# sd_notify


- Description
- Well-known assignments
- Return Value
  - Errors
- Notes
  - Standalone Implementations
    - C
    - Python
- Environment
- Examples
- History
- See Also


systemd

sd_notify

3

sd_notify

sd_notifyf

sd_pid_notify

sd_pid_notifyf

sd_pid_notify_with_fds

sd_pid_notifyf_with_fds

sd_notify_barrier

sd_pid_notify_barrier

Notify service manager about start-up completion and other service status changes

#include <systemd/sd-daemon.h>

int

sd_notify

int

unset_environment

const char *

state

int

sd_notifyf

int

unset_environment

const char *

format

…

int

sd_pid_notify

pid_t

pid

int

unset_environment

const char *

state

int

sd_pid_notifyf

pid_t

pid

int

unset_environment

const char *

format

...

int

sd_pid_notify_with_fds

pid_t

pid

int

unset_environment

const char *

state

const int *

fds

unsigned

n_fds

int

sd_pid_notifyf_with_fds

pid_t

pid

int

unset_environment

const int *

fds

size_t

n_fds

const char *

format

…

int

sd_notify_barrier

int

unset_environment

uint64_t

timeout

int

sd_pid_notify_barrier

pid_t

pid

int

unset_environment

uint64_t

timeout

# Description

`sd_notify()` may be called by a service to notify the service manager about state changes. It can be used to send arbitrary information, encoded in an environment-block-like string. Most importantly, it can be used for start-up or reload completion notifications.

If the `unset_environment` parameter is non-zero, `sd_notify()` will unset the `$NOTIFY_SOCKET` environment variable before returning (regardless of whether the function call itself succeeded or not). Further calls to `sd_notify()` will then silently do nothing, and the variable is no longer inherited by child processes.

The `state` parameter should contain a newline-separated list of variable assignments, similar in style to an environment block. A trailing newline is implied if none is specified. The string may contain any kind of variable assignments, but see the next section for a list of assignments understood by the service manager.

Note that systemd will accept status data sent from a service only if the `NotifyAccess=` option is correctly set in the service definition file. See `systemd.service(5)` for details.

Note that `sd_notify()` notifications may be attributed to units correctly only if either the sending process is still around at the time PID 1 processes the message, or if the sending process is explicitly runtime-tracked by the service manager. The latter is the case if the service manager originally forked off the process, i.e. on all processes that match `NotifyAccess=``main` or `NotifyAccess=``exec`. Conversely, if an auxiliary process of the unit sends an `sd_notify()` message and immediately exits, the service manager might not be able to properly attribute the message to the unit, and thus will ignore it, even if `NotifyAccess=``all` is set for it.

Hence, to eliminate all race conditions involving lookup of the client's unit and attribution of notifications to units correctly, `sd_notify_barrier()` may be used. This call acts as a synchronization point and ensures all notifications sent before this call have been picked up by the service manager when it returns successfully. Use of `sd_notify_barrier()` is needed for clients which are not invoked by the service manager, otherwise this synchronization mechanism is unnecessary for attribution of notifications to the unit.

`sd_notifyf()` is similar to `sd_notify()` but takes a `printf()`-like format string plus arguments.

`sd_pid_notify()` and `sd_pid_notifyf()` are similar to `sd_notify()` and `sd_notifyf()` but take a process ID (PID) to use as originating PID for the message as first argument. This is useful to send notification messages on behalf of other processes, provided the appropriate privileges are available. Effectively, this means that a privileged invocation of `sd_pid_notify()` may circumvent `NotifyAccess=main` or `NotifyAccess=exec` restrictions enforced for a service. If the PID argument is specified as 0, the process ID of the calling process is used, in which case the calls are fully equivalent to `sd_notify()` and `sd_notifyf()`.

`sd_pid_notify_with_fds()` is similar to `sd_pid_notify()` but takes an additional array of file descriptors. These file descriptors are sent along the notification message to the service manager. This is particularly useful for sending `FDSTORE=1` messages, as described above. The additional arguments are a pointer to the file descriptor array plus the number of file descriptors in the array. If the number of file descriptors is passed as 0, the call is fully equivalent to `sd_pid_notify()`, i.e. no file descriptors are passed. Note that file descriptors sent to the service manager on a message without `FDSTORE=1` are immediately closed on reception.

`sd_pid_notifyf_with_fds()` is a combination of `sd_pid_notify_with_fds()` and `sd_notifyf()`, i.e. it accepts both a PID and a set of file descriptors as input, and processes a format string to generate the state string.

`sd_notify_barrier()` allows the caller to synchronize against reception of previously sent notification messages and uses the `BARRIER=1` command. It takes a relative `timeout` value in microseconds which is passed to `ppoll(2)`. A value of UINT64_MAX is interpreted as infinite timeout.

`sd_pid_notify_barrier()` is just like `sd_notify_barrier()`, but allows specifying the originating PID for the notification message.

# Well-known assignments

The following assignments have a defined meaning:

READY=1
Tells the service manager that service startup is finished, or the service finished re-loading its configuration. This is only used by systemd if the service definition file has `Type=notify` or `Type=notify-reload` set. Since there is little value in signaling non-readiness, the only value services should send is `READY=1` (i.e. `READY=0` is not defined).

RELOADING=1
Tells the service manager that the service is beginning to reload its configuration. This is useful to allow the service manager to track the service's internal state, and present it to the user. Note that a service that sends this notification must also send a `READY=1` notification when it completed reloading its configuration. Reloads the service manager is notified about with this mechanisms are propagated in the same way as they are when originally initiated through the service manager. This message is particularly relevant for `Type=notify-reload` services, to inform the service manager that the request to reload the service has been received and is now being processed.

STOPPING=1
Tells the service manager that the service is beginning its shutdown. This is useful to allow the service manager to track the service's internal state, and present it to the user.

Starting with version 259, transitioning to the STOPPING state is final and cannot be undone with another notification. If after sending `STOPPING=1` the service doesn't terminate within the stop timeout, all remaining processes in the unit will be killed.

MONOTONIC_USEC=…
A field carrying the monotonic timestamp (as per `CLOCK_MONOTONIC`) formatted in decimal in µs, when the notification message was generated by the client. This is typically used in combination with `RELOADING=1`, to allow the service manager to properly synchronize reload cycles. See `systemd.service(5)` for details, specifically `Type=notify-reload`.

STATUS=…
Passes a single-line UTF-8 status string back to the service manager that describes the service state. This is free-form and can be used for various purposes: general state feedback, fsck-like programs could pass completion percentages and failing programs could pass a human-readable error message. Example: `STATUS=Completed 66% of file system check…`

NOTIFYACCESS=…
Reset the access to the service status notification socket during runtime, overriding `NotifyAccess=` setting in the service unit file. See `systemd.service(5)` for details, specifically `NotifyAccess=` for a list of accepted values.

ERRNO=…
If a service fails, the errno-style error code, formatted as string. Example: `ERRNO=2` for ENOENT.

BUSERROR=…
If a service fails, the D-Bus error-style error code. Example: `BUSERROR=org.freedesktop.DBus.Error.TimedOut`.

VARLINKERROR=…
If a service fails, the Varlink error-style error code. Example: `VARLINKERROR=org.varlink.service.InvalidParameter`.

EXIT_STATUS=…
The exit status of a service or the manager itself. Note that `systemd` currently does not consume this value when sent by services, so this assignment is only informational. The manager will send this notification to *its* notification socket, which may be used to collect an exit status from the system (a container or VM) as it shuts down. For example, `mkosi(1)` makes use of this. The value to return may be set via the `systemctl(1)` `exit` verb.

MAINPID=…
Change the main process ID (PID) of the service. This is especially useful in the case where the real main process is not directly forked off by the service manager. Example: `MAINPID=4711`.

MAINPIDFDID=…
The pidfd inode number of the new main process (specified through `MAINPID=`). This information can be acquired through `sd_pidfd_get_inode_id(3)` on the pidfd and is used to identify the process in a race-free fashion. Alternatively, a pidfd can be sent directly to the service manager (see

`MAINPIDFD=1` below).

MAINPIDFD=1
Similar to `MAINPID=` with `MAINPIDFDID=`, but the process is referenced directly by the pidfd passed to the service manager. This is useful if pidfd id is not supported on the system. Exactly one fd is expected for this notification.

WATCHDOG=1
Tells the service manager to update the watchdog timestamp. This is the keep-alive ping that services need to issue in regular intervals if `WatchdogSec=` is enabled for it. See `systemd.service(5)` for information how to enable this functionality and `sd_watchdog_enabled(3)` for the details of how the service can check whether the watchdog is enabled.

WATCHDOG=trigger
Tells the service manager that the service detected an internal error that should be handled by the configured watchdog options. This will trigger the same behaviour as if `WatchdogSec=` is enabled and the service did not send `WATCHDOG=1` in time. Note that `WatchdogSec=` does not need to be enabled for `WATCHDOG=trigger` to trigger the watchdog action. See `systemd.service(5)` for information about the watchdog behavior.

WATCHDOG_USEC=...
Reset `watchdog_usec` value during runtime. Notice that this is not available when using `sd_event_set_watchdog()` or `sd_watchdog_enabled()`. Example : `WATCHDOG_USEC=20000000`

EXTEND_TIMEOUT_USEC=...
Tells the service manager to extend the startup, runtime or shutdown service timeout corresponding the current state. The value specified is a time in microseconds during which the service must send a new message. A service timeout will occur if the message is not received, but only if the runtime of the current state is beyond the original maximum times of `TimeoutStartSec=`, `RuntimeMaxSec=`, and `TimeoutStopSec=`. See `systemd.service(5)` for effects on the service timeouts.

RESTART_RESET=1
Reset the restart counter of the service, which has the effect of restoring the restart duration to `RestartSec=` if `RestartSteps=` and `RestartMaxDelaySec=` are in use. For more information, refer to `systemd.service(5)`.

FDSTORE=1
Store file descriptors in the service manager. File descriptors sent this way will be held for the service by the service manager and will later be handed back using the usual file descriptor passing logic at the next start or restart of the service, see `sd_listen_fds(3)`. Any open sockets and other file descriptors which should not be closed during a restart may be stored this way. When a service is stopped, its file descriptor store is discarded and all file descriptors in it are closed, except when overridden with `FileDescriptorStorePreserve=`, see `systemd.service(5)`.

The service manager will accept messages for a service only if its `FileDescriptorStoreMax=` setting is non-zero (defaults to zero, see `systemd.service(5)`). The service manager will set the `$FDSTORE` environment variable for services that have the file descriptor store enabled, see `systemd.exec(5)`.

If `FDPOLL=0` is not set and the file descriptors are pollable (see `epoll_ctl(2)`), then any `EPOLLHUP` or `EPOLLERR` event seen on them will result in their automatic removal from the store.

Multiple sets of file descriptors may be sent in separate messages, in which case the sets are combined. The service manager removes duplicate file descriptors (those pointing to the same object) before passing them to the service.

This functionality should be used to implement services that can restart after an explicit request or a crash without losing state. Application state can either be serialized to a file in `/run/`, or better, stored in a `memfd_create(2)` memory file descriptor. Use `sd_pid_notify_with_fds()` to send messages with `FDSTORE=1`. It is recommended to combine `FDSTORE=` with `FDNAME=` to make it easier to manage the stored file descriptors.

For further information on the file descriptor store see the File Descriptor Store overview.

On kernels that support the Live Update Orchestrator, compatible file descriptors stored this way (such as `memfd_create(2)`) are additionally preserved across `kexec`-based reboots and handed back to the service on the other side, provided `FileDescriptorStorePreserve=yes` is set on the service (see `systemd.service(5)`). See the File Descriptor Store overview for details.

FDSTOREREMOVE=1
Removes file descriptors from the file descriptor store. This field needs to be combined with `FDNAME=` to specify the name of the file descriptors to remove.

FDNAME=…
When used in combination with `FDSTORE=1`, specifies a name for the submitted file descriptors. When used with `FDSTOREREMOVE=1`, specifies the name for the file descriptors to remove. This name is passed to the service during activation, and may be queried using `sd_listen_fds_with_names(3)`. File descriptors submitted without this field will be called `stored`.

The name may consist of arbitrary ASCII characters except control characters or `:`. It may not be longer than 255 characters. If a submitted name does not follow these restrictions, it is ignored.

Note that if multiple file descriptors are submitted in a single message, the specified name will be used for all of them. In order to assign different names to submitted file descriptors, submit them in separate messages.

FDPOLL=0
When used in combination with `FDSTORE=1`, disables polling of the submitted file descriptors regardless of whether or not they are pollable. As this option disables automatic cleanup of the submitted file descriptors on EPOLLERR and EPOLLHUP, care must be taken to ensure proper manual cleanup. Use of this option is not generally recommended except for when automatic cleanup has unwanted behavior such as prematurely discarding file descriptors from the store.

BARRIER=1
Tells the service manager that the client is explicitly requesting synchronization by means of

closing the file descriptor sent with this command. The service manager guarantees that the processing of a `BARRIER=1` command will only happen after all previous notification messages sent before this command have been processed. Hence, this command accompanied with a single file descriptor can be used to synchronize against reception of all previous status messages. Note that this command cannot be mixed with other notifications, and has to be sent in a separate message to the service manager, otherwise all assignments will be ignored. Note that sending 0 or more than 1 file descriptor with this command is a violation of the protocol.

The notification messages sent by services are interpreted by the service manager. Unknown assignments are ignored. Thus, it is safe (but often without effect) to send assignments which are not in this list. The protocol is extensible, but care should be taken to ensure private extensions are recognizable as such. Specifically, it is recommend to prefix them with `X_` followed by some namespace identifier. The service manager also sends some messages to *its* notification socket, which may then consumed by a supervising machine or container manager further up the stack. The service manager sends a number of extension fields, for example `X_SYSTEMD_UNIT_ACTIVE=`, for details see `systemd(1)`.

# Return Value

On failure, these calls return a negative errno-style error code. If `$NOTIFY_SOCKET` was not set and hence no status message could be sent, 0 is returned. If the status was sent, these functions return a positive value. In order to support both service managers that implement this scheme and those which do not, it is generally recommended to ignore the return value of this call. Note that the return value simply indicates whether the notification message was enqueued properly, it does not reflect whether the message could be processed successfully. Specifically, no error is returned when a file descriptor is attempted to be stored using `FDSTORE=1` but the service is not actually configured to permit storing of file descriptors (see above).

# Errors

Returned errors may indicate the following problems:

`-E2BIG`
More file descriptors passed at once than the system allows. On Linux the number of file descriptors that may be passed across `AF_UNIX` sockets at once is 253, see `unix(7)` for details.

# Notes

These functions send a single datagram with the state string as payload to the socket referenced in the `$NOTIFY_SOCKET` environment variable. If the first character of `$NOTIFY_SOCKET` is `/` or `@`, the string is understood as an `AF_UNIX` or Linux abstract namespace socket (respectively), and in both cases the datagram is accompanied by the process credentials of the sending service, using SCM_CREDENTIALS. If the string starts with `vsock:` then the string is understood as an `AF_VSOCK` address, which is useful for hypervisors/VMMs or other processes on the host to receive a notification when a virtual machine has finished booting. Note that in case the hypervisor does not support `SOCK_DGRAM` over `AF_VSOCK`, `SOCK_SEQPACKET` will be used instead. `vsock-stream`, `vsock-dgram` and `vsock-seqpacket` can be used instead of `vsock` to force usage of the corresponding socket type. The address should be in the form: `vsock:CID:PORT`. Note that unlike other uses of vsock, the CID is mandatory and cannot be `VMADDR_CID_ANY`. Note that PID1 will send the VSOCK packets from a privileged port (i.e.: lower than 1024), as an attempt to address concerns that unprivileged processes in the guest might try to send malicious notifications to the host, driving it to make destructive decisions based on them.

# Standalone Implementations

Note that, while using this library should be preferred in order to avoid code duplication, it is also possible to reimplement the simple readiness notification protocol without external dependencies, as demonstrated in the following self-contained examples from several languages:

## C

## Python

# Environment

`$NOTIFY_SOCKET`
Set by the service manager for supervised processes for status and start-up completion notification. This environment variable specifies the socket `sd_notify()` talks to. See above for details.

# Examples

Start-up Notification

When a service finished starting up, it might issue the following call to notify the service manager:

```
sd_notify(0, "READY=1");
```

Extended Start-up Notification

A service could send the following after completing initialization:

```
sd_notifyf(0, "READY=1\n"
              "STATUS=Processing requests…\n"
              "MAINPID=%lu",
           (unsigned long) getpid());
```

Error Cause Notification

A service could send the following shortly before exiting, on failure:

```
sd_notifyf(0, "STATUS=Failed to start up: %s\n"
              "ERRNO=%i",
           strerror_r(errnum, (char[1024]){}, 1024),
           errnum);
```

Store a File Descriptor in the Service Manager

To store an open file descriptor in the service manager, in order to continue operation after a service restart without losing state, use `FDSTORE=1`:

```
sd_pid_notify_with_fds(0, 0, "FDSTORE=1\nFDNAME=foobar", &fd, 1);
```

Eliminating race conditions

When the client sending the notifications is not spawned by the service manager, it may exit too quickly and the service manager may fail to attribute them correctly to the unit. To prevent such races, use `sd_notify_barrier()` to synchronize against reception of all notifications sent before this call is made.

```
sd_notify(0, "READY=1");
/* set timeout to 5 seconds */
sd_notify_barrier(0, 5 * 1000000);
```

# History

`sd_pid_notify()`, `sd_pid_notifyf()`, and `sd_pid_notify_with_fds()` were added in version 219.

`sd_notify_barrier()` was added in version 246.

`sd_pid_notifyf_with_fds()` and `sd_pid_notify_barrier()` were added in version 254.

# See Also

`systemd(1)`, `sd-daemon(3)`, `sd_listen_fds(3)`, `sd_listen_fds_with_names(3)`, `sd_watchdog_enabled(3)`, `daemon(7)`, `systemd.service(5)`

---

□□□ #2
□ □□□ □□□ 2026-10-31 14:58:16 UTC
□ □□□ □□□ 2026-10-31 14:58:17 UTC