Being systematic with Systemd

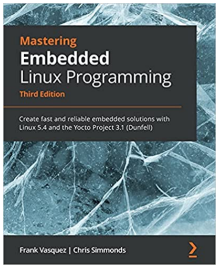
systemd for embedded Linux

Chris Simmonds

Embedded Linux Conference Europe 2022

License

About Chris Simmonds



- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at `https://2net.co.uk/`

@2net_software

`https://uk.linkedin.com/in/chrisdsimmonds/`

Agenda

- Systemd 101
- Loading services on demand
- Restarting services
- Watchdog
- Resource limits

Previously ...

- ELC-E 2019: We need to talk about systemd

https://2net.co.uk/slides/elc/systemd-csimmonds-elce-2019.pdf

This time ...

- Using systemd to boot and manage embedded Linux
- With demos

Concepts

- Bootstrapping a computer is best expressed as a hierarchy
  - some things can't start until other things have been started
  - by expressing dependencies between things you create a tree structure
  - systemd just needs to walk the tree to reach a goal, called a **target**
- Meta information is written in a simple form, called a **unit**
- Daemons are represented as **service** units
- We will meet other kinds of unit as we go on

Units

- All units have a `[Unit]` section
- Contains a description, reference to documentation and dependencies on other units
- Example: the Unit section from `/lib/systemd/system/dbus.service`

```
[Unit]
Description=D-Bus System Message Bus
Documentation=man:dbus-daemon(1)
Requires=dbus.socket
[...]
```

https://www.freedesktop.org/software/systemd/man/systemd.unit.html

Unit dependencies

- **Requires**: a list of units this unit depends on which must be started as well
- **Wants**: a weaker form of Requires: this unit is started even if any in the list fail
- **Conflicts**: a negative dependency: the units listed are stopped when this one is started and, conversely, if one of them is started, this one is stopped

These all operate on the **activation queue**

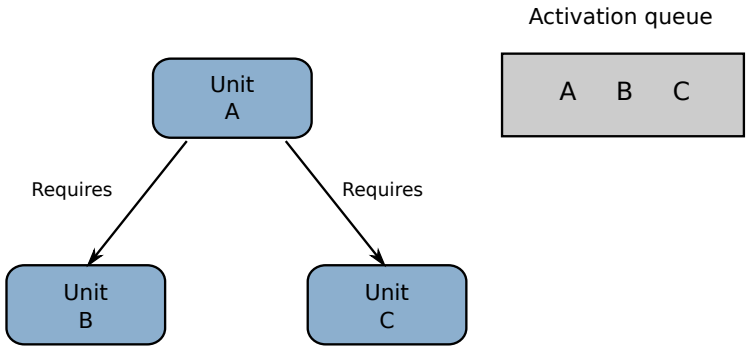
Order: Before and After

- These keywords determine the order that units are started
- **Before**: this unit should be started before the units listed
- **After**: this unit should be started after the units listed
- Example: start a daemon *after* the network target

```
[Unit]
Description=Lighttpd Web Server
After=network.target
[...]
```

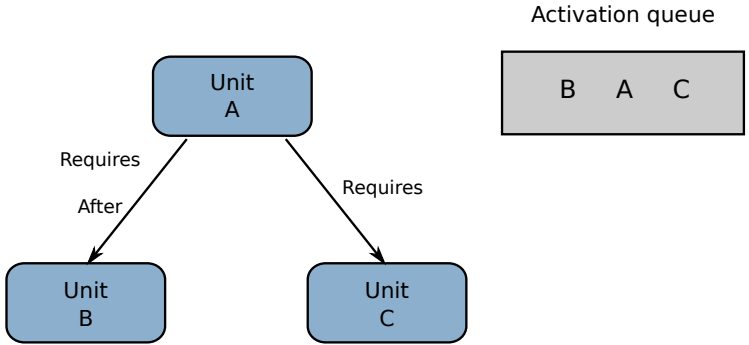
- Without Before or After, units are started in no particular order

Dependencies vs ordering



Starting Unit A will add A, B and C to the activation queue, but they may run in any order, even simultaneously

Dependencies vs ordering



Now, Unit B must run before Unit A
Unit C can run whenever it likes

Unit search path

- Systemd searches for units working from most specific to most general configuration
  - `/etc/systemd/system`: Local configuration
  - `/run/systemd/system`: Runtime configuration
  - `/lib/systemd/system`: Distribution-wide configuration
- To override a unit, just place a unit with the same name earlier in the sequence (usually `/etc/systemd/system`)
- To disable a unit, replace it with an empty file or a link to `/dev/null`

Service

- A service is a unit that controls a daemon
- Name ends in `.service`
- Has a `[Service]` section
- Example, `lighttpd.service`

```
[Unit]
Description=Lighttpd Web Server
After=network.target

[Service]
ExecStart=/usr/sbin/lighttpd -f /etc/lighttpd/lighttpd.conf -D
ExecReload=/bin/kill -HUP $MAINPID
```

https://www.freedesktop.org/software/systemd/man/systemd.service.html

Service Options

Type of service

```
Type=simple       (default) systemd launches the program in the background
    =oneshot      run once, do not retart
    =forking      the daemon runs in the background, e.g. by calling daemon(2)
```

Starting and restarting the daemon

```
ExecStart=        the program to run
ExecReload=       what to do following "systemctl restart"
```

Environment variable (see systemd.exec(5) for a full list)

```
MAINPID           the PID of the unit's main process
```

systemctl

`systemctl` is a command line interface for systemd. Here are some useful commands:

Command	Description
`start [unit]`	start a unit
`stop [unit]`	stop a unit
`enable[unit]`	install the unit, creating the `wants` link
`disable[unit]`	uninstall the unit
`status [unit]`	show status of a unit
`get-default`	show default target
`list-dependencies`	list dependency tree

https://www.freedesktop.org/software/systemd/man/systemctl.html

Systemd in Yocto Project

- Out-of-the-box, Yocto Project uses SystemV init daemon
- To switch to systemd, add this to a suitable conf file

conf/local.conf

```
INIT_MANAGER = "systemd"
```

Demo: start a daemon

Demo: start a daemon

called boris

Demo: start a daemon

called boris

Built using Yocto Project 4.0.1, systemd 250, target qemuarm

Target

- A Target is a Unit that lists dependencies on other Targets
- Name ends in `.target`
- Example, `/lib/systemd/system/multi-user.target`

```
[Unit]
Description=Multi-User System
Documentation=man:systemd.special(7)
Requires=basic.target
Conflicts=rescue.service rescue.target
After=basic.target rescue.service rescue.target
```

https://www.freedesktop.org/software/systemd/man/systemd.target.html

The default target

- At boot, systemd starts `default.target`
- Usually a symbolic link to the target desired
- Example

```
/etc/systemd/system/default.target -> /lib/systemd/system/multi-user.target
```

- Default target may be overridden on kernel command line:
  `system.unit=<new target>`

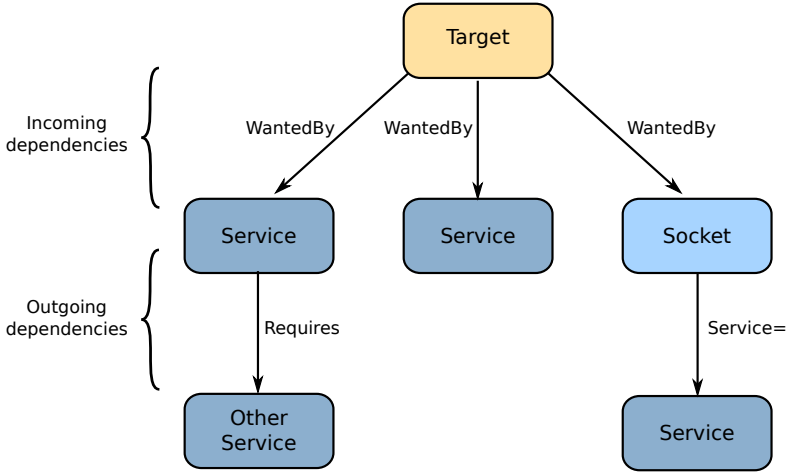
Reverse dependencies: WantedBy

- **Requires** and **Wants** create **outgoing** dependencies
- We also have **incoming** dependencies, which are links from other units to this unit
- Incoming dependencies are created by **WantedBy**
- WantedBy appears in the **Install** section

Example: a server that is started by multi-user.target

```
[Unit]
Description=A simple daemon
[Service]
ExecStart=/usr/bin/simpledaemon
[Install]
WantedBy=multi-user.target
```

Dependencies



The Install section

- The incoming link is created by `systemctl enable`
  - and deleted by `systemctl disable`
- The dependency is expressed as a symbolic link in subdirectory `<unit name>.wants`
- Example: installing simpledaemon creates this link

```
$ systemctl enable simpledaemon
$ ls -l /etc/systemd/system/multi-user.target.wants
/etc/systemd/system/multi-user.target.wants/simpledaemon.service ->
       /lib/systemd/system/simpledaemon.service
```

Preinstalling services in Yocto

- You want some services to be enabled in the system image
- In Yocto, this is handled by the systemd class

`simpledaemon.bb`

```
[...]
inherit systemd
SYSTEMD_SERVICE:${PN} = "simpledaemon.service"
[...]
```

Now the image contains

```
/etc/systemd/system/multi-user.target.wants -> /lib/systemd/system/simpledaemon.service
```

Demo: enable boris at boot

- Systemd 101
- Loading services on demand
- Restarting services
- Watchdog
- Resource limits

Loading services on demand

- The **socket** unit waits for some event, then starts a service when the event is triggered
- Name ends in `.socket`
- Example, `foo.socket`

```
[Unit]
Description=Start foo.service when a connection is received from TCP port 1234

[Socket]
ListenStream=1234
Accept=no

[Install]
WantedBy=sockets.target
```

https://www.freedesktop.org/software/systemd/man/systemd.socket.html

Types of "socket"

- A socket unit can wait on network and local sockets, FIFOs and other things through the **Listen*** option in the Socket section

Component	Address format	Example	Connection
ListenStream	port number	22	inet or inet6 socket
ListenStream	/[path name]	/run/socket	Local socket
ListenFIFO	/[path name]	/run/fifo	FIFO
ListenSpecial	/[path name]	/dev/rfkill	Device node or sysfs file
ListenNetlink	name	kobject-uevent	`AF_NETLINK` socket
ListenMessageQueue	/[mq name]	/messages	POSIX message queue
ListenUSBFunction	/[ffs mount]	/run/ffs_test	FunctionFS endpoint

Starting the service

- By default, a socket starts a service with the same name
  - `foo.socket` starts `foo.service`
- You can override with `Service` option

```
[Socket]
ListenStream=1234
Accept=no
Service=bar
```

ListenSpecial example

- ListenSpecial opens the file `O_RDONLY` (`O_RDWR` if Writable=yes) and blocks in epoll waiting for a POLLIN event (i.e. data to read)

`systemd-rfkill.socket`

```
[Socket]
ListenSpecial=/dev/rfkill
Writable=yes
```

The service can get an array of open fds from systemd via `sd_listen_fds`

See `SYSTEMD/src/rfkill/rfkill.c` for implementation

Service templates

- Some network daemons spawn a copy for each connection (e.g. sshd)(*)
  - indicated by setting `Accept=yes` in `[Socket]` section
- Use a **service template** to create a different service instance for each connection
- Template names are of the form `foo@.service`
  - the `@` is replaced by an instance name when the service is started

(*) replicating the behaviour of inetd and xinetd from days of yore

Service template example

ssh.socket

```
[Unit]
Description=OpenBSD Secure Shell server socket
Before=ssh.service
Conflicts=ssh.service

[Socket]
ListenStream=22
Accept=yes

[Install]
WantedBy=sockets.target
```

ssh@.service

```
[Unit]
Description=OpenBSD Secure Shell server
Documentation=man:sshd(8) man:sshd_config(5)
After=auditd.service

[Service]
EnvironmentFile=-/etc/default/ssh
ExecStart=-/usr/sbin/sshd -i $SSHD_OPTS
StandardInput=socket
RuntimeDirectory=sshd
RuntimeDirectoryPreserve=yes
RuntimeDirectoryMode=0755
```

The service is named after the template plus elements of the connection:

```
$ systemctl status ssh.socket
   Triggers: * ssh@0-192.168.4.110:22-192.168.4.28:35406.service
```

Demo: starting an ssh daemon

Timers

- A timer unit is similar to a socket, except the event is time triggered

`foo.timer`

```
[Unit]
Description=Wait 30 seconds before running foo.service
[Timer]
OnActiveSec=30sec
[Install]
WantedBy=timers.target
```

Delays for 30 seconds before running a service

The timer specification can also generate periodic or calendar events

https://www.freedesktop.org/software/systemd/man/systemd.timer.html

- Systemd 101
- Loading services on demand
- Restarting services
- Watchdog
- Resource limits

Restarting services

- What happens if a service terminates for some reason?
- systemd has a range of recovery options

Restart

- Restart is controlled by the **Restart** option in the `[Service]` section

```
Restart=no              (default) no restart action
       =on-success      only restart if exit(0), or on SIGHUP, SIGINT, SIGTERM
       =on-failure      restart if exit > 0, uncaught signal, watchdog timeout
       =on-watchdog     restart only in the case watchdog timesout
       =on-abort        restart only if uncaught signal
       =always          restart if the service terminates for *any* reason
```

Example:

```
[Service]
Restart=on-failure
```

Limiting restarts

- Sometimes, restarting the service just causes it to crash again
- You can control this behaviour by setting the maximum number of restarts that should be attempted in a given period

Example: if this service terminates twice in 30 seconds, leave it in the stopped state

```
[Unit]
StartLimitBurst=2
StartLimitIntervalSec=30

[Service]
ExecStart=/usr/bin/simpledaemon
Restart=on-failure
```

Trying to fix things

- Maybe there is some cleanup that needs to be done, or some remedial action
- You can tell systemd to run a unit on failure like this:

```
[Unit]
StartLimitBurst=2
StartLimitIntervalSec=30
OnFailure=simpledaemon-cleanup.service

[Service]
ExecStart=/usr/bin/simpledaemon
Restart=on-failure
```

More drastic action

- Maybe the service is critical, and leaving it stopped is not an option
- You can cause a reboot (in the hope that that solves the problem)

```
[Unit]
FailureAction=reboot
```

- Systemd 101
- Loading services on demand
- Restarting services
- Watchdog
- Resource limits

Watchdog

- A watchdog will cause a service to restart if it times out

Example: restart a service if the watchdog is not pinged within 30s

```
[Service]
WatchdogSec=30s
Restart=on-watchdog
```

Watchdog

- You ping the watchdog using `sd_notify` (part of libsystemd.so)

```
#include <systemd/sd-daemon.h>

[...]
    sd_notify(0, "WATCHDOG=1");
```

https://www.freedesktop.org/software/systemd/man/sd_notify.html

Hardware watchdog

- You can configure systemd to use a hardware watchdog
- systemd will configure the watchdog timeout and then attempt to ping it within that period (usually at RuntimeWatchdogSec/2)

`/lib/systemd/system.conf.d/*.conf, /etc/systemd/system.conf`

```
RuntimeWatchdogSec=       watchdog timeout, default "off"
WatchdogDevice=           default /dev/watchdog0
```

https://www.freedesktop.org/software/systemd/man/systemd-system.conf

- Systemd 101
- Loading services on demand
- Restarting services
- Watchdog
- Resource limits

Resource limits

- For resilience it is useful to set limits on the resources available to some daemons
  - and take remedial action when they do, e.g. stopping the daemon
- Example, here is a service with CPU quota 20%

```
[Service]
ExecStart=/usr/bin/simpledaemon
CPUQuota=20%
```

https://www.freedesktop.org/software/systemd/man/systemd.resource-control.html

Limits are enforced using **cgroups**

Setting a memory resource limit

```
[Service]
ExecStart=/usr/bin/simpledaemon
MemoryMax=4096K
```

The limit is enforced here:

```
$ cat /sys/fs/cgroup/memory/system.slice/eatmem.service/memory.limit_in_bytes
16777216
```

When the daemon exceeds the limit, it will be killed with SIGKILL

```
eatmem.service: Main process exited, code=killed, status=9/KILL
eatmem.service: Failed with result 'signal'.
```

Questions?

Slides at
https://2net.co.uk/slides/elc/systemd-csimmonds-elce-2022.pdf

@2net_software

https://uk.linkedin.com/in/chrisdsimmonds/