The GNU Shepherd Manual

For use with the GNU Shepherd 1.0.7
Last updated 28 September 2025

Wolfgang Jährling
Ludovic Courtès

Table of Contents

1 Introduction 1

2 Jump Start 2

3 `herd` and `shepherd` 4

3.1 Invoking `shepherd` 4
3.2 Invoking `herd` 6
3.3 Invoking `reboot` 7
3.4 Invoking `halt` 8

4 Services 9

4.1 Defining Services 9
4.2 Service Registry 13
4.3 Interacting with Services 13
4.4 Service De- and Constructors 15
4.5 Timers 20
4.6 The `root` Service 22
4.7 Legacy GOOPS Interface 23
4.8 Service Examples 24
4.9 Managing User Services 26

5 Service Collection 28

5.1 System Log Service 28
5.2 Log Rotation Service 30
5.3 Transient Service Maker 31
5.4 Timer Service 32
5.5 Monitoring Service 33
5.6 Read-Eval-Print Loop Service 33

6 Misc Facilities 35

6.1 Process Utilities 35
6.2 Errors 35
6.3 Communication 35

7 Internals 37

7.1 Coding Standards 37
7.2 Service Internals 37
7.3 Design Decisions 38

Appendix A GNU Free Documentation License .. 41

Concept Index **49**

Procedure and Macro Index **51**

Variable Index **53**

Type Index **54**

1 Introduction

This manual documents the GNU Daemon Shepherd, or GNU Shepherd for short. The Shepherd looks after system services, typically *daemons*. It is used to start and stop them in a reliable fashion. For instance, it will dynamically determine and start any other services that our desired service depends upon.

The Shepherd is the *init system* of the GNU operating system—it is the first user process that gets started, typically with PID 1, and runs as `root`. Normally the purpose of init systems is to manage all system-wide services, but the Shepherd can also be a useful tool assisting unprivileged users in the management of their own daemons.

Flexible software requires some time to master and the Shepherd is no different. But don't worry: this manual should allow you to get started quickly. Its first chapter is designed as a practical introduction to the Shepherd and should be all you need for everyday use (see Chapter 2 [Jump Start], page 2). In chapter two we will describe the `herd` and `shepherd` programs, and their relationship, in more detail (Chapter 3 [herd and shepherd], page 4). Subsequent chapters provide a full reference manual and plenty of examples, covering all of Shepherd's capabilities. Finally, the last chapter provides information for those souls brave enough to hack the Shepherd itself.

The Shepherd was formerly known as "dmd", which stands for *Daemon Managing Daemons* (or *Daemons-Managing Daemon*?).

This program is written in Guile Scheme. Guile is also the Shepherd's configuration language. See Section "Introduction" in *GNU Guile Reference Manual*, for an introduction to Guile. We have tried to make the Shepherd's basic features as accessible as possible—you should be able to use these even if you do not know how to program in Scheme. A basic grasp of Guile is required only if you wish to make use of the Shepherd's more advanced features.

2 Jump Start

The Shepherd comes with two main commands: `shepherd`, which is the daemon that manages services (see Section 3.1 [Invoking shepherd], page 4), and `herd`, the command to monitor and control `shepherd` (see Section 3.2 [Invoking herd], page 6). The `shepherd` command itself may run as an "init system", also known as PID 1[1], where it manages system services, or as a user, where it manages user services.

As a user and system administrator, the `herd` command is your main entry point; it allows you to get an overview of the services and their status:

```
herd status
```

... to get detailed information about a specific service, including recently-logged messages:

```
herd status sshd
```

... and to start, stop, restart, enable, or disable services, among other possible actions:

```
herd start nginx
herd stop tor
herd disable tor
herd restart guix-daemon
```

Each service may depend on other services. Services and their dependencies form a *graph*, which you can view, for instance with the help of xdot (`https://github.com/jrfonseca/xdot.py`), by running:

```
herd graph | xdot -
```

Since `shepherd` knows about these dependencies, the `herd start nginx` command above starts not just the `nginx` service, but also every service it depends on. Likewise, `herd stop tor` stops not just `tor` but also everything that depends on it.

Services may define *custom actions* in addition to `start`, `stop`, and the other actions mentioned above. For example, the `log-rotation` service has a custom `files` action to list log files subject to rotation (see Section 5.2 [Log Rotation Service], page 30):

```
herd files log-rotation
```

A special service is `root`, which is used for controlling the Shepherd itself (see Section 4.6 [The root Service], page 22). Its `status` action displays info about the `shepherd` process, including messages it recently logged, possibly on behalf of other services:

```
herd status root
```

The `load` action of `root` lets you dynamically load new service definitions from a configuration file, which we'll discuss later; note that this action takes the file name as an extra argument:

```
herd load root ~/additional-services.scm
```

To view a log of service events, including the time at which each service was started or stopped, run:

```
herd log
```

Last, you can view documentation about services and their actions:

```
$ herd doc root
```

[1] In that case it is the first process started by the system, hence the process identifier (PID) 1.

```
The root service is used to operate on shepherd itself.
$ herd doc root action power-off
power-off: Halt the system and turn it off.
```

That's pretty much all there is to know from the user and system administrator viewpoint.

Packagers, distro developers, and advanced system administrator, certainly noticed that we haven't talked about a crucial aspect for them: the configuration file for `shepherd`. That's certainly the most interesting and unique bit of the Shepherd: *configuration is a Scheme program*. You do not need to be an expert in the Scheme programming language to write it though: for common uses, configuration is entirely declarative and contains the same information you would find in a systemd `.service` file or anything similar. You can define services that spawn a process, others that wait for incoming connections and pass it on to a daemon (inetd-style or systemd-style *socket activation*), as well as timed services that run a program periodically. See Section 4.8 [Service Examples], page 24, to get started.

What's interesting is that you can go beyond the kind of services Shepherd offers you: You need a service that mounts/unmounts file systems? A service that sets up networking? One that runs a daemon in a container? This is where extensibility shines: you can write services that do this—and more. See Section 4.1 [Defining Services], page 9, for a reference of the programming interface to define services. And, whether or not you use Guix System, check out the variety of services it provides (see Section "Services" in *GNU Guix Reference Manual*).

3 herd and shepherd

The daemon that runs in the background and is responsible for controlling the services is `shepherd`, while the user interface tool is called `herd`: it's the command that allows you to actually *herd* your daemons[1]. To perform an action, like stopping a service or calling an action of a service, you use the herd program. It will communicate with shepherd over a Unix Domain Socket.

Thus, you start `shepherd` once, and then always use `herd` whenever you want to do something service-related. Both `shepherd` and `herd` understand the standard arguments `--help`, `--version` and `--usage`.

3.1 Invoking shepherd

The `shepherd` program has the following synopsis:

```
shepherd [option...]
```

When `shepherd` starts, it reads and evaluates a configuration file—actually a Scheme program that uses the Shepherd's programming interface to define, register, and start services (see Section 4.1 [Defining Services], page 9). When it is started with superuser privileges, it tries to read `/etc/shepherd.scm`; when started as a unprivileged user, it looks for a file called `$XDG_CONFIG_HOME/shepherd/init.scm`. If the `XDG_CONFIG_HOME` environment variable is undefined, `$HOME/.config/shepherd/init.scm` is used instead (see Section 4.9 [Managing User Services], page 26). The `--config` option described below lets you choose a different file.

As the final "d" suggests, `shepherd` is a daemon that runs in the background, so you will not interact with it directly. After it is started, `shepherd` listens on a socket, usually `/var/run/shepherd/socket` (this can be changed with the `--socket` option or the `SHEPHERD_SOCKET` environment variable). The *herd* tool sends commands to `shepherd` using this socket (see Section 3.2 [Invoking herd], page 6).

The `shepherd` command accepts the following options:

'`-c` *file*'
'`--config=`*file*'
: Read and evaluate *file* as the configuration file on startup.

  Scheme code in *file* is evaluated in the context of a fresh module where bindings from the `(shepherd service)` module are available, in addition to the default set of Guile bindings. It may typically perform three actions:

  1. defining services using the `service` procedure (see Section 4.1 [Defining Services], page 9);
  2. registering those services with `register-services` (see Section 4.2 [Service Registry], page 13);
  3. starting some or all of those services with `start-in-the-background` (see Section 4.3 [Interacting with Services], page 13).

[1] In the past, when the GNU Shepherd was known as GNU dmd, the `herd` command was called `deco`, for *DaEmon COntroller*.

See Section 4.8 [Service Examples], page 24, for examples of what *file* might look like.

Note that *file* is evaluated *asynchronously*: `shepherd` may start listening for client connections (with the `herd` command) before *file* has been fully loaded. Errors in *file* such as service startup failures or uncaught exceptions do *not* cause `shepherd` to stop. Instead the error is reported, leaving users the opportunity to inspect service state and, for example, to load an updated config file with:

```
herd load root file
```

'`-I`'
'`--insecure`'
: Do not check if the directory where the socket—our communication rendez-vous with `herd`—is located has permissions `700`. If this option is not specified, `shepherd` will abort if the permissions are not as expected.

'`-l [file]`'
'`--logfile[=file]`'
: Log output into *file*.

  For unprivileged users, the default log file is `$XDG_STATE_HOME/shepherd/shepherd.log` with `$XDG_STATE_HOME` defaulting to `$HOME/.local/state`.

'`--syslog[=file]`'
: Force logging to *file* (`/dev/log` by default), a *syslog* socket (see Section "Overview of Syslog" in *The GNU C Library Reference Manual*).

  When running as root, the default behavior is to connect to `/dev/log`. A syslog daemon, `syslogd`, is expected to read messages from there (see Section "syslogd invocation" in *GNU Inetutils*).

  When `/dev/log` is unavailable, for instance because `syslogd` is not running, as is the case during system startup and shutdown, `shepherd` falls back to the Linux kernel *ring buffer*, `/dev/kmsg`. If `/dev/kmsg` is missing, as is the case on other operating systems, it falls back to `/dev/console`.

'`--pid[=file]`'
: When `shepherd` is ready to accept connections, write its PID to *file* or to the standard output if *file* is omitted.

'`-s file`'
'`--socket=file`'
: Receive further commands on the socket special file *file*.

  If this option is not specified, *`localstatedir`*`/run/shepherd/socket` is taken when running as `root`; when running as an unprivileged user, `shepherd` listens to `/run/user/`*uid*`/shepherd/socket`, where *uid* is the user's numerical ID[2], or to `$XDG_RUNTIME_DIR/shepherd` when the `XDG_RUNTIME_DIR` environment variable is defined.

[2] On GNU/Linux, the `/run/user/`*uid* directory is typically created by elogind or by systemd, which are available in most distributions.

Note: If the listening socket is deleted, it becomes impossible to control `shepherd`. Thus, upon deletion, `shepherd` tries to reopen the socket (when running as PID 1) or shuts down (when running as an unprivileged user).

The latter behavior is useful when the listening socket is under `/run/user/`*uid* (the default) since that directory is usually deleted as soon as the user session terminates.

If `-` is specified as file name, commands will be read from standard input, one per line, as would be passed on a `herd` command line (see Section 3.2 [Invoking herd], page 6).

'`-S`'
'`--silent`'
Don't do output to stdout.

'`--quiet`' Synonym for `--silent`.

3.2 Invoking `herd`

The `herd` command is a generic client program to control a running instance of `shepherd` (see Section 3.1 [Invoking shepherd], page 4). When running as root, it communicates with the *system instance*—the process with PID 1; when running as a normal user, it communicates with the *user's instance*, which is a regular, unprivileged process managing the user's own services (this can be changed by passing the `--socket` option or by setting the `SHEPHERD_SOCKET` environment variable). For example, the following command displays the status of all the *system services*:

```
sudo herd status
```

Conversely, the command below displays the status of *user services*, assuming a user `shepherd` is running:

```
herd status
```

The command has the following synopsis:

```
herd [option...] action [service [arg...]]
```

It causes the *action* of the *service* to be invoked. When *service* is omitted and *action* is `status` or `detailed-status`, the `root` service is used[3] (see Section 4.6 [The root Service], page 22, for more information on the `root` service.)

For each action, you should pass the appropriate *arg*s. Actions that are available for every service are `start`, `stop`, `restart`, `status`, `enable`, `disable`, and `doc`.

If you pass a file name as an *arg*, it will be passed as-is to the Shepherd, thus if it is not an absolute name, it is local to the current working directory of `shepherd`, not to that of `herd`.

The `herd` command understands the following option:

[3] This shorthand does not work for other actions such as `stop`, because inadvertently typing `herd stop` would stop all the services, which could be pretty annoying.

'`-s file`'
'`--socket=file`'
: Send commands to the socket special file *file*. If this option is not specified, `localstatedir/run/shepherd/socket` is taken.

'`-n number`'
'`--log-history=number`'
: Display up to *number* lines of the of *service* when running:

```
herd status service -n number
```

By default up to ten lines are printed.

In addition, the options below are understood by service actions that spawn other processes or services, such as `herd schedule timer` (see [timer-schedule-command], page 32) and `herd spawn transient` (see Section 5.3 [Transient Service Maker], page 31).

'`--user=user`'
'`--group=group`'
: Run the process under the specified *user* and *group*.

'`-d directory`'
'`--working-directory=directory`'
: Run the process or service from *directory*.

'`-E environment`'
'`--environment-variable=environment`'
: Add *environment* to the environment variables of the service, where *environment* has the form `variable=value`.

'`-N name`'
'`--service-name=name`'
: Register the service under *name*.

'`--log-file=file`'
: Log service output to *file*.

The `herd` command returns zero on success, and a non-zero exit code on failure. In particular, it returns a non-zero exit code when *action* or *service* does not exist and when the given action failed.

3.3 Invoking `reboot`

The `reboot` command is a convenience client program to instruct the Shepherd (when used as an init system) to stop all running services and reboot the system. It has the following synopsis:

```
reboot [option...]
```

It is equivalent to running `herd stop shepherd`. The `reboot` command understands the following option:

'`-s file`'
'`--socket=file`'
: Send commands to the socket special file *file*. If this option is not specified, `localstatedir/run/shepherd/socket` is taken.

‘`-k`’
‘`--kexec`’ Reboot the system using Linux’s kexec (this is equivalent to running `herd kexec root`). The kernel that was previously loaded using the `kexec -l` *`file`* command is executed directly instead of rebooting into the BIOS, keeping the downtime to a minimum. See the `kexec` command documentation (`https://linux.die.net/man/8/kexec`) for more information.

This feature is only available on Linux-based systems. It has no effect on systems where kexec is unsupported or when no system was loaded for eventual kexec reboot. Last, if kexec reboot fails at run time, `shepherd` falls back to normal reboot.

3.4 Invoking `halt`

The `halt` command is a convenience client program to instruct the Shepherd (when used as an init system) to stop all running services and turn off the system. It has the following synopsis:

```
halt [option...]
```

It is equivalent to running `herd power-off shepherd`. As usual, the `halt` command understands the following option:

‘`-s` *`file`*’
‘`--socket=`*`file`*’
Send commands to the socket special file *file*. If this option is not specified, *`localstatedir`*`/run/shepherd/socket` is taken.

4 Services

The *service* is obviously a very important concept of the Shepherd. On the Guile level, a service is represented as a record of type `<service>`. Each service has a number of properties that you specify when you create it: how to start it, how to stop it, whether to respawn it when it stops prematurely, and so on. At run time, each service has associated *state*: whether it is running or stopped, what PID or other value is associated with it, and so on.

This section explains how to define services and how to query their configuration and state using the Shepherd's programming interfaces.

> **Note:** The programming interface defined in this section may only be used within a `shepherd` process. Examples where you may use it include:
>
> - the `shepherd` configuration file (see Section 4.8 [Service Examples], page 24);
> - as an argument to `herd eval root ...` (see Section 4.6 [The root Service], page 22);
> - at the REPL (see Section 5.6 [REPL Service], page 33).
>
> These procedures may not be used in Guile processes other than `shepherd` itself.

4.1 Defining Services

A service is created by calling the `service` procedure, from the `(shepherd service)` module (automatically visible from your configuration file), as in this example:

```
(service
  '(sshd ssh-daemon)  ;for convenience, give it two names
  #:start (make-forkexec-constructor
           '("/usr/sbin/sshd" "-D")
           #:pid-file "/etc/ssh/sshd.pid")
  #:stop (make-kill-destructor)
  #:respawn? #t)
```

The example above creates a service with two names, `sshd` and `ssh-daemon`. It is started by invoking `/usr/sbin/sshd`, and it is considered up and running as soon as its PID file `/etc/ssh/sshd.pid` is available. It is stopped by terminating the `sshd` process. Finally, should `sshd` terminate prematurely, it is automatically respawned. We will look at `#:start` and `#:stop` later (see Section 4.4 [Service De- and Constructors], page 15), but first, here is the reference of the `service` procedure and its optional keyword arguments.

`service` *provision* [*#:requirement '()*] [*#:one-shot? #f*] [*#:transient? #f*] [*#:respawn? #f*] [*#:start* (*const #t*)] [*#:stop* (*const #f*)] [*#:actions* (*actions*)] [*#:termination-handler default-service-termination-handler*] [*#:documentation #f*] [Procedure]

Return a new service with the given *provision*, a list of symbols denoting what the service provides. The first symbol in the list is the *canonical name* of the service, thus it must be unique.

The meaning of keyword arguments is as follows:

`#:requirement`
: `#:requirement` is, like *provision*, a list of symbols that specify services. In this case, they name what this service depends on: before the service can be started, services that provide those symbols must be started.

  Note that every name listed in `#:requirement` must be registered so it can be resolved (see Section 4.2 [Service Registry], page 13).

`#:respawn?`
: Specify whether the service should be respawned by `shepherd`. If this slot has the value `#t`, then, assuming the service has an associated process (its "running value" is a PID), restart the service if that process terminates.

  There is a limit to avoid endless respawning: when the service gets respawned "too fast", it is *disabled*—see `#:respawn-limit` below.

`#:respawn-delay`
: Specify the delay before a service is respawned, in seconds (including a fraction), for services marked with `#:respawn? #t`. Its default value is `(default-respawn-delay)` (see Section 4.4 [Service De- and Constructors], page 15).

`#:respawn-limit`
: Specify the limit that prevents `shepherd` from respawning too quickly the service marked with `#:respawn? #t`. Its default value is `(default-respawn-limit)` (see Section 4.4 [Service De- and Constructors], page 15).

  The limit is expressed as a pair of integers: the first integer, *n*, specifies a number of consecutive respawns and the second integer, *t*, specifies a number of seconds. If the service gets respawned more than *n* times over a period of *t* seconds, it is automatically *disabled* (see Section 4.3 [Interacting with Services], page 13). Once it is disabled, the service must be explicitly re-enabled using `herd enable` *`service`* before it can be started again.

  Consider the service below:

```
(service '(xyz)
         #:start (make-forkexec-constructor ...)
         #:stop (make-kill-destructor)
         #:respawn? #t
         #:respawn-limit '(3 . 5))
```

  The effect is that this service will be respawned at most 3 times over a period of 5 seconds; if its associated process terminates a fourth time during that period, the service will be marked as disabled.

`#:one-shot?`
: Whether the service is a *one-shot service*. A one-shot service is a service that, as soon as it has been successfully started, is marked as "stopped." Other services can nonetheless require one-shot services. One-shot ser-

vices are useful to trigger an action before other services are started, such as a cleanup or an initialization action.

As for other services, the `start` method of a one-shot service must return a truth value to indicate success, and false to indicate failure.

`#:transient?`
: Whether the service is a *transient service*. A transient service is automatically unregistered when it terminates, be it because its `stop` method is called or because its associated process terminates.

  This is useful in the uncommon case of synthesized services that may not be restarted once they have completed.

`#:start`
: Specify the *constructor* of the service, which will be called to start the service. This must be a procedure that accepts any number of arguments; those arguments will be those supplied by the user, for instance by passing them to `herd start`. If the starting attempt failed, it must return `#f` or throw an exception; otherwise, the return value is stored as the *running value* of the service.

  > **Note:** Constructors must terminate, successfully or not, in a timely fashion, typically less than a minute. Failing to do that, the service would remain in "starting" state and users would be unable to stop it.

  See Section 4.4 [Service De- and Constructors], page 15, for info on common service constructors.

`#:stop`
: This is the service *destructor*: a procedure of one or more arguments that should stop the service. It is called whenever the user explicitly stops the service; its first argument is the running value of the service, subsequent arguments are user-supplied. Its return value will again be stored as the running value, so it should return `#f` if it is now possible again to start the service at a later point.

  > **Note:** Destructors must also terminate in a timely fashion, typically less than a minute. Failing to do that, the service would remain in "stopping" state and users would be unable to stop it.

  See Section 4.4 [Service De- and Constructors], page 15, for info on common service destructors.

`#:termination-handler`
: The procedure to call when the process associated with the service terminates. It is passed the service, the PID of the terminating process, and its exit status, an integer as returned by `waitpid` (see Section "Processes" in *GNU Guile Reference Manual*).

  The default handler is the `default-service-termination-handler` procedure, which respawns the service if applicable.

`#:actions`
: The additional actions that can be performed on the service when it is running. A typical example for this is the `restart` action. The `actions` macro can be used to defined actions (see below).

A special service that every other service implicitly depends on is the `root` (also known as `shepherd`) service. See Section 4.6 [The root Service], page 22, for more information.

Services and their dependencies form a *graph*. At the command-line, you can view that export a representation of that graph that can be consumed by any application that understands the Graphviz format, such as xdot (`https://github.com/jrfonseca/xdot.py`):

```
herd graph | xdot -
```

Service actions are defined using the `action` procedure or the `actions` (plural) macro, as shown below.

`action` *name proc* [*doc*] [Procedure]
: Return a new action with the given *name*, a symbol, that executes *proc*, a one-argument procedure that is passed the service's running value. Use *doc* as the documentation of that action.

`actions` (*name proc*) ... [Macro]
: Create a value for the `#:actions` parameter of `service`. Each *name* is a symbol and each *proc* the corresponding procedure that will be called to perform the action. A *proc* has one argument, which will be the running value of the service.

Naturally, the `(shepherd service)` provides procedures to access this information for a given service object:

`service-provision` *service* [Procedure]
: Return the symbols provided by *service*.

`service-canonical-name` *service* [Procedure]
: Return the *canonical name* of *service*, which is the first element of the list returned by `service-provision`.

`service-requirement` *service* [Procedure]
: Return the list of services required by *service* as a list of symbols.

`one-shot-service?` *service* [Procedure]
`transient-service?` *service* [Procedure]
: Return true if *service* is a one-shot/transient service.

`respawn-service?` *service* [Procedure]
: Return true if *service* is meant to be respawned if its associated process terminates prematurely.

`service-respawn-delay` *service* [Procedure]
: Return the respawn delay of *service*, in seconds (an integer or a fraction or inexact number). See `#:respawn-delay` above.

`service-respawn-limit` *service* [Procedure]
Return the respawn limit of *service*, expressed as a pair—see `#:respawn-limit` above.

`service-documentation` *service* [Procedure]
Return the documentation (a string) of *service*.

4.2 Service Registry

At run time, `shepherd` maintains a *service registry* that maps service names to service records. Service dependencies are expressed as a list of names passed as `#:requirement` to the `service` procedure (see Section 4.1 [Defining Services], page 9); these names are looked up in the registry. Likewise, when running `herd start sshd` or similar commands (see Chapter 2 [Jump Start], page 2), the service name is looked up in the registry.

Consequently, every service must be appear in the registry before it can be used. A typical configuration file thus includes a call to the `register-services` procedure (see Section 4.8 [Service Examples], page 24). The following procedures let you interact with the registry.

`register-services` *services* [Procedure]
Register *services* so that they can be looked up by name, for instance when resolving dependencies.

Each name uniquely identifies one service. If a service with a given name has already been registered, arrange to have it replaced when it is next stopped. If it is currently stopped, replace it immediately.

`unregister-services` *services* [Procedure]
Remove all of *services* from the registry, stopping them if they are not already stopped.

`lookup-service` *name* [Procedure]
Return the service that provides *name*, `#f` if there is none.

`for-each-service` *proc* [Procedure]
Call *proc*, a procedure taking one argument, once for each registered service.

`lookup-running` *name* [Procedure]
Return the running service that provides *name*, or false if none.

4.3 Interacting with Services

What we have seen so far is the interface to *define* a service and to access it (see Section 4.1 [Defining Services], page 9). The procedures below, also exported by the `(shepherd service)` module, let you modify and access the state of a service. You may use them in your configuration file, for instance to start some or all of the services you defined (see Section 4.8 [Service Examples], page 24).

Under the hood, each service record has an associated *fiber* (really: an actor) that encapsulates its state and serves user requests—a fiber is a lightweight execution thread (see Section 7.2 [Service Internals], page 37).

The procedures below let you change the state of a service.

`start-service` *service* `.` *args* [Procedure]
: Start *service* and its dependencies, passing *args* to its `start` method. Return its running value, `#f` on failure.

`stop-service` *service* `.` *args* [Procedure]
: Stop *service* and any service that depends on it. Return the list of services that have been stopped (including transitive dependent services).

  If *service* is not running, print a warning and return its canonical name in a list.

`perform-service-action` *service the-action* `.` *args* [Procedure]
: Perform *the-action* (a symbol such as `'restart` or `'status`) on *service*, passing it *args*. The meaning of *args* depends on the action.

The `start-in-the-background` procedure, described below, is provided for your convenience: it makes it easy to start a set of services right from your configuration file, while letting `shepherd` run in the background.

`start-in-the-background` *services* [Procedure]
: Start the services named by *services*, a list of symbols, in the background. In other words, this procedure returns immediately without waiting until all of *services* have been started.

  This procedure can be useful in a configuration file because it lets you interact right away with shepherd using the `herd` command.

The following procedures let you query the current state of a service.

`service-running?` *service* [Procedure]
`service-stopped?` *service* [Procedure]
`service-enabled?` *service* [Procedure]
: Return true if *service* is currently running/stopped/enabled, false otherwise.

`service-status` *service* [Procedure]
: Return the status of *service* as a symbol, one of: `'stopped`, `'starting`, `'running`, or `'stopping`.

`service-running-value` *service* [Procedure]
: Return the current "running value" of *service*—a Scheme value associated with it. It is `#f` when the service is stopped; otherwise, it is a truth value, such as an integer denoting a PID (see Section 4.4 [Service De- and Constructors], page 15).

`service-status-changes` *service* [Procedure]
: Return the list of symbol/timestamp pairs representing recent state changes for *service*.

`service-startup-failures` *service* [Procedure]
`service-respawn-times` *service* [Procedure]
: Return the list of startup failure times or respawn times of *service*.

`service-process-exit-statuses` *services* [Procedure]
: Return the list of last exit statuses of *service*'s main process (most recent first).

`service-replacement` *service* [Procedure]
Return the *replacement* of *service*, or `#f` if there is none.

The replacement is the service that will replace *service* when it is eventually stopped.

`service-recent-messages` *service* [Procedure]
Return a list of messages recently logged by *service*—typically lines written by a daemon on standard output. Each element of the list is a timestamp/string pair where the timestamp is the number of seconds since January 1st, 1970 (an integer).

`service-log-file` *service* [Procedure]
Return the file where messages by *service* are logged, or `#f` if there is no such file, for instance because *service*'s output is logged by some mechanism not under shepherd's control.

See Section 7.2 [Service Internals], page 37, if you're curious about the nitty-gritty details!

4.4 Service De- and Constructors

Each service has a `start` procedure and a `stop` procedure, also referred to as its *constructor* and *destructor* (see Chapter 4 [Services], page 9). The procedures listed below return procedures that may be used as service constructors and destructors. They are flexible enough to cover most use cases and carefully written to complete in a timely fashion.

`make-forkexec-constructor` *command* [*#:user #f*] [*#:group #f*] [*#:supplementary-groups '()*] [*#:pid-file #f*] [*#:pid-file-timeout* (*default-pid-file-timeout*)] [*#:input-port #f*] [*#:log-file #f*] [*#:directory* (*default-service-directory*)] [*#:file-creation-mask #f*] [*#:create-session? #t*] [*#:resource-limits '()*] [*#:environment-variables* (*default-environment-variables*)] [Procedure]
Return a procedure that forks a child process, closes all file descriptors except the standard output and standard error descriptors, sets the current directory to *directory*, sets the umask to *file-creation-mask* unless it is `#f`, changes the environment to *environment-variables* (using the `environ` procedure), sets the current user to *user* the current group to *group* unless they are `#f` and supplementary groups to *supplementary-groups* unless they are `'()`, and executes *command* (a list of strings.) When *input-port* is set, it refers to a port that is used as the standard input of the created child process (`/dev/null` is used by default). When *create-session?* is true, the child process creates a new session with `setsid` and becomes its leader. The result of the procedure will be the `<process>` record representing the child process.

> **Note:** This will not work as expected if the process "daemonizes" (forks); in that case, you will need to pass `#:pid-file`, as explained below.

When *pid-file* is true, it must be the name of a PID file associated with the process being launched; the return value is the PID once that file has been created. If *pid-file* does not show up in less than *pid-file-timeout* seconds, the service is considered as failing to start.

When *log-file* is true, it names the file to which the service's standard output and standard error are redirected. *log-file* is created if it does not exist, otherwise it is appended to.

> **Note:** See Section 5.2 [Log Rotation Service], page 30, for a service to rotate log files specified via the `#:log-file` parameter.

Guile's `setrlimit` procedure is applied on the entries in *resource-limits*. For example, a valid value would be:

```
'((nproc 10 100)         ;number of processes
  (nofile 4096 4096))    ;number of open file descriptors
```

`make-kill-destructor` [*signal*] [*#:grace-period* (*default-process-termination-grace-period*)] [Procedure]

Return a procedure that sends *signal* to the process group of the PID given as argument, where *signal* defaults to `SIGTERM`. If the process is still running after *grace-period* seconds, send it `SIGKILL`. The procedure returns once the process has terminated.

This *does* work together with respawning services, because in that case the `stop` method of the `<service>` arranges so that the service is not respawned.

The `make-forkexec-constructor` procedure builds upon the following procedures.

`exec-command` *command* [*#:user #f*] [*#:group #f*] [*#:supplementary-groups '()*] [*#:log-file #f*] [*#:log-port #f*] [*#:input-port #f*] [*#:directory* (*default-service-directory*)] [*#:file-creation-mask #f*] [*#:create-session? #t*] [*#:resource-limits '()*] [*#:environment-variables* (*default-environment-variables*)] [Procedure]

`fork+exec-command` *command* [*#:user #f*] [*#:group #f*] [*#:supplementary-groups '()*] [*#:input-port #f*] [*#:log-file #f*] [*#:log-encoding "UTF-8"*] [*#:directory* (*default-service-directory*)] [*#:file-creation-mask #f*] [*#:create-session? #t*] [*#:resource-limits '()*] [*#:environment-variables* (*default-environment-variables*)] [Procedure]

Run *command* as the current process from *directory*, with *file-creation-mask* if it's true, with *rlimits*, and with *environment-variables* (a list of strings like `"PATH=/bin"`.) File descriptors 1 and 2 are kept as is or redirected to either *log-port* or *log-file* if it's true, whereas file descriptor 0 (standard input) points to *input-port* or `/dev/null`; all other file descriptors are closed prior to yielding control to *command*. When *create-session?* is true, call `setsid` first (see Section "Processes" in *GNU Guile Reference Manual*).

By default, *command* is run as the current user. If the *user* keyword argument is present and not false, change to *user* immediately before invoking *command*. *user* may be a string, indicating a user name, or a number, indicating a user ID. Likewise, *command* will be run under the current group, unless the *group* keyword argument is present and not false, and *supplementary-groups* is not `'()`.

`fork+exec-command` does the same as `exec-command`, but in a separate process whose PID it returns.

> **Warning:** The `shepherd`'s main even loop is responsible for calling `waitpid` to reap processes that have completed. User code *must not* call `waitpid` as that would compete with `waitpid` calls made by the main event loop, possibly leading to deadlocks.

Use `system*` or `spawn-command` when you need to spawn a command and wait for its exit status. Section 6.1 [Process Utilities], page 35.

`default-environment-variables` [Variable]
This parameter (see Section "Parameters" in *GNU Guile Reference Manual*) specifies the default list of environment variables to be defined when the procedures above create a new process.

It must be a list of strings where each string has the format *`name=value`*. It defaults to what `environ` returns when the program starts (see Section "Runtime Environment" in *GNU Guile Reference Manual*).

`user-environment-variables` [*name-or-id* (*getuid*)] [Procedure]
[*environment-variables* (*default-environment-variables*)]
Take the list *environment-variables*, replace `HOME` with home directory of the user and `USER` with the name of the user and return the result.

`default-pid-file-timeout` [Variable]
This parameter (see Section "Parameters" in *GNU Guile Reference Manual*) specifies the default PID file timeout in seconds, when `#:pid-file` is used (see above). It defaults to 5 seconds.

`default-process-termination-grace-period` [Variable]
This parameter (see Section "Parameters" in *GNU Guile Reference Manual*) specifies the "grace period" (in seconds) after which a process that has been sent `SIGTERM` or some other signal to gracefully exit is sent `SIGKILL` for immediate termination. It defaults to 5 seconds.

`default-respawn-delay` [Variable]
This parameter specifies the default value of the `#:respawn-delay` parameter of `service` (see Section 4.1 [Defining Services], page 9). It defaults to `0.1`, meaning a 100ms delay before respawning a service.

`default-respawn-limit` [Variable]
This parameter specifies the default value of the `#:respawn-limit` parameter of `service` (see Section 4.1 [Defining Services], page 9).

As an example, suppose you add this line to your configuration file:

```
(default-respawn-limit '(3 . 10))
```

The effect is that services will be respawned at most 3 times over a period of 10 seconds before being disabled.

One may also define services meant to be started *on demand*. In that case, shepherd listens for incoming connections on behalf of the program that handles them; when it accepts an incoming connection, it starts the program to handle them. The main benefit is that such services do not consume resources until they are actually used, and they do not slow down startup.

These services are implemented following the protocol of the venerable inetd "super server" (see Section "inetd invocation" in *GNU Inetutils*). Many network daemons can be invoked in "inetd mode"; this is the case, for instance, of `sshd`, the secure shell server of

the OpenSSH project. The Shepherd lets you define inetd-style services, specifically those in `nowait` mode where the daemon is passed the newly-accepted socket connection while `shepherd` is in charge of listening.

Listening endpoints for such services are described as records built using the `endpoint` procedure provided by the `(shepherd endpoints)` module:

`endpoint` *address* [*#:name* "*unknown*"] [*#:style SOCK_STREAM*] [*#:backlog 128*] [*#:socket-owner* (*getuid*)] [*#:socket-group* (*getgid*)] [*#:socket-directory-permissions #o755*] [*#:bind-attempts* (*default-bind-attempts*)] [Procedure]

Return a new endpoint called *name* of *address*, an address as return by `make-socket-address`, with the given *style* and *backlog*.

When *address* is of type `AF_INET6`, the endpoint is *IPv6-only*. Thus, if you want a service available both on IPv4 and IPv6, you need two endpoints. For example, below is a list of endpoints to listen on port 4444 on all the network interfaces, both in IPv4 and IPv6 ("0.0.0.0" for IPv4 and "::0" for IPv6):

```
(list (endpoint (make-socket-address AF_INET INADDR_ANY 4444))
      (endpoint (make-socket-address AF_INET6 IN6ADDR_ANY 4444)))
```

This is the list you would pass to `make-inetd-constructor` or `make-systemd-constructor`—see below.

When *address* is of type `AF_UNIX`, *socket-owner* and *socket-group* are strings or integers that specify its ownership and that of its parent directory; *socket-directory-permissions* specifies the permissions for its parent directory.

Upon '`EADDRINUSE`' ("Address already in use"), up to *bind-attempts* attempts will be made to `bind` on *address*, one every second.

`default-bind-attempts` [Variable]

This parameter specifies the number of times, by default, that `shepherd` will try to bind an endpoint address if it happens to be already in use.

The inetd service constructor takes a command and a list of such endpoints:

`make-inetd-constructor` *command endpoints* [*#:service-name-stem _*] [*#:requirements '()*] [*#:max-connections* (*default-inetd-max-connections*)] [*#:user #f*] [*#:group #f*] [*#:supplementary-groups '()*] [*#:directory* (*default-service-directory*)] [*#:file-creation-mask #f*] [*#:create-session? #t*] [*#:resource-limits '()*] [*#:environment-variables* (*default-environment-variables*)] [Procedure]

Return a procedure that opens sockets listening to *endpoints*, a list of objects as returned by `endpoint`, and accepting connections in the background.

Upon a client connection, a transient service running *command* is spawned. Only up to *max-connections* simultaneous connections are accepted; when that threshold is reached, new connections are immediately closed.

The remaining arguments are as for `make-forkexec-constructor`.

`make-inetd-destructor` [Procedure]

Return a procedure that terminates an inetd service.

The last type is *systemd-style services*. Like inetd-style services, those are started on demand when an incoming connection arrives, but using the protocol devised by the systemd service manager and referred to as *socket activation* (`https://www.freedesktop.org/software/systemd/man/daemon.html#Socket-Based%20Activation`). The main difference with inetd-style services is that shepherd hands over the listening socket(s) to the daemon; the daemon is then responsible for accepting incoming connections. A handful of environment variables are set in the daemon's execution environment (see below), which usually checks them using the libsystemd or libelogind client library helper functions (`https://www.freedesktop.org/software/systemd/man/sd_listen_fds.html`). The constructor and destructor for systemd-style daemons are described below.

`make-systemd-constructor` *command endpoints* [*#:lazy-start? #t*] [*#:user #f*] [*#:group #f*] [*#:supplementary-groups '()*] [*#:log-file #f*] [*#:directory* (*default-service-directory*)] [*#:file-creation-mask #f*] [*#:create-session? #t*] [*#:resource-limits '()*] [*#:environment-variables* (*default-environment-variables*)] [Procedure]

Return a procedure that starts *command*, a program and list of argument, as a systemd-style service listening on *endpoints*, a list of `<endpoint>` objects.

command is started on demand on the first connection attempt on one of *endpoints* when *lazy-start?* is true; otherwise it is started as soon as possible. It is passed the listening sockets for *endpoints* in file descriptors 3 and above; as such, it is equivalent to an `Accept=no` systemd socket unit (`https://www.freedesktop.org/software/systemd/man/systemd.socket.html`). The following environment variables are set in its environment:

`LISTEN_PID`
: It is set to the PID of the newly spawned process.

`LISTEN_FDS`
: It contains the number of sockets available starting from file descriptor 3—i.e., the length of *endpoints*.

`LISTEN_FDNAMES`
: The colon-separated list of endpoint names.

This must be paired with `make-systemd-destructor`.

`make-systemd-destructor` [Procedure]

Return a procedure that terminates a systemd-style service as created by `make-systemd-constructor`.

The following constructor/destructor pair is also available for your convenience, but we recommend using `make-forkexec-constructor` and `make-kill-destructor` instead (this is typically more robust than going through the shell):

`make-system-constructor` *command...* [Procedure]

The returned procedure will execute *command* in a shell and return `#t` if execution was successful, otherwise `#f`. For convenience, it takes multiple arguments which will be concatenated first.

`make-system-destructor` *`command...`* [Procedure]
Similar to `make-system-constructor`, but returns `#f` if execution of the *command* was successful, `#t` if not.

See Section 4.5 [Timers], page 20, for other service constructors and destructors: those for timers.

4.5 Timers

In addition to process, inetd-style, and systemd-style services discussed before, another type of service is available: *timers*.

Timers, you guessed it, execute a command or call a procedure periodically. They work like other services: they can be started, stopped, unloaded, and so on. The constructor and destructor shown below let you define a timer; they are exported by `(shepherd service timer)`, so make sure to add a line like this to your configuration file if you'd like to use them:

```
(use-modules (shepherd service timer))
```

Timers fire on calendar events: "every Sunday at noon", "everyday at 11AM and 8PM", "on the first of each month", etc. If you ever used cron, mcron, or systemd timers, this is similar. Calendar events are specified using the `calendar-event` procedure, which defines *constraints* on each calendar component: months, days of week, hours, minutes, and so on. Here are a few examples:

```
;; Every Sunday and every Wednesday, at noon.
(calendar-event #:hours '(12) #:minutes '(0)
                #:days-of-week '(sunday wednesday))

;; Everyday at 11AM and 8PM.
(calendar-event #:hours '(11 20) #:minutes '(0))

;; At 9:10AM on the first and fifteenth of each month.
(calendar-event #:days-of-month '(1 15)
                #:hours '(9) #:minutes '(10))
```

`calendar-event` [*#:months any-month*] [*#:days-of-month any-day*] [*#:days-of-week any-day-of-week*] [*#:hours any-hour*] [*#:minutes any-minute*] [*#:seconds '(0)*] [Procedure]
Return a calendar event that obeys the given constraints. Raise an error if one of the values is out of range.

All the arguments are lists of integers as commonly used in the Gregorian calendar, except for `days-of-week` which is a list of symbols denoting weekdays: `'monday`, `'tuesday`, etc.

> **Note:** All calendar events are understood in the timezone they occur in, should daylight saving time (DST) changes occur between consecutive events.
>
> For example, in Western and Central Europe, the waiting time between two consecutive daily events is 24 hours except on two occasions: it is 23

hours when switching from "normal" time (CET, or UTC+1) to summer time (CEST, or UTC+2), and it is 25 hours when switching from summer time to "normal" time.

Users familiar with the venerable Vixie cron can instead convert cron-style date specifications to a calendar event data structure using the `cron-string->calendar-event` procedure described below.

`cron-string->calendar-event` *str* [Procedure]

Convert *str*, which contains a Vixie cron date line, into the corresponding `calendar-event`. Raise an error if *str* is invalid.

A valid cron date line consists of 5 space-separated fields: minute, hour, day of month, month, and day of week. Each field can be an integer, or a comma-separate list of integers, or a range. Ranges are represented by two integers separated by a hyphen, optionally followed by slash and a number of repetitions. Here are examples:

`30 4 1,15 * *`
: 4:30AM on the 1st and 15th of each month;

`5 0 * * *`
: five minutes after midnight, every day;

`23 0-23/2 * * 1-5`
: 23 minutes after the hour every two hour, on weekdays.

To create a timer, you create a service with the procedures described below as its `start` and `stop` methods (see Section 4.1 [Defining Services], page 9).

`make-timer-constructor` *event* `action` [*#:occurrences +inf.0*] [*#:log-file #f*] [*#:max-duration #f*] [*#:wait-for-termination? #f*] [Procedure]

Return a procedure for use as the `start` method of a service. The procedure will perform *action* for *occurrences* iterations of `event`, a calendar event as returned by `calendar-event`. *action* may be either a command (returned by `command`) or a thunk; in the latter case, the thunk must be suspendable or it could block the whole shepherd process.

When *log-file* is true, log the output of *action* to that file rather than in the global shepherd log.

When *wait-for-termination?* is true, wait until *action* has finished before considering executing it again; otherwise, perform *action* strictly on every occurrence of *event*, at the risk of having multiple instances running concurrently.

When *max-duration* is true, it is the maximum duration in seconds that a run may last, provided *action* is a command. Past *max-duration* seconds, the timer's process is forcefully terminated with signal *termination-signal*.

`make-timer-destructor` [Procedure]

Return a procedure for the `stop` method of a service whose constructor was given by `make-timer-constructor`.

As we have seen above, `make-timer-constructor` can be passed a command to execute. Those are specified using the `command` procedure below.

`command` *line* [*#:user #f*] [*#:group #f*] [*#:supplementary-groups '()*] [*#:environment-variables (default-environment-variables)*] [*#:directory (default-service-directory)*] [*#:resource-limits '()*] [Procedure]

Return a new command for *line*, a program name and argument list, to be executed as *user* and *group*, with the given *environment-variables*, in *directory*, and with the given *resource-limits*.

These arguments are the same as for `fork+exec-command` and related procedures (see [exec-command], page 16).

It's also possible to add a `trigger` action to timer services, such that one can trigger it with:

```
herd trigger service
```

To do that, you would add the predefined `timer-trigger-action` to the service's `actions` field (see Section 4.1 [Defining Services], page 9).

`timer-trigger-action` [Variable]

This is the `trigger` service action. When invoked, its effect is to invoke the action passed to `make-timer-constructor`.

See [timer-example], page 26, to see how to put it all together.

Last, this module also provides `timer-service`, a sort of "meta" timer service that lets you dynamically create timed services from the command line. See Section 5.4 [Timer Service], page 32.

4.6 The `root` Service

The service `root` is special, because it is used to control the Shepherd itself. It has an alias `shepherd`. It provides the following actions (in addition to `enable`, `disable` and `restart` which do not make sense here).

`status` Displays which services are started and which ones are not.

`detailed-status`
Displays detailed information about every registered service.

`load` *file* Evaluate in the `shepherd` process the Scheme code in *file*, in a fresh module that uses the `(shepherd services)` module—as with the `--config` option of `shepherd` (see Section 3.1 [Invoking shepherd], page 4).

`eval` *exp* Likewise, evaluate Scheme expression *exp* in a fresh module with all the necessary bindings. Here is a couple of examples:

```
# herd eval root "(+ 2 2)"
4
# herd eval root "(getpid)"
1
# herd eval root "(lookup-running 'xorg-server)"
(service (version 0) (provides (xorg-server)) ...)
```

`unload` *service-name*
Attempt to remove the service identified by *service-name*. `shepherd` will first stop the service, if necessary, and then remove it from the list of registered

services. Any services depending upon *service-name* will be stopped as part of this process.

If *service-name* simply does not exist, output a warning and do nothing. If it exists, but is provided by several services, output a warning and do nothing. This latter case might occur for instance with the fictional service `web-server`, which might be provided by both `apache` and `nginx`. If *service-name* is the special string and `all`, attempt to remove all services except for the Shepherd itself.

`reload` *`file-name`*
: Unload all known optional services using unload's `all` option, then load *file-name* using `load` functionality. If file-name does not exist or `load` encounters an error, you may end up with no defined services. As these can be reloaded at a later stage this is not considered a problem. If the `unload` stage fails, `reload` will not attempt to load *file-name*.

`daemonize`
: Fork and go into the background. This should be called before respawnable services are started, as otherwise we would not get the `SIGCHLD` signals when they terminate.

`kexec`
: On GNU/Linux, reboot straight into a new Linux kernel previously loaded with the `kexec -l` *`file`* command. This is the action invoked by the `reboot -k` command. See Section 3.3 [Invoking reboot], page 7.

4.7 Legacy GOOPS Interface

From its inception in 2002 with negative version numbers (really!) up to version 0.9.x included, the Shepherd's service interface used GOOPS, Guile's object-oriented programming system (see Section "GOOPS" in *GNU Guile Reference Manual*). This interface was deprecated in 0.10.0 and removed in 1.0.0. The remainder of this section describes that interface and how to migrate from it.

The GOOPSy interface provided a `<service>` class whose instances you could access directly with `slot-ref`; generic functions such as `start` and `stop` had one method accepting a service object and another method accepting the name of a service as a symbol, which it would transparently resolve.

Fortunately, common idioms are easily converted to the current interface. For example, you would previously create a service like so:

```
(make <service>       ;former GOOPS interface
  #:provides '(something)
  #:requires '(another thing)
  #:start ...
  #:stop ...
  #:respawn? #t)
```

With the new interface (see Section 4.1 [Defining Services], page 9), you would write something very similar:

```
(service '(something)
  #:requirement '(another thing)
```

```
  #:start ...
  #:stop ...
  #:respawn? #t)
```

Likewise, instead of writing:

```
(start 'whatever)
```

... you would write:

```
(start-service (lookup-service 'whatever))
```

If in doubt, please get in touch with us at `help-guix@gnu.org`.

4.8 Service Examples

The configuration file of the `shepherd` command (see Section 3.1 [Invoking shepherd], page 4) defines, registers, and possibly starts *services*. Each service specifies other services it depends on and how it is started and stopped. The configuration file contains Scheme code that uses the programming interface of the `(shepherd service)` module (see Chapter 4 [Services], page 9).

Let's assume you want to define and register a service to start mcron, the daemon that periodically executes jobs in the background (see Section "Introduction" in *GNU mcron Manual*). That service is started by invoking the `mcron` command, after which `shepherd` should monitor the running process, possibly re-spawning it if it stops unexpectedly. Here's the configuration file for this one service:

```
(define mcron
  (service
    '(mcron)
    ;; Run /usr/bin/mcron without any command-line arguments.
    #:start (make-forkexec-constructor '("/usr/bin/mcron"))
    #:stop (make-kill-destructor)
    #:respawn? #t))

(register-services (list mcron))
```

You can write the snippet above in the default configuration file—`~/.config/shepherd/init.scm` if you run `shepherd` as an unprivileged user. When you launch it, `shepherd` will evaluate that configuration; thus it will define and register the `mcron` service, but it will *not* start it. To start the service, run:

```
herd start mcron
```

Alternatively, if you want `mcron` to be started automatically when `shepherd` starts, you can add this snippet at the end of the configuration file:

```
(start-in-the-background '(mcron))
```

Now let's take another example: `sshd`, the secure shell daemon of the OpenSSH project (`https://www.openssh.com`). We will pass `sshd` the `-D` option so that it does not "detach", making it easy for `shepherd` to monitor its process; we also tell `shepherd` to check its *PID file* to determine once it has started and is ready to accept connections:

```
(define sshd
  (service
```

```
    '(sshd ssh-daemon)  ;for convenience, give it two names
    #:start (make-forkexec-constructor
             '("/usr/sbin/sshd" "-D")
             #:pid-file "/etc/ssh/sshd.pid")
    #:stop (make-kill-destructor)
    #:respawn? #t))

(register-services (list sshd))
(start-in-the-background '(sshd))
```

Alternatively, we can start `sshd` in *inetd mode*: in that case, `shepherd` listens for connection and spawns `sshd` only upon incoming connections. The inetd mode is enabled by passing the `-i` command-line option:

```
(define sshd
  (service
    '(sshd ssh-daemon)
    #:start (make-inetd-constructor
             '("/usr/sbin/sshd" "-D" "-i")
             (list (endpoint
                    (make-socket-address AF_INET INADDR_ANY 22))
                   (endpoint
                    (make-socket-address AF_INET6 IN6ADDR_ANY 22)))
             #:max-connections 10)
    #:stop (make-inetd-destructor)
    #:respawn? #t))

(register-services (list sshd))
(start-in-the-background '(sshd))
```

The `make-socket-address` procedure calls above return the *listening addresses* (see Section "Network Socket Address" in *GNU Guile Reference Manual*). In this case, it specifies that `shepherd` will accept connections coming from any network interface ("0.0.0.0" in IPv4 notation and "::0" for IPv6) on port 22. The `endpoint` calls wrap these addresses in endpoint records (see [endpoints], page 18). When a client connects, `shepherd` accepts it and spawns `sshd -D -i` as a new *transient service*, passing it the client connection. The `#:max-connections` parameter instructs `shepherd` to accept at most 10 simultaneous client connections.

The example below—a service for the nginx (`https://nginx.org`) web server—illustrates a situation where a service constructor and destructor work by invoking a program with `system*` instead of the usual `make-forkexec-constructor` and `make-kill-destructor`.

```
(service
  '(nginx)
  #:start (lambda ()
            ;; The 'nginx' command daemonizes and returns immediately;
            ;; it eventually writes a PID file once successfully started.
            (and (zero? (system* "/usr/sbin/nginx"))
                 (read-pid-file "/var/run/nginx/pid")))
```

```
  #:stop (lambda _
           ;; Stop the nginx daemon by running this command.
           (unless (zero? (system* "/usr/sbin/nginx" "-s" "stop"))
             (error "failed to stop nginx"))
           #f))
```

See Section 6.1 [Process Utilities], page 35, for more on `system*` and related process helpers.

Let's now look at *timers*—services that run periodically, on chosen calendar events (see Section 4.5 [Timers], page 20). If you ever used systemd timers or the venerable cron, this is similar. The example below defines a timer that fires twice a day and runs the `updatedb` command as root (see Section "Invoking updatedb" in *Finding Files*):

```
;; Import the module for 'make-timer-constructor' & co.
(use-modules (shepherd service timer))

(define updatedb
  (service
    '(updatedb)
    #:start (make-timer-constructor
              ;; Fire at midnight and noon everyday.
              (calendar-event #:hours '(0 12) #:minutes '(0))
              (command '("/usr/bin/updatedb"
                         "--prunepaths=/tmp")))
    #:stop (make-timer-destructor)
    #:actions (list timer-trigger-action)))

(register-services (list updatedb))
(start-in-the-background '(updatedb))
```

Thanks to the `#:actions` bit above, you can also run `herd trigger updatedb` to trigger that job.

In these examples, we haven't discussed dependencies among services—the `#:requires` keyword of `<service>`—nor did we discuss systemd-style services. These are extensions of what we've seen so far. See Chapter 4 [Services], page 9, for details.

If you use Guix System, you will see that it contains a wealth of Shepherd service definitions. The nice thing is that those give you a *complete view* of what goes into the service—not just how the service is started and stopped, but also what software package is used and what configuration file is provided. See Section "Shepherd Services" in *GNU Guix Reference Manual*, for more info.

4.9 Managing User Services

The Shepherd can be used to manage services for an unprivileged user. First, you may want to ensure it is up and running every time you log in. One way to accomplish that is by adding the following lines to `~/.bash_profile` (see Section "Bash Startup Files" in *The GNU Bash Reference Manual*):

```
if [[ ! -S ${XDG_RUNTIME_DIR-$HOME/.cache}/shepherd/socket ]]; then
    shepherd
```

```
fi
```

Then, we suggest the following top-level `$XDG_CONFIG_HOME/shepherd/init.scm` file, which will automatically load individual service definitions from `~/.config/shepherd/init.d`:

```
(use-modules (shepherd service)
             ((ice-9 ftw) #:select (scandir)))

;; Send shepherd into the background
(perform-service-action root-service 'daemonize)

;; Load all the files in the directory 'init.d' with a suffix '.scm'.
(for-each
  (lambda (file)
    (load (string-append "init.d/" file)))
  (scandir (string-append (dirname (current-filename)) "/init.d")
           (lambda (file)
             (string-suffix? ".scm" file))))
```

Then, individual user services can be put in `$XDG_CONFIG_HOME/shepherd/init.d/`, e.g., for `ssh-agent`.

```
;; Add to your ~/.bash_profile:
;;
;; SSH_AUTH_SOCK=${XDG_RUNTIME_DIR-$HOME/.cache}/ssh-agent/socket
;; export SSH_AUTH_SOCK

(use-modules (shepherd support))

(define ssh-agent
  (service
    '(ssh-agent)
    #:documentation "Run `ssh-agent'"
    #:start (lambda ()
              (let ((socket-dir (string-append %user-runtime-dir "/ssh-agent")))
                (unless (file-exists? socket-dir)
                  (mkdir-p socket-dir)
                  (chmod socket-dir #o700))
                (fork+exec-command
                 `("ssh-agent" "-D" "-a" ,(string-append socket-dir "/socket"))
                 #:log-file (string-append %user-log-dir "/ssh-agent.log"))))
    #:stop (make-kill-destructor)
    #:respawn? #t))

(register-services (list ssh-agent))
(start-service ssh-agent)
```

5 Service Collection

The Shepherd comes with a collection of services that let you control it or otherwise extend its functionality. This chapter documents them.

5.1 System Log Service

Traditionally, GNU and Unix-like systems come with a system-wide logging mechanism called *syslog* that applications and services can use. This mechanism usually relies on a standalone process, the `syslogd` daemon, that listens for incoming messages, typically on the `/dev/log` Unix-domain socket, and writes them to files or terminals according to user settings (see Section "syslogd invocation" in *GNU Inetutils*). Each message sent to `syslogd` specifies its *priority* and originating *facility* along with the actual message (see Section "Overview of Syslog" in *The GNU C Library Reference Manual*).

The Shepherd provides an optional in-process service that can be used as a substitute for the traditional `syslogd` program. The configuration snippet below instantiates and registers that service for `shepherd` running as PID 1:

```
(use-modules (shepherd service system-log))

(register-services
  ;; Create a system log with the default settings.
  (list (system-log-service)))

;; Start it.
(start-service (lookup-service 'system-log))
```

The configuration above starts the service under the names `system-log` and `syslogd`. Messages are logged in files under `/var/log`, on `/dev/tty12` (the twelfth console terminal on Linux), and on the console for emergency messages. The `system-log` service is integrated with the log rotation service: log files it creates are subject to log rotation, in a "race-free" fashion (see Section 5.2 [Log Rotation Service], page 30).

The destination of syslog messages—the files, terminals, etc. where they are written—can be configured by passing a procedure as the `#:message-destination` argument of `system-log-service`:

```
(define (message-destination message)
  ;; Return the list of files where MESSAGE is to be written.
  (cond ((member (system-log-message-priority message)
                 (list (system-log-priority emergency)
                       (system-log-priority alert)))
         ;; Important messages go both to a file and to the console.
         '("/dev/console" "/var/log/messages))
        ((string-prefix? "sudo" (system-log-message-content message))
         ;; Messages coming from sudo go to a dedicated file.
         '("/var/log/sudo"))
        ((member (system-log-message-facility message)
                 (list (system-log-facility authorization)
                       (system-log-facility authorization/private)))
```

```
             ;; Security-sensitive messages are written only to this
             ;; one file.
             '("/var/log/secure"))
            (else
             ;; Everything else goes to /var/log/messages.
             '("/var/log/messages"))))

(define my-syslogd
  ;; Customized system log instance.
  (system-log-service #:message-destination message-destination))
```

The example above shows how to assign different destinations based on message priority, facility, and content. Note that each message can be written to zero, one, or more files.

The interface to the system log service is provided by the `(shepherd service system-log)` module and described thereafter.

`system-log-message?` *obj* [Procedure]
Return true if *obj* is a system log message.

`system-log-message-facility` *message* [Procedure]
`system-log-message-priority` *message* [Procedure]
`system-log-message-content` *message* [Procedure]
`system-log-message-sender` *message* [Procedure]
Return the given attribute of *message*, a system log message:

- Its facility, an integer as constructed by `system-log-facility`. For example, the expression `(system-log-facility mail)` evaluates to the value of the "mail" facility.
- Its priority, an integer as constructed by `system-log-priority`. For example, `(system-log-priority debug)` returns the "debug" priority.
- Its content, a string.
- Its sender, either `#f`, in which case the message originates from the local host, or a socket address as constructed by `make-socket-address` if the message originates from a different host (see below).

`system-log-service` [*sources*] [*#:provision '(system-log syslogd)*] [Procedure]
[*#:requirement '()*] [*#:kernel-log-file (kernel-log-file)*]
[*#:message-destination (default-message-destination-procedure)*]
[*#:date-format default-logfile-date-format*] [*#:history-size (default-log-history-size)*] [*#:max-silent-time (default-max-silent-time)*]
Return the system log service (*syslogd*) with the given *provision* and *requirement* (lists of symbols). The service accepts connections on *sources*, a list of `<endpoint>` objects; optionally it also reads messages from `#:kernel-log-file`, which defaults to `/proc/kmsg` (Linux) or `/dev/klog` (the Hurd) when running as root.

Log messages are passed to *message-destination*, a one-argument procedure that must return the list of files to write it to. Write a mark to log files when no message has been logged for more than *max-silent-time* seconds (this feature is disabled by setting

max-silent-time to `#f`). Timestamps in log files are formatted according to *date-format*, a format string for `strftime`, including delimiting space—e.g., `\"%c \"` for a format identical to that of traditional syslogd implementations.

Keep up to *history-size* messages in memory for the purposes of allowing users to view recent messages without opening various files.

`default-message-destination-procedure` [Procedure]
Return a procedure that, given a system log message, returns a "good" default destination to log it to.

`kernel-log-file` [Variable]
This SRFI-39 parameter denotes the location of the kernel log file, by default `/proc/kmsg` on Linux and `/dev/klog` on the Hurd.

`default-max-silent-time` [Variable]
This SRFI-39 parameter denotes the maximum number of seconds after which syslogd prints a "mark" in its files if it has not printed anything for this long.

5.2 Log Rotation Service

All these services produce many log files; they're useful, for sure, but wouldn't it be nice to archive them or even delete them periodically? The *log rotation* service does exactly that. Once you've enabled it, it periodically *rotates* the log files of services—those specified via the `#:log-file` argument of service constructors (see Section 4.4 [Service De- and Constructors], page 15)—those of the system log (see Section 5.1 [System Log Service], page 28), and also, optionally, "external" log files produced by some other mechanism.

By "rotating" we mean this: if a service produces `/var/log/my-service.log`, then rotating it consists in periodically renaming it and compressing it to obtain, say, `/var/log/my-service.log.1.gz`—after renaming the *previous* `my-service.1.log.gz` to `my-service.log.2.gz`, and so on[1]. Files older than some configured threshold are deleted instead of being renamed. The process is race-free: if the service is running, not a single line that it logs is lost during rotation.

To enable the log rotation service, you can add the following lines to your configuration file (see Section 4.8 [Service Examples], page 24):

```
(use-modules (shepherd service log-rotation))

(register-services
  ;; Create a service that rotates log files once a week.
  (list (log-rotation-service)))

;; Start it.
(start-service (lookup-service 'log-rotation))
```

This creates a `log-rotation` service, which is in fact a timed service (see Section 4.5 [Timers], page 20). By default it rotates logs once a week and you can see past and upcoming runs in the usual way:

```
herd status log-rotation
```

[1] This is comparable to what the venerable logrotate tool would do.

You can also trigger it explicitly at any time, like so:

```
herd trigger log-rotation
```

Last, you can list log files subject to rotation:

```
herd files log-rotation
```

The default settings should be good for most use cases, but you can change them by passing the `log-rotation-service` procedure a number of arguments—see the reference documentation below.

`log-rotation-service` [*event*] [*#:provision '(log-rotation)*] [*#:requirement '()*] [*#:external-log-files '()*] [*#:compression (%default-log-compression)*] [*#:expiry (%default-log-expiry)*] [*#:rotation-size-threshold (%default-rotation-size-threshold)*] [Procedure]

Return a timed service that rotates service logs along with *external-log-files* (a list of file names such as `/var/log/nginx/access.log` corresponding to "external" log files not passed as `#:log-file` to any service) on every occurrence of *event*, a calendar event.

Compress log files according to *method*, which can be one of `'gzip`, `'zstd`, `'none`, or a one-argument procedure that is passed the file name. Log files smaller than *rotation-size-threshold* are not rotated; copies older than *expiry* seconds are deleted.

Last, *provision* and *requirement* are lists of symbols specifying what the service provides and requires, respectively. Specifying *requirement* is useful to ensure, for example, that log rotation runs only if the service that mounts the file system that hosts log files is up.

5.3 Transient Service Maker

The *transient service maker* lets you run commands in the background, and it does so by wrapping those commands in transient services (see Section 4.1 [Defining Services], page 9). It is similar to the `systemd-run` (`https://www.freedesktop.org/software/systemd/man/latest/systemd-run.html`) command, which you might have encountered before.

A simple configuration file that uses this service looks like this:

```
(use-modules (shepherd service transient))

(register-services (list (transient-service)))
```

This creates a service called `transient` that has a `spawn` action, which you can use like this:

```
# Run 'make' from the current directory.
herd spawn transient -d "$PWD" -- make -j4

# Run 'rsync' from the home directory, inheriting
# the 'SSH_AUTH_SOCK' environment variable.
herd spawn transient \
  --log-file=backup.log \
  -E SSH_AUTH_SOCK=$SSH_AUTH_SOCK -- \
```

```
rsync -e ssh -vur . backup.example.org:
```

Each of these `herd spawn transient` commands creates a new transient service. Like any other service, they can be inspected and stopped. Running `herd stop transient` stops all the currently running transients.

The command runs from the directory specified by `default-service-directory` or from that specified by the `--working-directory` option of `herd`; it starts with the environment variables in `default-environment-variables`, augmented with `HOME` when running as a different user, with the addition of variables passed with `--environment-variable`. See Section 3.2 [Invoking herd], page 6, for more info on influential command-line options.

`transient-service` [*provision*] [*#:requirement '()*] [Procedure]
Return a service with the given *provision* and *requirement*. The service has a `spawn` action that lets users run commands in the background.

5.4 Timer Service

The `(shepherd service timer)` provides the `timer` service, which lets you dynamically create timed services (see Section 4.5 [Timers], page 20), from the command line, in a way similar to the traditional `at` command:

```
# Run the 'mail' command, as root, as 12PM.
herd schedule timer at 12:00 -- \
  mail --to=charlie -s "Lunch time!"

# Run the 'mpg123' command as user 'charlie' at 7AM, from charlie's
# home directory.
herd schedule timer at 07:00 --user=charlie -- \
  mpg123 Music/alarm.mp3

# Run 'rsync' from the ~/doc directory, inheriting the 'SSH_AUTH_SOCK'
# environment variable.  Log the output in 'backup.log'.
herd schedule timer at 13:00 \
  --log-file=backup.log \
  -d ~/doc -E SSH_AUTH_SOCK=$SSH_AUTH_SOCK -- \
  rsync -e ssh -vur . backup.example.org:
```

Each of these `herd schedule timer` commands creates a new timed service, which, like any other service, can be inspected and stopped; those services are transient and vanish after they have executed their command (see Section 4.1 [Defining Services], page 9).

The command runs from the directory specified by `default-service-directory` or from that specified by the `--working-directory` option of `herd`; it has with the environment variables in `default-environment-variables`, augmented with `HOME` when running as a different user, with the addition of variables passed with `--environment-variable`. See Section 3.2 [Invoking herd], page 6, for more info on influential command-line options.

This `timer` service can be added to your configuration like so:

```
(use-modules (shepherd service timer))

(register-services (list (timer-service)))
```

The reference follows.

`timer-service` [*provision*] [*#:requirement '*()] [Procedure]
Return a timer service with the given *provision* and *requirement*. The service has a `schedule` action that lets users schedule command execution similar to the venerable `at` command.

5.5 Monitoring Service

The *monitoring service*, as its name suggests, monitors resource usage of the shepherd daemon. It does so by periodically logging information about key resources: heap size (memory usage), open file descriptors, and so on. It is a simple and useful way to check whether resource usage remains under control.

To use it, a simple configuration file that uses this service and nothing else would look like this:

```
(use-modules (shepherd service monitoring))

(register-services
  ;; Create a monitoring service that logs every 15 minutes.
  (list (monitoring-service #:period (* 15 60))))

;; Start it!
(start-service (lookup-service 'monitoring))
```

Using the `herd` command, you can get immediate resource usage logging:

```
$ herd log monitoring
service names: 3; heap: 8.77 MiB; file descriptors: 20
```

You can also change the logging period; for instance, here is how you'd change it to 30 minutes:

```
$ herd period monitoring 30
```

The `(shepherd service monitoring)` module exports the following bindings:

`monitoring-service` [*#:period* (*default-monitoring-period*)] [Procedure]
Return a service that will monitor shepherd resource usage by printing it every *period* seconds.

`default-monitoring-period` [Variable]
This parameter specifies the default monitoring period, in seconds.

5.6 Read-Eval-Print Loop Service

Scheme wouldn't be Scheme without support for *live hacking*, and your favorite service manager had to support it too! The *REPL service* provides a read-eval-print loop (REPL) that lets you interact with it from the comfort of the Guile REPL (see Section "Running Guile Interactively" in *GNU Guile Reference Manual*).

The service listens for connections on a Unix-domain socket—by default `/var/run/shepherd/repl` when running as root and `/run/user/`*uid*`/shepherd/repl` otherwise—and spawns a new service for each client connection. Clients can use the REPL

as they would do with a "normal" REPL, except that it lets them inspect and modify the state of the `shepherd` process itself.

> **Caveat:** The live REPL is a powerful tool in support of live hacking and debugging, but it's also a dangerous one: depending on the code you execute, you could lock the `shepherd` process, make it crash, or who knows what.
>
> One particular aspect to keep in mind is that `shepherd` currently uses Fibers in such a way that scheduling among fibers is cooperative and non-preemptive. Beware!

A configuration file that enables the REPL service looks like this:

```
(use-modules (shepherd service repl))

(register-services (list (repl-service)))
```

With that in place, you can later start the REPL:

```
herd start repl
```

From there you can connect to the REPL socket. If you use Emacs, you might fancy doing it with Geiser's `geiser-connect-local` function (see *Geiser User Manual*).

The `(shepherd service repl)` module exports the following bindings.

`repl-service` [*socket-file*] [Procedure]
: Return a REPL service that listens to *socket-file*.

`default-repl-socket-file` [Variable]
: This parameter specifies the socket file name `repl-service` uses by default.

6 Misc Facilities

This is a list of facilities which are available to code running inside of the Shepherd and is considered generally useful, but is not directly related to one of the other topic covered in this manual.

6.1 Process Utilities

When writing custom service actions, you may find it useful to spawn helper processes. The `(shepherd service)` module provides helpers that are safe to use within the `shepherd` process. See [system-star-example], page 25, for a typical use case.

`system` *command* [Procedure]
`system*` *command arguments ...* [Procedure]
Run *command* and return its exit status, which can be examined with `status:exit-val` and related procedures (see Section "Processes" in *GNU Guile Reference Manual*).

This works exactly like the `system` and `system*` procedures provided by Guile but the `shepherd` process redefines them so they cooperate with its own even loop and child process termination handling.

`spawn-command` *command* [*#:user #f*] [*#:group #f*] [Procedure]
[*#:environment-variables* (*default-environment-variables*)] [*#:directory* (*default-service-directory*)] [*#:resource-limits* '()] [*#:log-file #f*]
Like `system*`, spawn *command* (a list of strings) and return its exit status. Additionally accept the same keyword arguments as `fork+exec-command`: `#:user`, `#:directory`, and so on. [exec-command], page 16.

6.2 Errors

`assert` *expr* [macro]
If *expr* yields `#f`, display an appropriate error message and throw an `assertion-failed` exception.

`without-system-error` *expr...* [macro]
Evaluates the *expr*s, not going further if a system error occurs, but also doing nothing about it.

6.3 Communication

The `(shepherd comm)` module provides primitives that allow clients such as `herd` to connect to `shepherd` and send it commands to control or change its behavior (see Section 4.1 [Defining Services], page 9).

Currently, clients may only send *commands*, represented by the `<shepherd-command>` type. Each command specifies a service it applies to, an action name, a list of strings to be used as arguments, and a working directory. Commands are instantiated with `shepherd-command`:

`shepherd-command` *action service* [*#:arguments* '()] [Procedure]
[*#:directory* (*getcwd*)]
Return a new command (a `<shepherd-command>`) object for *action* on *service*.

Commands may then be written to or read from a communication channel with the following procedures:

`write-command` *command port* [Procedure]
Write *command* to *port*.

`read-command` *port* [Procedure]
Receive a command from *port* and return it.

In practice, communication with `shepherd` takes place over a Unix-domain socket, as discussed earlier (see Section 3.1 [Invoking shepherd], page 4). Clients may open a connection with the procedure below.

`open-connection` [*file*] [Procedure]
Open a connection to the daemon, using the Unix-domain socket at *file*, and return the socket.

When *file* is omitted, the default socket is used.

The daemon writes output to be logged or passed to the currently-connected client using `local-output`:

`local-output` *format-string . args* [Procedure]
This procedure should be used for all output operations in the Shepherd. It outputs the *args* according to the *format-string*, then inserts a newline. It writes to whatever is the main output target of the Shepherd, which might be multiple at the same time in future versions.

Under the hood, `write-command` and `read-command` write/read commands as s-expressions (sexps). Each sexp is intelligible and specifies a protocol version. The idea is that users can write their own clients rather than having to invoke `herd`. For instance, when you type `herd status`, what is sent over the wire is the following sexp:

```
(shepherd-command
  (version 0)
  (action status) (service root)
  (arguments ()) (directory "/data/src/dmd"))
```

The reply is also an sexp, along these lines:

```
(reply (version 0)
       (result (((service ...) ...)))
       (error #f) (messages ()))
```

This reply indicates that the `status` action was successful, because `error` is `#f`, and gives a list of sexps denoting the status of services as its `result`. The `messages` field is a possibly-empty list of strings meant to be displayed as is to the user.

7 Internals

This chapter contains information about the design and the implementation details of the Shepherd for people who want to hack it.

The GNU Shepherd is developed by a group of people in connection with Guix System (`https://www.gnu.org/software/guix/`), GNU's advanced distribution, but it can be used on other distros as well. You're very much welcome to join us! You can report bugs to `bug-guix@gnu.org` and send patches or suggestions to `guix-devel@gnu.org`.

7.1 Coding Standards

About formatting: Use common sense and GNU Emacs (which actually is the same, of course), and you almost can't get the formatting wrong. Formatting should be as in Guile and Guix, basically. See Section "Coding Style" in *GNU Guix Reference Manual*, for more info.

7.2 Service Internals

Under the hood, each service record has an associated *fiber*, a lightweight execution thread and `shepherd` as a whole follows the CSP (Concurrent Sequential Processes) model (see Section "Introduction" in *Fibers*). A service's fiber encapsulates all the *state* of its corresponding service: its status (whether it's running, stopped, etc.), its "running value" (such as the PID of its associated process), the time at which its status changed, and so on. Procedures that access the state of a service, such as `service-status`, or that modify it, such as `start-service` (see Section 4.3 [Interacting with Services], page 13), merely send a message to the service's associated fiber.

This pattern follows the *actor model*: each of these per-service fibers is an *actor*[1]. There are several benefits:

- each actor has a linear control flow that is easy to reason about;
- access and modification of the service state are race-free since they are all handled sequentially by its actor;
- the actor's code is purely functional, which again makes it easier to reason about it.

There are other actors in the code, such as the service registry (see Section 4.2 [Service Registry], page 13). Fibers are used pervasively throughout the code to achieve concurrency.

Note that Fibers is set up such that the `shepherd` process has only one POSIX thread (this is mandated by POSIX for processes that call `fork`, with all its warts), and fibers are scheduled in a cooperative fashion. This means that it is possible to block the `shepherd` process for instance by running a long computation or by waiting on a socket that is not marked as `SOCK_NONBLOCK`. Be careful!

We think this programming model makes the code base not only more robust, but also very fun to work with—we hope you'll enjoy it too!

[1] There is one noteworthy difference compared to the actor model. In the "real" actor model, messaging among actors is asynchronous; in the CSP model, messaging is synchronous, which means that the sender and receiver must always *rendezvous* and their send/receive operation blocks until the other end is here to receive/send the message. See Section "Context" in *Fibers*, for more info.

7.3 Design Decisions

> **Note:** This section was written by Wolfgang Jährling back in 2003 and documents the original design of what was then known as GNU dmd. The main ideas remain valid but some implementation details and goals have changed.

The general idea of a service manager that uses dependencies, similar to those of a Makefile, came from the developers of the GNU Hurd, but as few people are satisfied with System V Init, many other people had the same idea independently. Nevertheless, the Shepherd was written with the goal of becoming a replacement for System V Init on GNU/Hurd, which was one of the reasons for choosing the extension language of the GNU project, Guile, for implementation (another reason being that it makes it just so much easier).

The runlevel concept (i.e. thinking in *groups* of services) is sometimes useful, but often one also wants to operate on single services. System V Init makes this hard: While you can start and stop a service, `init` will not know about it, and use the runlevel configuration as its source of information, opening the door for inconsistencies (which fortunately are not a practical problem usually). In the Shepherd, this was avoided by having a central entity that is responsible for starting and stopping the services, which therefore knows which services are actually started (if not completely improperly used, but that is a requirement which is impossible to avoid anyway). While runlevels are not implemented yet, it is clear that they will sit on top of the service concept, i.e. runlevels will merely be an optional extension that the service concept does not rely on. This also makes changes in the runlevel design easier when it may become necessary.

The consequence of having a daemon running that controls the services is that we need another program as user interface which communicates with the daemon. Fortunately, this makes the commands necessary for controlling services pretty short and intuitive, and gives the additional bonus of adding some more flexibility. For example, it is easiely possible to grant password-protected control over certain services to unprivileged users, if desired.

An essential aspect of the design of the Shepherd (which was already mentioned above) is that it should always know exactly what is happening, i.e. which services are started and stopped. The alternative would have been to not use a daemon, but to save the state on the file system, again opening the door for inconsistencies of all sorts. Also, we would have to use a separate program for respawning a service (which just starts the services, waits until it terminates and then starts it again). Killing the program that does the respawning (but not the service that is supposed to be respawned) would cause horrible confusion. My understanding of "The Right Thing" is that this conceptionally limited strategy is exactly what we do not want.

The way dependencies work in the Shepherd took a while to mature, as it was not easy to figure out what is appropriate. I decided to not make it too sophisticated by trying to guess what the user might want just to theoretically fulfill the request we are processing. If something goes wrong, it is usually better to tell the user about the problem and let her fix it, taking care to make finding solutions or workarounds for problems (like a misconfigured service) easy. This way, the user is in control of what happens and we can keep the implementation simple. To make a long story short, *we don't try to be too clever*, which is usually a good idea in developing software.

If you wonder why I was giving a "misconfigured service" as an example above, consider the following situation, which actually is a wonderful example for what was said in the previous paragraph: Service X depends on symbol S, which is provided by both A and B. A depends on AA, B depends on BB. AA and BB conflict with each other. The configuration of A contains an error, which will prevent it from starting; no service is running, but we want to start X now. In resolving its dependencies, we first try to start A, which will cause AA to be started. After this is done, the attempt of starting A fails, so we go on to B, but its dependency BB will fail to start because it conflicts with the running service AA. So we fail to provide S, thus X cannot be started. There are several possibilities to deal with this:

- When starting A fails, terminate those services which have been started in order to fulfill its dependencies (directly and indirectly). In case AA was running already, we would not want to terminate it. Well, maybe we would, to avoid the conflict with BB. But even if we would find out somehow that we need to terminate AA to eventually start X, is the user aware of this and wants this to happen (assuming AA was running already)? Probably not, she very likely has assumed that starting A succeeds and thus terminating AA is not necessary. Remember, unrelated (running) services might depend in AA. Even if we ignore this issue, this strategy is not only complicated, but also far from being perfect: Let's assume starting A succeeds, but X also depends on a service Z, which requires BB. In that case, we would need to detect in the first place that we should not even try to start A, but directly satisfy X's dependency on S with B.
- We could do it like stated above, but stop AA only if we know we won't need it anymore (for resolving further dependencies), and start it only when it does not conflict with anything that needs to get started. But should we stop it if it conflicts with something that *might* get started? (We do not always know for sure what we will start, as starting a service might fail and we want to fall back to a service that also provides the particular required symbol in that case.) I think that either decision will be bad in one case or another, even if this solution is already horribly complicated.
- When we are at it, we could just calculate a desired end-position, and try to get there by starting (and stopping!) services, recalculating what needs to be done whenever starting a service fails, also marking that particular service as unstartable, except if it fails to start because a dependency could not be resolved (or maybe even then?). This is even more complicated. Instead of implementing this and thereby producing code that (a) nobody understands, (b) certainly has a lot of bugs, (c) will be unmaintainable and (d) causes users to panic because they won't understand what will happen, I decided to do the following instead:
- Just report the error, and let the user fix it (in this case, fix the configuration of A) or work around it (in this case, disable A so that we won't start AA but directly go on to starting B).

I hope you can agree that the latter solution after all is the best one, because we can be sure to not do something that the user does not want us to do. Software should not run amok. This explanation was very long, but I think it was necessary to justify why the Shepherd uses a very primitive algorithm to resolve dependencies, despite the fact that it could theoretically be a bit more clever in certain situations.

One might argue that it is possible to ask the user if the planned actions are ok with her, and if the plan changes ask again, but especially given that services are supposed to usually work, I see few reasons to make the source code of the Shepherd more complicated than necessary. If you volunteer to write *and* maintain a more clever strategy (and volunteer to explain it to everyone who wants to understand it), you are welcome to do so, of course. . .

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

A

actions of services ... 12
actor model, for services ... 37
assertions ... 35
`at` command, with timer service ... 32

B

background, running commands ... 31

C

calendar events, for timers ... 20
canonical name of services ... 9
concurrent sequential processes, CSP ... 37
configuration file ... 4
configuration file, examples ... 24
constructor of a service ... 11
constructors, generation of ... 15
cron, converting date specifications ... 21
CSP, concurrent sequential processes ... 37

D

daemon ... 4
daemon controller ... 4
daylight saving time (DST), for timers ... 20
deco, daemon controller ... 4
destructor of a service ... 11
destructors, generation of ... 15
disabled service ... 17
dmd ... 1

E

endpoints, for inetd services ... 18
endpoints, for systemd services ... 18
evaluating code in shepherd ... 22
extensibility ... 3

G

generating constructors ... 15
generating destructors ... 15
GOOPS, former interface ... 23
graph of services ... 2, 12
Guile ... 1

H

herd ... 4, 6, 7, 8
herding, of daemons ... 4

I

inetd mode, example ... 25
inetd-style services ... 17
insecure ... 5
invoking `shepherd` ... 4

K

kexec, rebooting into a new Linux kernel ... 7

L

log file ... 5
log rotation ... 30
logging ... 5

N

nginx, service example ... 25

O

on-demand, starting services ... 17, 19
one-shot services ... 10
output ... 36

P

periodic services ... 20
protocol, between `shepherd` and its clients ... 36

R

read-eval-print loop, REPL ... 33
rebooting into a new Linux kernel with kexec ... 7
relative file names ... 4
REPL, read-eval-print loop ... 33
replacement, or a service ... 15
respawn delay ... 10, 17
respawn limit ... 10, 17
respawning services ... 10
root service ... 22
rotating log files of services ... 30

S

Scheme ... 1
security ... 5
service ... 9
service actions ... 12
service graph ... 2
service manager ... 1
service registry ... 13
shepherd ... 4
`shepherd` Invocation ... 4
socket activation, starting services ... 19
socket file, for `shepherd` ... 4
socket special file ... 5
spawning a program, from within shepherd ... 35
special services ... 22
starting a service ... 11
starting services, via socket activation ... 19
stopping a service ... 11
syslog ... 5
syslogd (system log) ... 28
system errors ... 35
system log (syslogd) ... 28
systemd-style services ... 19

T

termination of a service's process ... 11
timed services (timers) ... 20
timer periodicity ... 20
timer service, like `at` ... 32
timer, example ... 26
timers (timed services) ... 20
transient service maker ... 31
transient services ... 11

V

Vixie cron, converting date specifications ... 21

Procedure and Macro Index

A

action 12
actions 12
assert 35

C

calendar-event 20
command 22
cron-string->calendar-event 21

D

default-message-destination-procedure 30

E

endpoint 18
exec-command 16

F

for-each-service 13
fork+exec-command 16

L

local-output 36
log-rotation-service 31
lookup-running 13
lookup-service 13

M

make-forkexec-constructor 15
make-inetd-constructor 18
make-inetd-destructor 18
make-kill-destructor 16
make-system-constructor 19
make-system-destructor 20
make-systemd-constructor 19
make-systemd-destructor 19
make-timer-constructor 21
make-timer-destructor 21
monitoring-service 33

O

one-shot-service? 12
open-connection 36

P

perform-service-action 14

R

read-command 36
register-services 13
repl-service 34
respawn-service? 12

S

service 9
service-canonical-name 12
service-documentation 13
service-enabled? 14
service-log-file 15
service-process-exit-statuses 14
service-provision 12
service-recent-messages 15
service-replacement 15
service-requirement 12
service-respawn-delay 12
service-respawn-limit 13
service-respawn-times 14
service-running-value 14
service-running? 14
service-startup-failures 14
service-status 14
service-status-changes 14
service-stopped? 14
shepherd-command 36
spawn-command 35
start 24
start-in-the-background 14
start-service 14
stop 24
stop-service 14
system 35
system* 35
system-log-message-content 29
system-log-message-facility 29
system-log-message-priority 29
system-log-message-sender 29
system-log-message? 29
system-log-service 29

T

timer-service 33
transient-service 32
transient-service? 12

U

`unregister-services` 13
`user-environment-variables` 17

W

`without-system-error` 35
`write-command` 36

Variable Index

D

default-bind-attempts 18
default-environment-variables 17
default-max-silent-time 30
default-monitoring-period 33
default-pid-file-timeout 17
default-process-termination-grace-period 17
default-repl-socket-file 34
default-respawn-delay 17
default-respawn-limit 17

K

kernel-log-file 30

T

timer-trigger-action 22

X

XDG_RUNTIME_DIR 5

Type Index

`<service>` 9, 23
`<shepherd-command>` 35