NAME

systemd.unit − Unit configuration

SYNOPSIS

service.service, *socket*.socket, *device*.device, *mount*.mount, *automount*.automount, *swap*.swap, *target*.target, *path*.path, *timer*.timer, *slice*.slice, *scope*.scope

System Unit Search Path

```
/etc/systemd/system.control/*
/run/systemd/system.control/*
/run/systemd/transient/*
/run/systemd/generator.early/*
/etc/systemd/system/*
/etc/systemd/system.attached/*
/run/systemd/system/*
/run/systemd/system.attached/*
/run/systemd/generator/*
...
/lib/systemd/system/*
/run/systemd/generator.late/*
```

User Unit Search Path

```
~/.config/systemd/user.control/*
$XDG_RUNTIME_DIR/systemd/user.control/*
$XDG_RUNTIME_DIR/systemd/transient/*
$XDG_RUNTIME_DIR/systemd/generator.early/*
~/.config/systemd/user/*
$XDG_CONFIG_DIRS/systemd/user/*
/etc/systemd/user/*
$XDG_RUNTIME_DIR/systemd/user/*
/run/systemd/user/*
$XDG_RUNTIME_DIR/systemd/generator/*
$XDG_DATA_HOME/systemd/user/*
$XDG_DATA_DIRS/systemd/user/*
...
/usr/lib/systemd/user/*
$XDG_RUNTIME_DIR/systemd/generator.late/*
```

DESCRIPTION

A unit file is a plain text ini−style file that encodes information about a service, a socket, a device, a mount point, an automount point, a swap file or partition, a start−up target, a watched file system path, a timer controlled and supervised by **systemd**(1), a resource management slice or a group of externally created processes. See **systemd.syntax**(7) for a general description of the syntax.

This man page lists the common configuration options of all the unit types. These options need to be configured in the [Unit] or [Install] sections of the unit files.

In addition to the generic [Unit] and [Install] sections described here, each unit may have a type−specific section, e.g. [Service] for a service unit. See the respective man pages for more information: **systemd.service**(5), **systemd.socket**(5), **systemd.device**(5), **systemd.mount**(5), **systemd.automount**(5), **systemd.swap**(5), **systemd.target**(5), **systemd.path**(5), **systemd.timer**(5), **systemd.slice**(5), **systemd.scope**(5).

Unit files are loaded from a set of paths determined during compilation, described in the next section.

Valid unit names consist of a "name prefix" and a dot and a suffix specifying the unit type. The "unit prefix" must consist of one or more valid characters (ASCII letters, digits, ":", "−", "_", ".", and "\"). The total length of the unit name including the suffix must not exceed 256 characters. The type suffix must be one of ".service", ".socket", ".device", ".mount", ".automount", ".swap", ".target", ".path", ".timer", ".slice", or

".scope".

Units names can be parameterized by a single argument called the "instance name". The unit is then constructed based on a "template file" which serves as the definition of multiple services or other units. A template unit must have a single "@" at the end of the name (right before the type suffix). The name of the full unit is formed by inserting the instance name between "@" and the unit type suffix. In the unit file itself, the instance parameter may be referred to using "%i" and other specifiers, see below.

Unit files may contain additional options on top of those listed here. If systemd encounters an unknown option, it will write a warning log message but continue loading the unit. If an option or section name is prefixed with **X−**, it is ignored completely by systemd. Options within an ignored section do not need the prefix. Applications may use this to include additional information in the unit files. To access those options, applications need to parse the unit files on their own.

Units can be aliased (have an alternative name), by creating a symlink from the new name to the existing name in one of the unit search paths. For example, systemd−networkd.service has the alias dbus−org.freedesktop.network1.service, created during installation as a symlink, so when **systemd** is asked through D−Bus to load dbus−org.freedesktop.network1.service, it'll load systemd−networkd.service. As another example, default.target — the default system target started at boot — is commonly symlinked (aliased) to either multi−user.target or graphical.target to select what is started by default. Alias names may be used in commands like **disable**, **start**, **stop**, **status**, and similar, and in all unit dependency directives, including *Wants=*, *Requires=*, *Before=*, *After=*. Aliases cannot be used with the **preset** command.

Aliases obey the following restrictions: a unit of a certain type (".service", ".socket", ...) can only be aliased by a name with the same type suffix. A plain unit (not a template or an instance), may only be aliased by a plain name. A template instance may only be aliased by another template instance, and the instance part must be identical. A template may be aliased by another template (in which case the alias applies to all instances of the template). As a special case, a template instance (e.g. "alias@inst.service") may be a symlink to different template (e.g. "template@inst.service"). In that case, just this specific instance is aliased, while other instances of the template (e.g. "alias@foo.service", "alias@bar.service") are not aliased. Those rule preserve the requirement that the instance (if any) is always uniquely defined for a given unit and all its aliases.

Unit files may specify aliases through the *Alias=* directive in the [Install] section. When the unit is enabled, symlinks will be created for those names, and removed when the unit is disabled. For example, reboot.target specifies *Alias=ctrl−alt−del.target*, so when enabled, the symlink /etc/systemd/system/ctrl−alt−del.service pointing to the reboot.target file will be created, and when Ctrl+Alt+Del is invoked, **systemd** will look for the ctrl−alt−del.service and execute reboot.service. **systemd** does not look at the [Install] section at all during normal operation, so any directives in that section only have an effect through the symlinks created during enablement.

Along with a unit file foo.service, the directory foo.service.wants/ may exist. All unit files symlinked from such a directory are implicitly added as dependencies of type *Wants=* to the unit. Similar functionality exists for *Requires=* type dependencies as well, the directory suffix is .requires/ in this case. This functionality is useful to hook units into the start−up of other units, without having to modify their unit files. For details about the semantics of *Wants=*, see below. The preferred way to create symlinks in the .wants/ or .requires/ directory of a unit file is by embedding the dependency in [Install] section of the target unit, and creating the symlink in the file system with the **enable** or **preset** commands of **systemctl**(1).

Along with a unit file foo.service, a "drop−in" directory foo.service.d/ may exist. All files with the suffix ".conf" from this directory will be merged in the alphanumeric order and parsed after the main unit file itself has been parsed. This is useful to alter or add configuration settings for a unit, without having to modify unit files. Each drop−in file must contain appropriate section headers. For instantiated units, this logic will first look for the instance ".d/" subdirectory (e.g. "foo@bar.service.d/") and read its ".conf" files, followed by the template ".d/" subdirectory (e.g. "foo@.service.d/") and the ".conf" files there. Moreover for unit names containing dashes ("−"), the set of directories generated by repeatedly truncating the unit name after all dashes is searched too. Specifically, for a unit name foo−bar−baz.service not only the regular drop−in directory foo−bar−baz.service.d/ is searched but also both foo−bar−.service.d/ and foo−.service.d/. This is useful for defining common drop−ins for a set of related units, whose names begin with a common prefix. This scheme is particularly useful for mount, automount and slice units, whose systematic naming structure is built around dashes as component separators. Note that equally named drop−in files further down the prefix hierarchy override those further up, i.e. foo−bar−.service.d/10−override.conf overrides foo−.service.d/10−override.conf.

In cases of unit aliases (described above), dropins for the aliased name and all aliases are loaded. In the example of default.target aliasing graphical.target, default.target.d/, default.target.wants/, default.target.requires/, graphical.target.d/, graphical.target.wants/, graphical.target.requires/ would all be read. For templates, dropins for the template, any template aliases, the template instance, and all alias instances are read. When just a specific template instance is aliased, then the dropins for the target template, the target template instance, and the alias template instance are read.

In addition to /etc/systemd/system, the drop−in ".d/" directories for system services can be placed in /lib/systemd/system or /run/systemd/system directories. Drop−in files in /etc/ take precedence over those in /run/ which in turn take precedence over those in /lib/. Drop−in files under any of these directories take precedence over unit files wherever located. Multiple drop−in files with different names are applied in lexicographic order, regardless of which of the directories they reside in.

Units also support a top−level drop−in with *type*.d/, where *type* may be e.g. "service" or "socket", that allows altering or adding to the settings of all corresponding unit files on the system. The formatting and precedence of applying drop−in configurations follow what is defined above. Files in *type*.d/ have lower precedence compared to files in name−specific override directories. The usual rules apply: multiple drop−in files with different names are applied in lexicographic order, regardless of which of the directories they reside in, so a file in *type*.d/ applies to a unit only if there are no drop−ins or masks with that name in directories with higher precedence. See Examples.

Note that while systemd offers a flexible dependency system between units it is recommended to use this functionality only sparingly and instead rely on techniques such as bus−based or socket−based activation which make dependencies implicit, resulting in a both simpler and more flexible system.

As mentioned above, a unit may be instantiated from a template file. This allows creation of multiple units from a single configuration file. If systemd looks for a unit configuration file, it will first search for the literal unit name in the file system. If that yields no success and the unit name contains an "@" character, systemd will look for a unit template that shares the same name but with the instance string (i.e. the part between the "@" character and the suffix) removed. Example: if a service getty@tty3.service is requested and no file by that name is found, systemd will look for getty@.service and instantiate a service from that configuration file if it is found.

To refer to the instance string from within the configuration file you may use the special "%i" specifier in many of the configuration options. See below for details.

If a unit file is empty (i.e. has the file size 0) or is symlinked to /dev/null, its configuration will not be loaded and it appears with a load state of "masked", and cannot be activated. Use this as an effective way to fully disable a unit, making it impossible to start it even manually.

The unit file format is covered by the **Interface Portability and Stability Promise**[1].

STRING ESCAPING FOR INCLUSION IN UNIT NAMES

Sometimes it is useful to convert arbitrary strings into unit names. To facilitate this, a method of string escaping is used, in order to map strings containing arbitrary byte values (except **NUL**) into valid unit names and their restricted character set. A common special case are unit names that reflect paths to objects in the file system hierarchy. Example: a device unit dev−sda.device refers to a device with the device node /dev/sda in the file system.

The escaping algorithm operates as follows: given a string, any "/" character is replaced by "−", and all other characters which are not ASCII alphanumerics, ":", "_" or "." are replaced by C−style "\x2d" escapes. In addition, "." is replaced with such a C−style escape when it would appear as the first character in the escaped string.

When the input qualifies as absolute file system path, this algorithm is extended slightly: the path to the root directory "/" is encoded as single dash "−". In addition, any leading, trailing or duplicate "/" characters are removed from the string before transformation. Example: /foo//bar/baz/ becomes "foo−bar−baz".

This escaping is fully reversible, as long as it is known whether the escaped string was a path (the unescaping results are different for paths and non−path strings). The **systemd-escape**(1) command may be used to apply and reverse escaping on arbitrary strings. Use **systemd−escape −−path** to escape path strings, and **systemd−escape** without **−−path** otherwise.

AUTOMATIC DEPENDENCIES

Implicit Dependencies

A number of unit dependencies are implicitly established, depending on unit type and unit configuration. These implicit dependencies can make unit configuration file cleaner. For the implicit dependencies in each unit type, please refer to section "Implicit Dependencies" in respective man pages.

For example, service units with *Type=dbus* automatically acquire dependencies of type *Requires=* and *After=* on dbus.socket. See **systemd.service**(5) for details.

Default Dependencies

Default dependencies are similar to implicit dependencies, but can be turned on and off by setting *DefaultDependencies=* to *yes* (the default) and *no*, while implicit dependencies are always in effect. See section "Default Dependencies" in respective man pages for the effect of enabling *DefaultDependencies=* in each unit types.

For example, target units will complement all configured dependencies of type *Wants=* or *Requires=* with dependencies of type *After=* unless *DefaultDependencies=no* is set in the specified units. See **systemd.target**(5) for details. Note that this behavior can be turned off by setting *DefaultDependencies=no*.

UNIT FILE LOAD PATH

Unit files are loaded from a set of paths determined during compilation, described in the two tables below. Unit files found in directories listed earlier override files with the same name in directories lower in the list.

When the variable *$SYSTEMD_UNIT_PATH* is set, the contents of this variable overrides the unit load path. If *$SYSTEMD_UNIT_PATH* ends with an empty component (":"), the usual unit load path will be appended to the contents of the variable.

Table 1. Load path when running in system mode (−−system).

Path	Description
/etc/systemd/system.control	Persistent and transient configuration created using the dbus API
/run/systemd/system.control	
/run/systemd/transient	Dynamic configuration for transient units
/run/systemd/generator.early	Generated units with high priority (see *early−dir* in **systemd.generator**(7))
/etc/systemd/system	System units created by the administrator
/run/systemd/system	Runtime units
/run/systemd/generator	Generated units with medium priority (see *normal−dir* in **systemd.generator**(7))
/usr/local/lib/systemd/system	System units installed by the administrator
/lib/systemd/system	System units installed by the distribution package manager
/run/systemd/generator.late	Generated units with low priority (see *late−dir* in **systemd.generator**(7))

Table 2. Load path when running in user mode (−−user).

Path	Description
$XDG_CONFIG_HOME/systemd/user.control or ˜/.config/systemd/user.control	Persistent and transient configuration created using the dbus API (*$XDG_CONFIG_HOME* is used if set, ˜/.config otherwise)
$XDG_RUNTIME_DIR/systemd/user.control	
/run/systemd/transient	Dynamic configuration for transient units
/run/systemd/generator.early	Generated units with high priority (see *early−dir* in **systemd.generator**(7))
$XDG_CONFIG_HOME/systemd/user or $HOME/.config/systemd/user	User configuration (*$XDG_CONFIG_HOME* is used if set, ˜/.config otherwise)
$XDG_CONFIG_DIRS/systemd/user or /etc/xdg/systemd/user	Additional configuration directories as specified by the XDG base directory specification (*$XDG_CONFIG_DIRS* is used if set, /etc/xdg otherwise)
/etc/systemd/user	User units created by the administrator
$XDG_RUNTIME_DIR/systemd/user	Runtime units (only used when $XDG_RUNTIME_DIR is set)
/run/systemd/user	Runtime units
$XDG_RUNTIME_DIR/systemd/generator	Generated units with medium priority (see *normal−dir* in **systemd.generator**(7))
$XDG_DATA_HOME/systemd/user or $HOME/.local/share/systemd/user	Units of packages that have been installed in the home directory (*$XDG_DATA_HOME* is used if set, ˜/.local/share otherwise)
$XDG_DATA_DIRS/systemd/user or /usr/local/share/systemd/user and /usr/share/systemd/user	Additional data directories as specified by the XDG base directory specification (*$XDG_DATA_DIRS* is used if set, /usr/local/share and /usr/share otherwise)
$dir/systemd/user for each *$dir* in *$XDG_DATA_DIRS*	Additional locations for installed user units, one for each entry in *$XDG_DATA_DIRS*
/usr/local/lib/systemd/user	User units installed by the administrator
/usr/lib/systemd/user	User units installed by the distribution package manager
$XDG_RUNTIME_DIR/systemd/generator.late	Generated units with low priority (see *late−dir* in **systemd.generator**(7))

The set of load paths for the user manager instance may be augmented or changed using various environment variables. And environment variables may in turn be set using environment generators, see **systemd.environment-generator**(7). In particular, *$XDG_DATA_HOME* and *$XDG_DATA_DIRS* may be easily set using **systemd-environment-d-generator**(8). Thus, directories listed here are just the defaults. To see the actual list that would be used based on compilation options and current environment use

```
systemd−analyze −−user unit−paths
```

Moreover, additional units might be loaded into systemd from directories not on the unit load path by creating a symlink pointing to a unit file in the directories. You can use **systemctl link** for this operation. See **systemctl**(1) for its usage and precaution.

UNIT GARBAGE COLLECTION

The system and service manager loads a unit's configuration automatically when a unit is referenced for the first time. It will automatically unload the unit configuration and state again when the unit is not needed anymore ("garbage collection"). A unit may be referenced through a number of different mechanisms:

1. Another loaded unit references it with a dependency such as *After=*, *Wants=*, ...
2. The unit is currently starting, running, reloading or stopping.
3. The unit is currently in the **failed** state. (But see below.)
4. A job for the unit is pending.
5. The unit is pinned by an active IPC client program.
6. The unit is a special "perpetual" unit that is always active and loaded. Examples for perpetual units are the root mount unit −.mount or the scope unit init.scope that the service manager itself lives in.
7. The unit has running processes associated with it.

The garbage collection logic may be altered with the *CollectMode=* option, which allows configuration whether automatic unloading of units that are in **failed** state is permissible, see below.

Note that when a unit's configuration and state is unloaded, all execution results, such as exit codes, exit signals, resource consumption and other statistics are lost, except for what is stored in the log subsystem.

Use **systemctl daemon−reload** or an equivalent command to reload unit configuration while the unit is already loaded. In this case all configuration settings are flushed out and replaced with the new configuration (which however might not be in effect immediately), however all runtime state is saved/restored.

[UNIT] SECTION OPTIONS

The unit file may include a [Unit] section, which carries generic information about the unit that is not dependent on the type of unit:

Description=

A short human readable title of the unit. This may be used by **systemd** (and other UIs) as a user−visible label for the unit, so this string should identify the unit rather than describe it, despite the name. This string also shouldn't just repeat the unit name. "Apache2 Web Server" is a good example. Bad examples are "high−performance light−weight HTTP server" (too generic) or "Apache2" (meaningless for people who do not know Apache, duplicates the unit name). **systemd** may use this string as a noun in status messages ("Starting *description*...", "Started *description*.", "Reached target *description*.", "Failed to start *description*."), so it should be capitalized, and should not be a full sentence, or a phrase with a continuous verb. Bad examples include "exiting the container" or "updating the database once per day.".

Documentation=

A space−separated list of URIs referencing documentation for this unit or its configuration. Accepted are only URIs of the types "http://", "https://", "file:", "info:", "man:". For more information about the syntax of these URIs, see **uri**(7). The URIs should be listed in order of relevance, starting with the most relevant. It is a good idea to first reference documentation that explains what the unit's purpose is, followed by how it is configured, followed by any other related documentation. This option may be specified more than once, in which case the specified list of URIs is merged. If the empty string is assigned to this option, the list is reset and all prior assignments will have no effect.

Wants=

Configures (weak) requirement dependencies on other units. This option may be specified more than once or multiple space−separated units may be specified in one option in which case dependencies for all listed names will be created. Dependencies of this type may also be configured outside of the unit configuration file by adding a symlink to a .wants/ directory accompanying the unit file. For details, see above.

Units listed in this option will be started if the configuring unit is. However, if the listed units fail to start or cannot be added to the transaction, this has no impact on the validity of the transaction as a whole, and this unit will still be started. This is the recommended way to hook the start−up of one unit to the start−up of another unit.

Note that requirement dependencies do not influence the order in which services are started or stopped. This has to be configured independently with the *After=* or *Before=* options. If unit foo.service pulls in unit bar.service as configured with *Wants=* and no ordering is configured with *After=* or *Before=*, then both units will be started simultaneously and without any delay between them if foo.service is activated.

Requires=

Similar to *Wants=*, but declares a stronger requirement dependency. Dependencies of this type may also be configured by adding a symlink to a .requires/ directory accompanying the unit file.

If this unit gets activated, the units listed will be activated as well. If one of the other units fails to activate, and an ordering dependency *After=* on the failing unit is set, this unit will not be started. Besides, with or without specifying *After=*, this unit will be stopped if one of the other units is explicitly stopped.

Often, it is a better choice to use *Wants=* instead of *Requires=* in order to achieve a system that is more robust when dealing with failing services.

Note that this dependency type does not imply that the other unit always has to be in active state when this unit is running. Specifically: failing condition checks (such as *ConditionPathExists=*, *ConditionPathIsSymbolicLink=*, ... — see below) do not cause the start job of a unit with a *Requires=* dependency on it to fail. Also, some unit types may deactivate on their own (for example, a service process may decide to exit cleanly, or a device may be unplugged by the user), which is not propagated to units having a *Requires=* dependency. Use the *BindsTo=* dependency type together with *After=* to ensure that a unit may never be in active state without a specific other unit also in active state (see below).

Requisite=

Similar to *Requires=*. However, if the units listed here are not started already, they will not be started and the starting of this unit will fail immediately. *Requisite=* does not imply an ordering dependency, even if both units are started in the same transaction. Hence this setting should usually be combined with *After=*, to ensure this unit is not started before the other unit.

When *Requisite=b.service* is used on a.service, this dependency will show as *RequisiteOf=a.service* in property listing of b.service. *RequisiteOf=* dependency cannot be specified directly.

BindsTo=

Configures requirement dependencies, very similar in style to *Requires=*. However, this dependency type is stronger: in addition to the effect of *Requires=* it declares that if the unit bound to is stopped, this unit will be stopped too. This means a unit bound to another unit that suddenly enters inactive state will be stopped too. Units can suddenly, unexpectedly enter inactive state for different reasons: the main process of a service unit might terminate on its own choice, the backing device of a device unit might be unplugged or the mount point of a mount unit might be unmounted without involvement of the system and service manager.

When used in conjunction with *After=* on the same unit the behaviour of *BindsTo=* is even stronger. In this case, the unit bound to strictly has to be in active state for this unit to also be in active state. This not only means a unit bound to another unit that suddenly enters inactive state, but also one that is bound to another unit that gets skipped due to a failed condition check (such as *ConditionPathExists=*, *ConditionPathIsSymbolicLink=*, ... — see below) will be stopped, should it be running. Hence, in many cases it is best to combine *BindsTo=* with *After=*.

When *BindsTo=b.service* is used on a.service, this dependency will show as *BoundBy=a.service* in property listing of b.service. *BoundBy=* dependency cannot be specified directly.

PartOf=

Configures dependencies similar to *Requires=*, but limited to stopping and restarting of units. When systemd stops or restarts the units listed here, the action is propagated to this unit. Note that this is a one−way dependency — changes to this unit do not affect the listed units.

When *PartOf=b.service* is used on a.service, this dependency will show as *ConsistsOf=a.service* in property listing of b.service. *ConsistsOf=* dependency cannot be specified directly.

Upholds=

Configures dependencies similar to *Wants=*, but as long a this unit is up, all units listed in *Upholds=* are started whenever found to be inactive or failed, and no job is queued for them. While a *Wants=* dependency on another unit has a one−time effect when this units started, a *Upholds=* dependency on it has a continuous effect, constantly restarting the unit if necessary. This is an alternative to the *Restart=* setting of service units, to ensure they are kept running whatever happens.

When *Upholds=b.service* is used on a.service, this dependency will show as *UpheldBy=a.service* in the property listing of b.service. The *UpheldBy=* dependency cannot be specified directly.

Conflicts=

A space−separated list of unit names. Configures negative requirement dependencies. If a unit has a *Conflicts=* setting on another unit, starting the former will stop the latter and vice versa.

Note that this setting does not imply an ordering dependency, similarly to the *Wants=* and *Requires=* dependencies described above. This means that to ensure that the conflicting unit is stopped before the other unit is started, an *After=* or *Before=* dependency must be declared. It doesn't matter which of the two ordering dependencies is used, because stop jobs are always ordered before start jobs, see the discussion in *Before=*/*After=* below.

If unit A that conflicts with unit B is scheduled to be started at the same time as B, the transaction will either fail (in case both are required parts of the transaction) or be modified to be fixed (in case one or both jobs are not a required part of the transaction). In the latter case, the job that is not required will be removed, or in case both are not required, the unit that conflicts will be started and the unit that is conflicted is stopped.

Before=, *After=*

These two settings expect a space−separated list of unit names. They may be specified more than once, in which case dependencies for all listed names are created.

Those two settings configure ordering dependencies between units. If unit foo.service contains the setting **Before=bar.service** and both units are being started, bar.service's start−up is delayed until foo.service has finished starting up. *After=* is the inverse of *Before=*, i.e. while *Before=* ensures that the configured unit is started before the listed unit begins starting up, *After=* ensures the opposite, that the listed unit is fully started up before the configured unit is started.

When two units with an ordering dependency between them are shut down, the inverse of the start−up order is applied. I.e. if a unit is configured with *After=* on another unit, the former is stopped before the latter if both are shut down. Given two units with any ordering dependency between them, if one unit is shut down and the other is started up, the shutdown is ordered before the start−up. It doesn't matter if the ordering dependency is *After=* or *Before=*, in this case. It also doesn't matter which of the two is shut down, as long as one is shut down and the other is started up; the shutdown is ordered before the start−up in all cases. If two units have no ordering dependencies between them, they are shut down or started up simultaneously, and no ordering takes place. It depends on the unit type when precisely a unit has finished starting up. Most importantly, for service units start−up is considered completed for the purpose of *Before=*/*After=* when all its configured start−up commands have been invoked and they either failed or reported start−up success. Note that this does includes *ExecStartPost=* (or *ExecStopPost=* for the shutdown case).

Note that those settings are independent of and orthogonal to the requirement dependencies as configured by *Requires=*, *Wants=*, *Requisite=*, or *BindsTo=*. It is a common pattern to include a unit name in both the *After=* and *Wants=* options, in which case the unit listed will be started before the unit that is configured with these options.

Note that *Before=* dependencies on device units have no effect and are not supported. Devices generally become available as a result of an external hotplug event, and systemd creates the corresponding device unit without delay.

OnFailure=

A space−separated list of one or more units that are activated when this unit enters the "failed" state. A service unit using *Restart=* enters the failed state only after the start limits are reached.

OnSuccess=

A space−separated list of one or more units that are activated when this unit enters the "inactive" state.

PropagatesReloadTo=, *ReloadPropagatedFrom=*

A space−separated list of one or more units to which reload requests from this unit shall be propagated to, or units from which reload requests shall be propagated to this unit, respectively. Issuing a reload request on a unit will automatically also enqueue reload requests on all units that are linked to it using these two settings.

PropagatesStopTo=, *StopPropagatedFrom=*

A space−separated list of one or more units to which stop requests from this unit shall be propagated to, or units from which stop requests shall be propagated to this unit, respectively. Issuing a stop request on a unit will automatically also enqueue stop requests on all units that are linked to it using these two settings.

JoinsNamespaceOf=

For units that start processes (such as service units), lists one or more other units whose network and/or temporary file namespace to join. This only applies to unit types which support the *PrivateNetwork=*, *NetworkNamespacePath=*, *PrivateIPC=*, *IPCNamespacePath=*, and *PrivateTmp=* directives (see **systemd.exec**(5) for details). If a unit that has this setting set is started, its processes will see the same /tmp/, /var/tmp/, IPC namespace and network namespace as one listed unit that is started. If multiple listed units are already started, it is not defined which namespace is joined. Note that this setting only has an effect if *PrivateNetwork=*/*NetworkNamespacePath=*, *PrivateIPC=*/*IPCNamespacePath=* and/or *PrivateTmp=* is enabled for both the unit that joins the namespace and the unit whose namespace is joined.

RequiresMountsFor=

Takes a space−separated list of absolute paths. Automatically adds dependencies of type *Requires=* and *After=* for all mount units required to access the specified path.

Mount points marked with **noauto** are not mounted automatically through local−fs.target, but are still honored for the purposes of this option, i.e. they will be pulled in by this unit.

OnFailureJobMode=

Takes a value of "fail", "replace", "replace−irreversibly", "isolate", "flush", "ignore−dependencies" or "ignore−requirements". Defaults to "replace". Specifies how the units listed in *OnFailure=* will be enqueued. See **systemctl**(1)'s **−−job−mode=** option for details on the possible values. If this is set to "isolate", only a single unit may be listed in *OnFailure=*.

IgnoreOnIsolate=
: Takes a boolean argument. If **true**, this unit will not be stopped when isolating another unit. Defaults to **false** for service, target, socket, timer, and path units, and **true** for slice, scope, device, swap, mount, and automount units.

StopWhenUnneeded=
: Takes a boolean argument. If **true**, this unit will be stopped when it is no longer used. Note that, in order to minimize the work to be executed, systemd will not stop units by default unless they are conflicting with other units, or the user explicitly requested their shut down. If this option is set, a unit will be automatically cleaned up if no other active unit requires it. Defaults to **false**.

RefuseManualStart=, *RefuseManualStop=*
: Takes a boolean argument. If **true**, this unit can only be activated or deactivated indirectly. In this case, explicit start−up or termination requested by the user is denied, however if it is started or stopped as a dependency of another unit, start−up or termination will succeed. This is mostly a safety feature to ensure that the user does not accidentally activate units that are not intended to be activated explicitly, and not accidentally deactivate units that are not intended to be deactivated. These options default to **false**.

AllowIsolate=
: Takes a boolean argument. If **true**, this unit may be used with the **systemctl isolate** command. Otherwise, this will be refused. It probably is a good idea to leave this disabled except for target units that shall be used similar to runlevels in SysV init systems, just as a precaution to avoid unusable system states. This option defaults to **false**.

DefaultDependencies=
: Takes a boolean argument. If **yes**, (the default), a few default dependencies will implicitly be created for the unit. The actual dependencies created depend on the unit type. For example, for service units, these dependencies ensure that the service is started only after basic system initialization is completed and is properly terminated on system shutdown. See the respective man pages for details. Generally, only services involved with early boot or late shutdown should set this option to **no**. It is highly recommended to leave this option enabled for the majority of common units. If set to **no**, this option does not disable all implicit dependencies, just non−essential ones.

CollectMode=
: Tweaks the "garbage collection" algorithm for this unit. Takes one of **inactive** or **inactive−or−failed**. If set to **inactive** the unit will be unloaded if it is in the **inactive** state and is not referenced by clients, jobs or other units — however it is not unloaded if it is in the **failed** state. In **failed** mode, failed units are not unloaded until the user invoked **systemctl reset−failed** on them to reset the **failed** state, or an equivalent command. This behaviour is altered if this option is set to **inactive−or−failed**: in this case the unit is unloaded even if the unit is in a **failed** state, and thus an explicitly resetting of the **failed** state is not necessary. Note that if this mode is used unit results (such as exit codes, exit signals, consumed resources, ...) are flushed out immediately after the unit completed, except for what is stored in the logging subsystem. Defaults to **inactive**.

FailureAction=, *SuccessAction=*
: Configure the action to take when the unit stops and enters a failed state or inactive state. Takes one of **none**, **reboot**, **reboot−force**, **reboot−immediate**, **poweroff**, **poweroff−force**, **poweroff−immediate**, **exit**, and **exit−force**. In system mode, all options are allowed. In user mode, only **none**, **exit**, and **exit−force** are allowed. Both options default to **none**.

  If **none** is set, no action will be triggered. **reboot** causes a reboot following the normal shutdown procedure (i.e. equivalent to **systemctl reboot**). **reboot−force** causes a forced reboot which will terminate all processes forcibly but should cause no dirty file systems on reboot (i.e. equivalent to **systemctl reboot −f**) and **reboot−immediate** causes immediate execution of the **reboot**(2) system call, which might result in data loss (i.e. equivalent to **systemctl reboot −ff**). Similarly, **poweroff**, **poweroff−force**, **poweroff−immediate** have the effect of powering down the system with similar semantics. **exit** causes the manager to exit following the normal shutdown procedure, and **exit−force** causes it terminate without shutting down services. When **exit** or **exit−force** is used by default the exit status of the main process of the unit (if this applies) is returned from the service manager. However, this may be overridden with *FailureActionExitStatus=*/*SuccessActionExitStatus=*, see below.

FailureActionExitStatus=, *SuccessActionExitStatus=*
: Controls the exit status to propagate back to an invoking container manager (in case of a system service) or service manager (in case of a user manager) when the *FailureAction=*/*SuccessAction=* are set to **exit** or **exit−force** and the action is triggered. By default the exit status of the main process of the triggering unit (if this applies) is propagated. Takes a value in the range 0...255 or the empty string to request default behaviour.

JobTimeoutSec=, *JobRunningTimeoutSec=*
: *JobTimeoutSec=* specifies a timeout for the whole job that starts running when the job is queued. *JobRunningTimeoutSec=* specifies a timeout that starts running when the queued job is actually started. If either limit is reached, the job will be cancelled, the unit however will not change state or even enter the "failed" mode.

  Both settings take a time span with the default unit of seconds, but other units may be specified, see **systemd.time**(5). The default is "infinity" (job timeouts disabled), except for device units where *JobRunningTimeoutSec=* defaults to *DefaultTimeoutStartSec=*.

  Note: these timeouts are independent from any unit−specific timeouts (for example, the timeout set with *TimeoutStartSec=* in service units). The job timeout has no effect on the unit itself. Or in other words: unit−specific timeouts are useful to abort unit state changes, and revert them. The job timeout set with this option however is useful to abort only the job waiting for the unit state to change.

JobTimeoutAction=, *JobTimeoutRebootArgument=*
: *JobTimeoutAction=* optionally configures an additional action to take when the timeout is hit, see description of *JobTimeoutSec=* and *JobRunningTimeoutSec=* above. It takes the same values as *StartLimitAction=*. Defaults to **none**.

  JobTimeoutRebootArgument= configures an optional reboot string to pass to the **reboot**(2) system call.

StartLimitIntervalSec=interval, *StartLimitBurst=burst*
: Configure unit start rate limiting. Units which are started more than *burst* times within an *interval* time span are not permitted to start any more. Use *StartLimitIntervalSec=* to configure the checking interval and *StartLimitBurst=* to configure how many starts per interval are allowed.

  interval is a time span with the default unit of seconds, but other units may be specified, see **systemd.time**(5). Defaults to *DefaultStartLimitIntervalSec=* in manager configuration file, and may be set to 0 to disable any kind of rate limiting. *burst* is a number and defaults to *DefaultStartLimitBurst=* in manager configuration file.

  These configuration options are particularly useful in conjunction with the service setting *Restart=* (see **systemd.service**(5)); however, they apply to all kinds of starts (including manual), not just those triggered by the *Restart=* logic.

  Note that units which are configured for *Restart=*, and which reach the start limit are not attempted to be restarted anymore; however, they may still be restarted manually or from a timer or socket at a later point, after the *interval* has passed. From that point on, the restart logic is activated again. **systemctl reset−failed** will cause the restart rate counter for a service to be flushed, which is useful if the administrator wants to manually start a unit and the start limit interferes with that. Rate−limiting is enforced after any unit condition checks are executed, and hence unit activations with failing conditions do not count towards the rate limit.

  When a unit is unloaded due to the garbage collection logic (see above) its rate limit counters are flushed out too. This means that configuring start rate limiting for a unit that is not referenced continuously has no effect.

This setting does not apply to slice, target, device, and scope units, since they are unit types whose activation may either never fail, or may succeed only a single time.

StartLimitAction=
: Configure an additional action to take if the rate limit configured with *StartLimitIntervalSec=* and *StartLimitBurst=* is hit. Takes the same values as the *FailureAction=*/*SuccessAction=* settings. If **none** is set, hitting the rate limit will trigger no action except that the start will not be permitted. Defaults to **none**.

RebootArgument=
: Configure the optional argument for the **reboot**(2) system call if *StartLimitAction=* or *FailureAction=* is a reboot action. This works just like the optional argument to **systemctl reboot** command.

SourcePath=
: A path to a configuration file this unit has been generated from. This is primarily useful for implementation of generator tools that convert configuration from an external configuration file format into native unit files. This functionality should not be used in normal units.

Conditions and Asserts

Unit files may also include a number of *Condition...=* and *Assert...=* settings. Before the unit is started, systemd will verify that the specified conditions and asserts are true. If not, the starting of the unit will be (mostly silently) skipped (in case of conditions), or aborted with an error message (in case of asserts). Failing conditions or asserts will not result in the unit being moved into the "failed" state. The conditions and asserts are checked at the time the queued start job is to be executed. The ordering dependencies are still respected, so other units are still pulled in and ordered as if this unit was successfully activated, and the conditions and asserts are executed the precise moment the unit would normally start and thus can validate system state after the units ordered before completed initialization. Use condition expressions for skipping units that do not apply to the local system, for example because the kernel or runtime environment doesn't require their functionality.

If multiple conditions are specified, the unit will be executed if all of them apply (i.e. a logical AND is applied). Condition checks can use a pipe symbol ("|") after the equals sign ("Condition...=|..."), which causes the condition to become a *triggering* condition. If at least one triggering condition is defined for a unit, then the unit will be started if at least one of the triggering conditions of the unit applies and all of the regular (i.e. non−triggering) conditions apply. If you prefix an argument with the pipe symbol and an exclamation mark, the pipe symbol must be passed first, the exclamation second. If any of these options is assigned the empty string, the list of conditions is reset completely, all previous condition settings (of any kind) will have no effect.

The *AssertArchitecture=*, *AssertVirtualization=*, ... options are similar to conditions but cause the start job to fail (instead of being skipped). The failed check is logged. Units with failed conditions are considered to be in a clean state and will be garbage collected if they are not referenced. This means that when queried, the condition failure may or may not show up in the state of the unit.

Note that neither assertion nor condition expressions result in unit state changes. Also note that both are checked at the time the job is to be executed, i.e. long after depending jobs and it itself were queued. Thus, neither condition nor assertion expressions are suitable for conditionalizing unit dependencies.

The **condition** verb of **systemd-analyze**(1) can be used to test condition and assert expressions.

Except for *ConditionPathIsSymbolicLink=*, all path checks follow symlinks.

ConditionArchitecture=
: Check whether the system is running on a specific architecture. Takes one of "x86", "x86−64", "ppc", "ppc−le", "ppc64", "ppc64−le", "ia64", "parisc", "parisc64", "s390", "s390x", "sparc", "sparc64", "mips", "mips−le", "mips64", "mips64−le", "alpha", "arm", "arm−be", "arm64", "arm64−be", "sh", "sh64", "m68k", "tilegx", "cris", "arc", "arc−be", or "native".

The architecture is determined from the information returned by **uname**(2) and is thus subject to **personality**(2). Note that a *Personality=* setting in the same unit file has no effect on this condition. A special architecture name "native" is mapped to the architecture the system manager itself is compiled for. The test may be negated by prepending an exclamation mark.

ConditionFirmware=
Check whether the system's firmware is of a certain type. Possible values are: "uefi" (for systems with EFI), "device−tree" (for systems with a device tree) and "device−tree−compatible(xyz)" (for systems with a device tree that is compatible to "xyz").

ConditionVirtualization=
Check whether the system is executed in a virtualized environment and optionally test whether it is a specific implementation. Takes either boolean value to check if being executed in any virtualized environment, or one of "vm" and "container" to test against a generic type of virtualization solution, or one of "qemu", "kvm", "amazon", "zvm", "vmware", "microsoft", "oracle", "powervm", "xen", "bochs", "uml", "bhyve", "qnx", "openvz", "lxc", "lxc−libvirt", "systemd−nspawn", "docker", "podman", "rkt", "wsl", "proot", "pouch", "acrn" to test against a specific implementation, or "private−users" to check whether we are running in a user namespace. See **systemd-detect-virt**(1) for a full list of known virtualization technologies and their identifiers. If multiple virtualization technologies are nested, only the innermost is considered. The test may be negated by prepending an exclamation mark.

ConditionHost=
ConditionHost= may be used to match against the hostname or machine ID of the host. This either takes a hostname string (optionally with shell style globs) which is tested against the locally set hostname as returned by **gethostname**(2), or a machine ID formatted as string (see **machine-id**(5)). The test may be negated by prepending an exclamation mark.

ConditionKernelCommandLine=
ConditionKernelCommandLine= may be used to check whether a specific kernel command line option is set (or if prefixed with the exclamation mark — unset). The argument must either be a single word, or an assignment (i.e. two words, separated by "="). In the former case the kernel command line is searched for the word appearing as is, or as left hand side of an assignment. In the latter case, the exact assignment is looked for with right and left hand side matching. This operates on the kernel command line communicated to userspace via /proc/cmdline, except when the service manager is invoked as payload of a container manager, in which case the command line of PID 1 is used instead (i.e. /proc/1/cmdline).

ConditionKernelVersion=
ConditionKernelVersion= may be used to check whether the kernel version (as reported by **uname −r**) matches a certain expression (or if prefixed with the exclamation mark does not match it). The argument must be a list of (potentially quoted) expressions. For each of the expressions, if it starts with one of "<", "<=", "=", "!=", ">=", ">" a relative version comparison is done, otherwise the specified string is matched with shell−style globs.

Note that using the kernel version string is an unreliable way to determine which features are supported by a kernel, because of the widespread practice of backporting drivers, features, and fixes from newer upstream kernels into older versions provided by distributions. Hence, this check is inherently unportable and should not be used for units which may be used on different distributions.

ConditionEnvironment=
ConditionEnvironment= may be used to check whether a specific environment variable is set (or if prefixed with the exclamation mark — unset) in the service manager's environment block. The argument may be a single word, to check if the variable with this name is defined in the environment block, or an assignment ("*name*=*value*"), to check if the variable with this exact value is defined. Note that the environment block of the service manager itself is checked, i.e. not any variables defined with *Environment=* or *EnvironmentFile=*, as described above. This is particularly useful when the service manager runs inside a containerized environment or as per−user service manager, in order to check for

variables passed in by the enclosing container manager or PAM.

ConditionSecurity=

ConditionSecurity= may be used to check whether the given security technology is enabled on the system. Currently, the recognized values are "selinux", "apparmor", "tomoyo", "ima", "smack", "audit", "uefi−secureboot" and "tpm2". The test may be negated by prepending an exclamation mark.

ConditionCapability=

Check whether the given capability exists in the capability bounding set of the service manager (i.e. this does not check whether capability is actually available in the permitted or effective sets, see **capabilities**(7) for details). Pass a capability name such as "CAP_MKNOD", possibly prefixed with an exclamation mark to negate the check.

ConditionACPower=

Check whether the system has AC power, or is exclusively battery powered at the time of activation of the unit. This takes a boolean argument. If set to "true", the condition will hold only if at least one AC connector of the system is connected to a power source, or if no AC connectors are known. Conversely, if set to "false", the condition will hold only if there is at least one AC connector known and all AC connectors are disconnected from a power source.

ConditionNeedsUpdate=

Takes one of /var/ or /etc/ as argument, possibly prefixed with a "!" (to invert the condition). This condition may be used to conditionalize units on whether the specified directory requires an update because /usr/'s modification time is newer than the stamp file .updated in the specified directory. This is useful to implement offline updates of the vendor operating system resources in /usr/ that require updating of /etc/ or /var/ on the next following boot. Units making use of this condition should order themselves before **systemd-update-done.service**(8), to make sure they run before the stamp file's modification time gets reset indicating a completed update.

If the *systemd.condition−needs−update=* option is specified on the kernel command line (taking a boolean), it will override the result of this condition check, taking precedence over any file modification time checks. If the kernel command line option is used, systemd−update−done.service will not have immediate effect on any following *ConditionNeedsUpdate=* checks, until the system is rebooted where the kernel command line option is not specified anymore.

Note that to make this scheme effective, the timestamp of /usr/ should be explicitly updated after its contents are modified. The kernel will automatically update modification timestamp on a directory only when immediate children of a directory are modified; an modification of nested files will not automatically result in mtime of /usr/ being updated.

Also note that if the update method includes a call to execute appropriate post−update steps itself, it should not touch the timestamp of /usr/. In a typical distribution packaging scheme, packages will do any required update steps as part of the installation or upgrade, to make package contents immediately usable. *ConditionNeedsUpdate=* should be used with other update mechanisms where such an immediate update does not happen.

ConditionFirstBoot=

Takes a boolean argument. This condition may be used to conditionalize units on whether the system is booting up for the first time. This roughly means that /etc/ is unpopulated (for details, see "First Boot Semantics" in **machine-id**(5)). This may be used to populate /etc/ on the first boot after factory reset, or when a new system instance boots up for the first time.

For robustness, units with *ConditionFirstBoot=yes* should order themselves before first−boot−complete.target and pull in this passive target with *Wants=*. This ensures that in a case of an aborted first boot, these units will be re−run during the next system startup.

If the *systemd.condition−first−boot=* option is specified on the kernel command line (taking a boolean), it will override the result of this condition check, taking precedence over /etc/machine−id existence checks.

ConditionPathExists=
Check for the existence of a file. If the specified absolute path name does not exist, the condition will fail. If the absolute path name passed to *ConditionPathExists=* is prefixed with an exclamation mark ("!"), the test is negated, and the unit is only started if the path does not exist.

ConditionPathExistsGlob=
ConditionPathExistsGlob= is similar to *ConditionPathExists=*, but checks for the existence of at least one file or directory matching the specified globbing pattern.

ConditionPathIsDirectory=
ConditionPathIsDirectory= is similar to *ConditionPathExists=* but verifies that a certain path exists and is a directory.

ConditionPathIsSymbolicLink=
ConditionPathIsSymbolicLink= is similar to *ConditionPathExists=* but verifies that a certain path exists and is a symbolic link.

ConditionPathIsMountPoint=
ConditionPathIsMountPoint= is similar to *ConditionPathExists=* but verifies that a certain path exists and is a mount point.

ConditionPathIsReadWrite=
ConditionPathIsReadWrite= is similar to *ConditionPathExists=* but verifies that the underlying file system is readable and writable (i.e. not mounted read−only).

ConditionPathIsEncrypted=
ConditionPathIsEncrypted= is similar to *ConditionPathExists=* but verifies that the underlying file system's backing block device is encrypted using dm−crypt/LUKS. Note that this check does not cover ext4 per−directory encryption, and only detects block level encryption. Moreover, if the specified path resides on a file system on top of a loopback block device, only encryption above the loopback device is detected. It is not detected whether the file system backing the loopback block device is encrypted.

ConditionDirectoryNotEmpty=
ConditionDirectoryNotEmpty= is similar to *ConditionPathExists=* but verifies that a certain path exists and is a non−empty directory.

ConditionFileNotEmpty=
ConditionFileNotEmpty= is similar to *ConditionPathExists=* but verifies that a certain path exists and refers to a regular file with a non−zero size.

ConditionFileIsExecutable=
ConditionFileIsExecutable= is similar to *ConditionPathExists=* but verifies that a certain path exists, is a regular file, and marked executable.

ConditionUser=
ConditionUser= takes a numeric "UID", a UNIX user name, or the special value "@system". This condition may be used to check whether the service manager is running as the given user. The special value "@system" can be used to check if the user id is within the system user range. This option is not useful for system services, as the system manager exclusively runs as the root user, and thus the test result is constant.

ConditionGroup=
ConditionGroup= is similar to *ConditionUser=* but verifies that the service manager's real or effective group, or any of its auxiliary groups, match the specified group or GID. This setting does not support the special value "@system".

ConditionControlGroupController=
Check whether given cgroup controllers (e.g. "cpu") are available for use on the system or whether the legacy v1 cgroup or the modern v2 cgroup hierarchy is used.

Multiple controllers may be passed with a space separating them; in this case the condition will only pass if all listed controllers are available for use. Controllers unknown to systemd are ignored. Valid controllers are "cpu", "cpuacct", "io", "blkio", "memory", "devices", and "pids". Even if available in the kernel, a particular controller may not be available if it was disabled on the kernel command line with *cgroup_disable=controller*.

Alternatively, two special strings "v1" and "v2" may be specified (without any controller names). "v2" will pass if the unified v2 cgroup hierarchy is used, and "v1" will pass if the legacy v1 hierarchy or the hybrid hierarchy are used (see the discussion of *systemd.unified_cgroup_hierarchy* and *systemd.legacy_systemd_cgroup_controller* in **systemd.service**(5) for more information).

ConditionMemory=

Verify that the specified amount of system memory is available to the current system. Takes a memory size in bytes as argument, optionally prefixed with a comparison operator "<", "<=", "=", "!=", ">=", ">". On bare−metal systems compares the amount of physical memory in the system with the specified size, adhering to the specified comparison operator. In containers compares the amount of memory assigned to the container instead.

ConditionCPUs=

Verify that the specified number of CPUs is available to the current system. Takes a number of CPUs as argument, optionally prefixed with a comparison operator "<", "<=", "=", "!=", ">=", ">". Compares the number of CPUs in the CPU affinity mask configured of the service manager itself with the specified number, adhering to the specified comparison operator. On physical systems the number of CPUs in the affinity mask of the service manager usually matches the number of physical CPUs, but in special and virtual environments might differ. In particular, in containers the affinity mask usually matches the number of CPUs assigned to the container and not the physically available ones.

ConditionCPUFeature=

Verify that a given CPU feature is available via the "CPUID" instruction. This condition only does something on i386 and x86−64 processors. On other processors it is assumed that the CPU does not support the given feature. It checks the leaves "1", "7", "0x80000001", and "0x80000007". Valid values are: "fpu", "vme", "de", "pse", "tsc", "msr", "pae", "mce", "cx8", "apic", "sep", "mtrr", "pge", "mca", "cmov", "pat", "pse36", "clflush", "mmx", "fxsr", "sse", "sse2", "ht", "pni", "pclmul", "monitor", "ssse3", "fma3", "cx16", "sse4_1", "sse4_2", "movbe", "popcnt", "aes", "xsave", "osxsave", "avx", "f16c", "rdrand", "bmi1", "avx2", "bmi2", "rdseed", "adx", "sha_ni", "syscall", "rdtscp", "lm", "lahf_lm", "abm", "constant_tsc".

ConditionOSRelease=

Verify that a specific "key=value" pair is set in the host's **os-release**(5).

Other than exact matching with "=", and "!=", relative comparisons are supported for versioned parameters (e.g. "VERSION_ID"). The comparator can be one of "<", "<=", "=", "!=", ">=" and ">".

AssertArchitecture=, *AssertVirtualization=*, *AssertHost=*, *AssertKernelCommandLine=*, *AssertKernelVersion=*, *AssertEnvironment=*, *AssertSecurity=*, *AssertCapability=*, *AssertACPower=*, *AssertNeedsUpdate=*, *AssertFirstBoot=*, *AssertPathExists=*, *AssertPathExistsGlob=*, *AssertPathIsDirectory=*, *AssertPathIsSymbolicLink=*, *AssertPathIsMountPoint=*, *AssertPathIsReadWrite=*, *AssertPathIsEncrypted=*, *AssertDirectoryNotEmpty=*, *AssertFileNotEmpty=*, *AssertFileIsExecutable=*, *AssertUser=*, *AssertGroup=*, *AssertControlGroupController=*, *AssertMemory=*, *AssertCPUs=*, *AssertOSRelease=*

Similar to the *ConditionArchitecture=*, *ConditionVirtualization=*, ..., condition settings described above, these settings add assertion checks to the start−up of the unit. However, unlike the conditions settings, any assertion setting that is not met results in failure of the start job (which means this is logged loudly). Note that hitting a configured assertion does not cause the unit to enter the "failed" state (or in fact result in any state change of the unit), it affects only the job queued for it. Use assertion expressions for units that cannot operate when specific requirements are not met, and when this is something the administrator or user should look into.

MAPPING OF UNIT PROPERTIES TO THEIR INVERSES

Unit settings that create a relationship with a second unit usually show up in properties of both units, for example in **systemctl show** output. In some cases the name of the property is the same as the name of the configuration setting, but not always. This table lists the properties that are shown on two units which are connected through some dependency, and shows which property on "source" unit corresponds to which property on the "target" unit.

Table 3. Forward and reverse unit properties

"Forward" property	"Reverse" property	Where used	
Before=	*After=*	[Unit] section	
After=	*Before=*		
Requires=	*RequiredBy=*	[Unit] section	[Install] section
Wants=	*WantedBy=*	[Unit] section	[Install] section
PartOf=	*ConsistsOf=*	[Unit] section	an automatic property
BindsTo=	*BoundBy=*	[Unit] section	an automatic property
Requisite=	*RequisiteOf=*	[Unit] section	an automatic property
Triggers=	*TriggeredBy=*	Automatic properties, see notes below	
Conflicts=	*ConflictedBy=*	[Unit] section	an automatic property
PropagatesReloadTo=	*ReloadPropagatedFrom=*	[Unit] section	
ReloadPropagatedFrom=	*PropagatesReloadTo=*		
Following=	n/a	An automatic property	

Note: *WantedBy=* and *RequiredBy=* are used in the [Install] section to create symlinks in .wants/ and .requires/ directories. They cannot be used directly as a unit configuration setting.

Note: *ConsistsOf=*, *BoundBy=*, *RequisiteOf=*, *ConflictedBy=* are created implicitly along with their reverses and cannot be specified directly.

Note: *Triggers=* is created implicitly between a socket, path unit, or an automount unit, and the unit they activate. By default a unit with the same name is triggered, but this can be overridden using *Sockets=*, *Service=*, and *Unit=* settings. See **systemd.service**(5), **systemd.socket**(5), **systemd.path**(5), and **systemd.automount**(5) for details. *TriggeredBy=* is created implicitly on the triggered unit.

Note: *Following=* is used to group device aliases and points to the "primary" device unit that systemd is using to track device state, usually corresponding to a sysfs path. It does not show up in the "target" unit.

[INSTALL] SECTION OPTIONS

Unit files may include an [Install] section, which carries installation information for the unit. This section is not interpreted by **systemd**(1) during runtime; it is used by the **enable** and **disable** commands of the **systemctl**(1) tool during installation of a unit.

Alias=

A space−separated list of additional names this unit shall be installed under. The names listed here must have the same suffix (i.e. type) as the unit filename. This option may be specified more than once, in which case all listed names are used. At installation time, **systemctl enable** will create symlinks from these names to the unit filename. Note that not all unit types support such alias names, and this setting is not supported for them. Specifically, mount, slice, swap, and automount units do not support aliasing.

WantedBy=, *RequiredBy=*

This option may be used more than once, or a space−separated list of unit names may be given. A symbolic link is created in the .wants/ or .requires/ directory of each of the listed units when this unit is installed by **systemctl enable**. This has the effect that a dependency of type *Wants=* or *Requires=* is added from the listed unit to the current unit. The primary result is that the current unit will be started when the listed unit is started. See the description of *Wants=* and *Requires=* in the [Unit] section for details.

WantedBy=foo.service in a service bar.service is mostly equivalent to **Alias=foo.service.wants/bar.service** in the same file. In case of template units, **systemctl enable** must be called with an instance name, and this instance will be added to the .wants/ or .requires/ list of the listed unit. E.g. **WantedBy=getty.target** in a service getty@.service will result in **systemctl enable getty@tty2.service** creating a getty.target.wants/getty@tty2.service link to getty@.service.

Also=

Additional units to install/deinstall when this unit is installed/deinstalled. If the user requests installation/deinstallation of a unit with this option configured, **systemctl enable** and **systemctl disable** will automatically install/uninstall units listed in this option as well.

This option may be used more than once, or a space−separated list of unit names may be given.

DefaultInstance=

In template unit files, this specifies for which instance the unit shall be enabled if the template is enabled without any explicitly set instance. This option has no effect in non−template unit files. The specified string must be usable as instance identifier.

The following specifiers are interpreted in the Install section: %a, %b, %B, %g, %G, %H, %i, %j, %l, %m, %n, %N, %o, %p, %u, %U, %v, %w, %W, %%. For their meaning see the next section.

SPECIFIERS

Many settings resolve specifiers which may be used to write generic unit files referring to runtime or unit parameters that are replaced when the unit files are loaded. Specifiers must be known and resolvable for the setting to be valid. The following specifiers are understood:

Table 4. Specifiers available in unit files

EXAMPLES

Example 1. Allowing units to be enabled

The following snippet (highlighted) allows a unit (e.g. foo.service) to be enabled via **systemctl enable**:

```
[Unit]
Description=Foo

[Service]
ExecStart=/usr/sbin/foo−daemon

[Install]
WantedBy=multi−user.target
```

After running **systemctl enable**, a symlink /etc/systemd/system/multi−user.target.wants/foo.service linking to the actual unit will be created. It tells systemd to pull in the unit when starting multi−user.target. The inverse **systemctl disable** will remove that symlink again.

Example 2. Overriding vendor settings

There are two methods of overriding vendor settings in unit files: copying the unit file from /lib/systemd/system to /etc/systemd/system and modifying the chosen settings. Alternatively, one can create a directory named *unit*.d/ within /etc/systemd/system and place a drop−in file *name*.conf there that only changes the specific settings one is interested in. Note that multiple such drop−in files are read if present, processed in lexicographic order of their filename.

The advantage of the first method is that one easily overrides the complete unit, the vendor unit is not parsed at all anymore. It has the disadvantage that improvements to the unit file by the vendor are not automatically incorporated on updates.

The advantage of the second method is that one only overrides the settings one specifically wants, where updates to the unit by the vendor automatically apply. This has the disadvantage that some future updates by the vendor might be incompatible with the local changes.

This also applies for user instances of systemd, but with different locations for the unit files. See the section on unit load paths for further details.

Suppose there is a vendor−supplied unit /lib/systemd/system/httpd.service with the following contents:

```
[Unit]
Description=Some HTTP server
After=remote−fs.target sqldb.service
Requires=sqldb.service
AssertPathExists=/srv/webserver

[Service]
Type=notify
ExecStart=/usr/sbin/some−fancy−httpd−server
Nice=5

[Install]
WantedBy=multi−user.target
```

Now one wants to change some settings as an administrator: firstly, in the local setup, /srv/webserver might not exist, because the HTTP server is configured to use /srv/www instead. Secondly, the local configuration makes the HTTP server also depend on a memory cache service, memcached.service, that should be pulled in (*Requires=*) and also be ordered appropriately (*After=*). Thirdly, in order to harden the service a bit more, the administrator would like to set the *PrivateTmp=* setting (see **systemd.exec**(5) for details). And lastly, the administrator would like to reset the niceness of the service to its default value of 0.

The first possibility is to copy the unit file to /etc/systemd/system/httpd.service and change the chosen settings:

```
[Unit]
Description=Some HTTP server
After=remote−fs.target sqldb.service memcached.service
Requires=sqldb.service memcached.service
AssertPathExists=/srv/www

[Service]
Type=notify
ExecStart=/usr/sbin/some−fancy−httpd−server
Nice=0
PrivateTmp=yes

[Install]
WantedBy=multi−user.target
```

Alternatively, the administrator could create a drop−in file /etc/systemd/system/httpd.service.d/local.conf with the following contents:

```
[Unit]
After=memcached.service
Requires=memcached.service
# Reset all assertions and then re−add the condition we want
AssertPathExists=
AssertPathExists=/srv/www

[Service]
Nice=0
PrivateTmp=yes
```

Note that for drop−in files, if one wants to remove entries from a setting that is parsed as a list (and is not a dependency), such as *AssertPathExists=* (or e.g. *ExecStart=* in service units), one needs to first clear the list before re−adding all entries except the one that is to be removed. Dependencies (*After=*, etc.) cannot be reset to an empty list, so dependencies can only be added in drop−ins. If you want to remove dependencies, you have to override the entire unit.

Example 3. Top level drop−ins with template units

Top level per−type drop−ins can be used to change some aspect of all units of a particular type. For example by creating the /etc/systemd/system/service.d/ directory with a drop−in file, the contents of the drop−in file can be applied to all service units. We can take this further by having the top−level drop−in instantiate a secondary helper unit. Consider for example the following set of units and drop−in files where we install an *OnFailure=* dependency for all service units.

/etc/systemd/system/failure−handler@.service:

```
[Unit]
Description=My failure handler for %i

[Service]
Type=oneshot
# Perform some special action for when %i exits unexpectedly.
ExecStart=/usr/sbin/myfailurehandler %i
```

We can then add an instance of failure−handler@.service as an *OnFailure=* dependency for all service units.

```
/etc/systemd/system/service.d/10−all.conf:

[Unit]
OnFailure=failure−handler@%N.service
```

Now, after running **systemctl daemon−reload** all services will have acquired an *OnFailure=* dependency on failure−handler@%N.service. The template instance units will also have gained the dependency which results in the creation of a recursive dependency chain. We can break the chain by disabling the drop−in for the template instance units via a symlink to /dev/null:

```
mkdir /etc/systemd/system/failure−handler@.service.d/
ln −s /dev/null /etc/systemd/system/failure−handler@.service.d/10−all.conf
systemctl daemon−reload
```

This ensures that if a failure−handler@.service instance fails it will not trigger an instance named failure−handler@failure−handler.service.

SEE ALSO

systemd(1), **systemctl**(1), **systemd-system.conf**(5), **systemd.special**(7), **systemd.service**(5), **systemd.socket**(5), **systemd.device**(5), **systemd.mount**(5), **systemd.automount**(5), **systemd.swap**(5), **systemd.target**(5), **systemd.path**(5), **systemd.timer**(5), **systemd.scope**(5), **systemd.slice**(5), **systemd.time**(7), **systemd-analyze**(1), **capabilities**(7), **systemd.directives**(7), **uname**(1)

NOTES

1. Interface Portability and Stability Promise
   https://systemd.io/PORTABILITY_AND_STABILITY/