                       |   C   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | not-container                 |   S   |   C   | > /run mounted              |   Q   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | power-status-changed(7)       |   S   |   I   | > (5)                       |   I   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | recovery                      |   S   |   G   | Boot (<5)                   |   R   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | remote-filesystems(7)         |   S   |   M   | > (6)                       |       |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | runlevel(7)                   |   M   |   T   | > (7) + (8)                 |       |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | socket(7)                     |   S   |   S   | > socket connection         |       |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  5  | startup(7)                    |   S   |   I   | Boot                        |   J   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | started(7)                    |   S   |   I   | > job started               |   K   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | starting(7)                   |   H   |   I   | < job starts                |   K   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  8  | static-network-up             |   S   |   N   | > last static IF up         |       |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | stopped(7)                    |   S   |   I   | > job stopped               |   K   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | stopping(7)                   |   H   |   I   | < job stops                 |   K   |
```

```
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|     | unmounted-remote-filesystems  |   H   |   V   | > last remote FS unmounted  |   L   |
|-----+-------------------------------+-------+-------+-----------------------------+-------|
|  6  | virtual-filesystems(7)        |   S   |   M   | > last virtual FS (1)       |   M   |
+-------------------------------------------------------------------------------------------+

Key: 'DM' is an abbreviation for Display Manager. 'FS' is an abbreviation for filesystem. 'IF'
```

```
is an abbreviation for Network Interface.

Table 2. Table 2: Event Types.

+------------------------------------------------------------------------------------+
| Ref | Event Type | Notes                                                             |
|-----+------------+-----------------------------------------------------------------|
|  H  | Hook       | Blocking. Waits for events that start on or stop on this event. |
|-----+------------+-----------------------------------------------------------------|
|  M  | Method     | Blocking task.                                                  |
|-----+------------+-----------------------------------------------------------------|
|  S  | Signal     | Non-blocking.                                                   |
+------------------------------------------------------------------------------------+

Table 3. Table 3: Event Emitters.

+--------------------------------------------------------------------------+
| Ref | Emitter                          | Notes                           |
|-----+----------------------------------+---------------------------------|
|  A  | System Administrator (initiator) | Technically emitted by init(8). |
|-----+----------------------------------+---------------------------------|
|  B  | dbus-daemon(1)                   | Run with "--activation=upstart" |
|-----+----------------------------------+---------------------------------|
|  C  | container-detect job             |                                 |
|-----+----------------------------------+---------------------------------|
|  D  | Display Manager                  | e.g. lightdm/gdm/kdm/xdm.       |
|-----+----------------------------------+---------------------------------|
|  F  | ifup(8) or ifdown(8)             | See /etc/network/.              |
|-----+----------------------------------+---------------------------------|
|  G  | bootloader or initramfs          |                                 |
|-----+----------------------------------+---------------------------------|
|  I  | init(8)                          |                                 |
|-----+----------------------------------+---------------------------------|
|  M  | mountall(8)                      |                                 |
|-----+----------------------------------+---------------------------------|
|  N  | network-interface job            |                                 |
|-----+----------------------------------+---------------------------------|
|  S  | upstart-socket-bridge(8)         |                                 |
|-----+----------------------------------+---------------------------------|
|  T  | telinit(8), shutdown(8)          |                                 |
|-----+----------------------------------+---------------------------------|
|  U  | upstart-udev-bridge(8)           |                                 |
|-----+----------------------------------+---------------------------------|
|  V  | System V init system             |                                 |
|-----+----------------------------------+---------------------------------|
|  X  | failsafe job                     |                                 |
+--------------------------------------------------------------------------+

Table 4. Table 4: Event Summary Notes.

+--------------------------------------------------------------------------------------------+
| Note | Detail                                                                              |
|------+-------------------------------------------------------------------------------------|
|  A   | Requires administrator to press Control-Alt-Delete key combination on the console.  |
```

```
|------+-------------------------------------------------------------------------------------|
|  B   | Event generated when user performs graphical login.                                 |
|------+-------------------------------------------------------------------------------------|
|      | These are specific examples. upstart-udev-bridge(8) will emit events which match    |
|  C   | the pattern, "S-device-A" where 'S' is the udev subsystem and 'A' is the udev       |
|      | action. See udev(7) and for further details. If you have sysfs mounted, you can     |
|      | look in /sys/class/ for possible values for subsystem.                              |
|------+-------------------------------------------------------------------------------------|
|  D   | Note this is in the singular - there is no 'filesystems' event.                     |
|------+-------------------------------------------------------------------------------------|
```

```
|  E   | Emitted when administrator presses Alt-UpArrow key combination on the console.      |
|------+---------------------------------------------------------------------------------------|
|  F   | Denotes Display Manager running (about to be displayed), but no users logged in      |
|      | yet.                                                                                 |
|------+---------------------------------------------------------------------------------------|
|  G   | Generated for each mount that completes successfully.                                |
|------+---------------------------------------------------------------------------------------|
|  H   | Emitted when mount attempt for single entry from fstab(5) for any filesystem type    |
|      | is about to begin.                                                                   |
|------+---------------------------------------------------------------------------------------|
|  I   | Emitted when Upstart receives the SIGPWR signal.                                     |
|------+---------------------------------------------------------------------------------------|
|  J   | Initial event.                                                                       |
|------+---------------------------------------------------------------------------------------|
|  K   | Although the events are emmitted by init(8), the instigator may be initctl(8) if a   |
|      | System Administrator has manually started or stopped a job.                          |
|------+---------------------------------------------------------------------------------------|
|  L   | /etc/init/umountnfs.sh.                                                              |
|------+---------------------------------------------------------------------------------------|
|  M   | Emitted when all virtual filesystems (such as /proc) mounted.                        |
|------+---------------------------------------------------------------------------------------|
|  N   | Emitted when the --dev-wait-time timeout is exceeded for mountall(8). This defaults  |
|      | to 30 seconds.                                                                       |
|------+---------------------------------------------------------------------------------------|
|  O   | Emitted when the X(7) display manager exits at shutdown or reboot, to hand off to    |
|      | the shutdown splash manager.                                                         |
|------+---------------------------------------------------------------------------------------|
|  P   | Emitted by /etc/init.d/networking just prior to stopping all non-local network       |
|      | interfaces.                                                                          |
|------+---------------------------------------------------------------------------------------|
|  Q   | Either 'container' or 'not-container' is emitted (depending on the environment),     |
|      | but not both.                                                                        |
|------+---------------------------------------------------------------------------------------|
|      | Emitted by either the initramfs or bootloader (for example grub) as the initial      |
|  R   | event (rather than startup(7)) to denote the system has booted into recovery mode.   |
|      | If recovery was successful, the standard startup(7) event is then emitted, allowing  |
|      | the system to boot as normal.                                                        |
|------+---------------------------------------------------------------------------------------|
|      | Emitted to indicate the system has failed to boot within the expected time. This     |
|  S   | event will trigger other jobs to forcibly attempt to bring the system into a usable  |
|      | state.                                                                               |
+----------------------------------------------------------------------------------------------+
```

28 Footer

---

1 Yes.

2 `initctl show-config -e`. See initctl show-config.

3 Job would start "as early as possible": when the `startup` event is emitted (see Startup Process). It would *also* be run if the confusingly-named job called "`stopped`" begun to start (see Starting a Job). It would also be run *again* if the also confusingly-named job "`started`" begun to stop (see Stopping a Job). The example chose names that were designed to be confusing. Clearly, in reality you should only create jobs with sensible names that refer to the application they run.

4 Three times.

5 `/tmp` is not mounted.

Document generated from reStructuredText plaintext markup source on 2013-01-25 at 08:12:20 from Bazaar branch branch. See Upstart Documenters and Upstart Cookbook. (revision $Revision-Id).

6 Short answer: "`/usr/bin/myapp`" will *never* run. Long answer: This job attempts to only start `myapp` if it is not disabled by checking its configuration file. However, there are two fatal flaws here:

- The script section does not handle the scenario where `/etc/default/myapp` does not exist. If it doesn't exist, the script will *immediately exit* causing the job to fail to start. See Debugging a Script Which Appears to be Behaving Oddly to understand why.
- Even if the `/etc/default/myapp` configuration file exists, the job will fail due to the use of expect fork and respawn with a script section.

A corrected version of the Job Configuration File is:

```
start on runlevel [2345]

env CONFIG=/etc/default/myapp

expect fork
respawn

pre-start
  [ -f "$CONFIG" ] || stop && exit 0
  enabled=$(grep ENABLED=1 $CONFIG || :)
  [ -z "$enabled" ] && exit 0
end script

exec /usr/bin/myapp
```

Or, if you need to pass options from the config file to the daemon, you could say:

```
start on runlevel [2345]

env CONFIG=/etc/default/myapp

expect fork
respawn

pre-start
  [ -f "$CONFIG" ] || stop && exit 0
  enabled=$(grep ENABLED=1 $CONFIG || :)
  [ -z "$enabled" ] && exit 0
end script

script
  . $CONFIG
  exec myapp $MYAPP_OPTIONS
end script
```

Note how the config file is sourced in the script section and how we specify the shell keyword `exec` to ensure no sub-shell is created (thus allowing Upstart to track the correct PID).

7 Recall that Upstart has no knowledge of disks whatsoever. In Ubuntu, it relies upon mountall (ubuntu-specific) to handle mounting of disks.

8 Note the method for obtaining the PID of the instance of Upstart running in the LXC container assumes only one other container is running.

9 Note that some shells (including Bash) change their behaviour if invoked as `/bin/sh`. Consult your shells documentation for specifics.

10 Commands to be run as root directly for clarity. However, you should consider using sudo(8) rather than running a root shell. Due to the way sudo works, you have to modify your behaviour slightly. For example, rather than running the following in a root shell:

```
# echo hello > /tmp/root.txt
```

You would instead run the command below in a **non-root** shell:

```
$ echo hello | sudo tee /tmp/root.txt
```

Note that you should not use *sudo within* a job. See Changing User.

11 If there is a `script` or `exec` section and this process is running, state will be `pre-stop`, else it will be `stopping`.

12 Note that the `exec` line is taken directly from the `org.freedesktop.ConsoleKit.service` file.

13 Upstart was written specifically for Ubuntu, although this does not mean that it cannot run on any other Linux-based system. Upstart was first introduced into Ubuntu in release 6.10 ("Edgy Eft"). See http://www.ubuntu.com/news/610released

14 This section of the document contains Ubuntu-specific examples of events. Other operating systems which use Upstart may not implement the same behaviour.

15(1, 2) This job is not actually available in Ubuntu yet, but is expected to be added early in the 11.10 development cycle.

16(1, 2) Note that pre-stop does not behave in the same manner as other script sections. See bug 703800 (https://bugs.launchpad.net/ubuntu/+source/upstart/+bug/703800)

17 For status on chroot support, see bugs 430224 and 728531: - https://bugs.launchpad.net/ubuntu/+source/upstart/+bug/430224 - https://bugs.launchpad.net/ubuntu/+source/upstart/+bug/728531

18 https://bugs.launchpad.net/upstart/+bug/406397

19 https://bugs.launchpad.net/upstart/+bug/888910

20 http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=607844

21 A series of blog posts by Scott James Remnant gives further details on events and how they are used. See [22], [23], and [24].

22(1, 2) http://upstart.at/2010/12/08/events-are-like-signals/

23(1, 2) http://upstart.at/2011/01/06/events-are-like-hooks/

24 http://upstart.at/2010/12/16/events-are-like-methods/

25(1, 2) http://upstart.at/2011/03/25/visualisation-of-jobs-and-events-in-ubuntu-natty/

26 http://upstart.at/2011/03/16/checking-jobs-and-events-in-ubuntu-natty/

27 http://upstart.at/2011/03/11/override-files-in-ubuntu-natty/

28 Ubuntu will kill any jobs still running at system shutdown using `/etc/init.d/sendsigs`.

29 Note that there is no "`startup`" job (and hence no `/etc/init/startup.conf` file).

30 It is worth noting that Unix and Linux systems are confined by standards to the runlevels specified in the Runlevels section. However, in principle Upstart allows any number of runlevels.

31 https://wiki.ubuntu.com/ReplacementInit

32 http://people.canonical.com/~jhunt/upstart/devel/upstart_objects.png

33 http://people.canonical.com/~jhunt/upstart/utils/upstart_menu.sh

34 http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Technical_Notes/deployment.html